MQSeries Everyplace Configuration Guide Version 1.1

June 2002

Barry Aldred
IBM Corporation
Hursley Park
Winchester
UK SO21 2JN

barry_aldred@uk.ibm.com

Take Note!

Before using this report be sure to read the general information under "Appendix A: Notices" on page 133.

License warning

MQSeries Everyplace – Configuration Guide version 1.1 is supplied under the terms of the International Program License Agreement. Defect correction will be provided under that agreement for users holding valid MQSeries Everyplace deployment license(s) until the end of service date, June 30, 2003.

Please refer to *http://www.ibm.com/software/mqseries* for details of the license conditions pertaining to the MQSeries Everyplace product.

Second Edition, June 2002

This edition applies to Version 1.1 of *MQSeries Everyplace – Configuration guide* and to all subsequent releases and modifications unless otherwise indicated in new editions.

Table of contents

Figures

Examples

Notices

References in this report to IBM products or programs do not imply that IBM intends to make these available in all countries in which IBM operates.

Information contained in this SupportPac has not been submitted to any formal IBM test and is distributed "AS-IS". The use of this information and the implementation of any of the techniques is the responsibility of the reader. Much depends on the ability of the reader to evaluate these data and project the results to their operational environment.

Trademarks and service marks

The following terms, used in this publication, are trademarks of the IBM Corporation in the United States or other countries or both:

- IBM
- MQSeries
- MQSeries Everyplace
- MQe
- Websphere

The following terms are trademarks of other companies:

- Windows 98, Windows NT, Windows 2000, Windows XP – Microsoft Corporation

Summary of amendments

Date	Changes
18 January 2002	Version 1.0 (Initial release)
20 June 2002	Version 1.1 Addition of class requirements. Examples upgraded to MQe_Explorer v 1.27. More detail included on the registry and the serving of mini-certificates. Naming rules added. Miscellaneous minor changes and additions

Preface

This book describes how to configure MQSeries Everyplace queue managers and networks. It complements the existing product publications and contains code samples and configuration data appropriate to common usage scenarios.

The samples and illustrations relate to MQSeries Everyplace version 1.27 and its associated management tool MQSeries Everyplace MQe_Explorer version 1.27. Later versions of these products may exhibit minor differences.

Bibliography

- *MQSeries Everyplace Version 1.2: Introduction,* IBM Corporation, SC34-5843
- *MQSeries Everyplace Version 1.2: Java Programming Guide,* IBM Corporation, SC34-5845
- *MQSeries Everyplace Version 1.2: Java Programming Reference,* IBM Corporation, SC34-5846
- *Websphere MQ Everyplace SupportPac ED02: Using MQSeries Everyplace with WebSphere Everyplace Server*
- *Websphere MQ Everyplace SupportPac ES02: MQSeries Everyplace – MQe_Explorer*
- *Websphere MQ Everyplace SupportPac ES03: MQSeries Everyplace – WTLS Mini-Certificate Server*
- *Websphere MQ SupportPac MA88: MQSeries Classes for Java and MQSeries Classes for Java Message Service*

Related material

- *Websphere MQ Everyplace SupportPac ED01: MQSeries Everyplace - Get Started*
- *Websphere MQ Everyplace SupportPac EA01: MQSeries Everyplace - XML conversion utility*
- *Websphere MQ Everyplace SupportPac EAP1: MQSeries Everyplace - Device code for Palm OS*
- *Websphere MQ Everyplace SupportPac EP01 MQSeries Everyplace – Performance report*
- *Websphere MQ Integrator SupportPac ID03: MQSeries Integrator – Working with MQSeries Everyplace*

Web references

The following URLs provide useful resources for both MQSeries Everyplace and MQe_Explorer:

Download sites

IBM WebSphere MQ SupportPacs:

http://www.ibm.com/software/mqseries/txppacs/

IBM Boulder (MQSeries Everyplace product code downloads):

http://www6.software.ibm.com/dl/mqsem/mqsem-p

IBM Visual Age Micro Edition (Java stacks & related technologies):

http://www.embedded.oti.com

Microsoft Corp. (Windows JVM downloads):

http://www.microsoft.com/java/download.htm

Newsgroups

IBM Software Group (MQSeries Everyplace newsgroup):

news://news.software.ibm.com/ibm.software.websphere.mqeveryplace

1 Preparation

The instructions, illustrations and examples in this book are orientated towards a Windows operating system (e.g. Windows 2000 or Windows XP). MQSeries Everyplace (MQe) itself runs on a very broad range of platforms, but Windows has been chosen here simply because more tools exist for this platform and it is consequently the easiest place to begin. Even if you intend to deploy later on a non-Windows platform, it is strongly recommended that you learn the basics first using MQSeries Everyplace on Windows. This book should then provide you with the basic understanding there that you will need to be able to configure MQe queue managers and networks on any other platform.

1.1 Installation

You must first install the following products, and in the sequence listed below:

1. *MQSeries Everyplace for Multiplatforms, v1.27 or later.*
2. *MQSeries SupportPac MA88: MQSeries Classes for Java*
   (only required if you intend to send/receive messages to an MQSeries queue manager).
3. *MQSeries SupportPac ES02: MQSeries Everyplace – MQe_Explorer v1.27 or later.*
4. *MQSeries SupportPac ES03: MQSeries Everyplace – WTLS Mini-Certificate Server v1.27 or later (MQe_MiniCertServer).*

The code is available from the sites listed in the section *Download sites* on page xii. In this book it will always be assumed that the default options and directories have been used during installation, although custom settings should not give rise to difficulties provided that the necessary path changes are made to the examples and instructions.

Before installing MQe_Explorer[1] read the installation instructions in the *MQe_Explorer User Guide* and be certain to:

- Comply with the Microsoft JVM pre-requisite requirement.
- Set the *classpath* variable correctly.

Follow the instructions in the *MQe_Explorer User Gui*de until the end of the section *Configuring a first queue manager*, in the *Getting started* chapter. This will confirm that the base software has correctly been installed.

1.2 Reading

The MQSeries Everyplace for Multiplatforms v1.27 product includes the *MQSeries Everyplace Version 1.2: Introduction* book, which explains the essentials of MQe and presents the basic concepts. It is assumed that readers are familiar with this material.

The MQe_Explorer is used extensively in this SupportPac to illustrate principles and to provide examples. To gain familiarity with this management tool you are strongly recommended to work through at least the first sample script *Concepts and objects* in the *MQe_Explorer User Guide*. The following two scripts, *Basic messaging* and *Advanced messaging* also provide useful information on networking concepts and the use of MQe.

The *User Guide* contains a fourth script, *Gateway configuration and usage*, that explores how MQe and MQ messaging networks can be linked together so that they can exchange

[1] If MQe_Explorer has been previously installed, ensure that the options are reset to the default. Use *Tools→Options*.

messages. This subject is outside the scope of this SupportPac, though information is provided here on the creation of a gateway queue manager.

The following chapters of the *MQSeries Everyplace Version 1.2: Java Programming Guide* describe the essentials of MQe programming:

Overview

Getting started

MQeFields

Queue managers, messages and queues

Rules

Administering messaging resources

If you intend to use MQe certificate-based security, then it may be helpful to browse the *Getting started* chapter of the *MQe_MiniCertServer User Guide*.

1.3 Programming samples

Installation and sample execution

As far as possible, sample code has been supplied with this SupportPac to illustrate the use of the programming examples. The Java source code is supplied in the file *MQeConfigGuide.java*; the compiled class in the file *MQeConfigGuide.class*. This class file should be installed in the directory:

C:\Program Files\MQe\Java\examples\mqe_configuration_guide

The samples must executed from a command prompt[2]; run each example using the string:

jview[3] *examples.mqe_configuration_guide.MQeConfigGuide <example no>*

where <*example no*> is an integer identifying the example (see text)

The above instructions assume that the MQe default installation instructions have been used; the *classpath* will have been set to *C:\Program Files\MQe\Java*.

Detailed instructions for running samples is provided, where appropriate, in highlighted boxes in the text, for example:

Executing the sample – *Running a client queue manager*:

Use the command: *MQeConfigGuide 1*. The sample code is contained in the methods *startClientQMgr()* and *stopClientQMgr()*. By default, the MQe_Explorer-created *myClientQMgr.ini* file is used; by editing the sample code, the method *createClientEnvironment()* can be used instead.

Generally, samples to run are provided only after several code portions have been illustrated. The method(s) of interest in *MQeConfigGuide* are always identified in the highlighted box. To appreciate exactly what is being run, inspect the logic in the *MQeConfigGuide.main()* method.

[2] These examples have not been designed to run from the *Tools→Load* menu of MQe_Explorer.
[3] For a non-Windows platform, substitute *jview* with the appropriate command to start the JVM.

In many cases it may be necessary or desirable to modify the sample code. The code extracts in this book do not purport to show all the relevant code in every case; the extracts that follow include only the essentials of what is required. For example, exception handling is generally not included unless it is essential to the example. Before using the example code in this book, always check the source samples shipped in *MQeConfigGuide.java*.

Coding standards

As far as possible a consistent set of variables names is used across all the examples. Some of the more important variables used are:

```
byte                        returnCode;          //admin return code
byte[]                      adminKey;            //unique identifier
int                         waitTime;            //wait time (mS)
MQeAdminMsg                 adminMsg;            //admin message – normally
                                                 //        a specific subclass
MQeAdminMsg                 adminReply;          //admin reply message
MQeChannelListener          localChannelListener; //client/server listener
MQeChannelManager           localChannelManager; //client/server channel mgr.
MQeFields                   alias;               //alias section
MQeFields                   channelManager;      //channel manager section
MQeFields                   environment;         //environment parameters
MQeFields                   listener;            //listener section
MQeFields                   mqBridge;            //MQ bridge section
MQeFields                   msgFilter;           //message filter
MQeFields                   parms;               //admin parameters
MQeFields                   queueManager;        //queue mgr. section
MQeFields                   registry;            //registry section
MQeMQBridges                localBridges;        //bridges object
MQeQueueManager             qMgr;                //local queue manager
MQeQueueManagerConfigure    qMgrConfig;          //queue manager configurator
String                      targetQMgrName;      //target qMgr name
String                      targetQName;         //target queue name
```

Example 1-1: Common variables

2 The basics

2.1 MQe objects

Understanding MQe configuration requires an appreciation of a number of basic MQe objects. The most important of these are:

- *Queue manager*
  - A queue manager represents an addressable instance of MQe. Each has a unique name that distinguishes it from any other MQe queue manager. More than one queue manager can exist on a single machine. Queue managers own other MQe objects, including *queues*, *connections* and *channels*. The job of the queue manager is to manage these resources and to make them available to application programs.
  - Many different types of queue manager can exist, depending upon the needs of applications. Not only do they differ in their collection of queues, connections, channels and other objects, but also they may differ in their expected behaviour. MQe identifies four distinct roles for queue managers:
    - *Client*

      The essence of a client is that it supplies messages to, or gets messages from, a server.
    - *Peer*

      A lightweight queue manager that can freely communicate in a community of its peers.
    - *Server*

      Servers are expected to provide services to many attached client queue managers. Servers do not normally instigate data transfers.
    - *Gateway*

      A server queue manager that also has the capability to exchange messages with MQSeries base messaging queue managers.
  - These distinctions are very soft in MQe, though behaviour can be rigidly enforced where needed (e.g. in order to respect the rules concerning access through firewalls). On the other hand, it is easy to configure a single queue manager with all the attributes assigned above to clients, peers, servers and gateways[4].
- *Queue*
  - A queue is an addressable entity within a queue manager. Each has a unique name that distinguishes it from any other queue on that same queue manager. Many different types of queue exist but each is generally concerned with messages in some way; one or more of storing, processing or moving messages.

[4] Server and gateway queue managers may be given peer capabilities, quite independently of their server and gateway functionality.

- The properties of a queue depending upon the type of that queue. Queues that store messages (either permanently or temporarily) have properties such as *message store* and *storage adapter*. The message store determines how messages are mapped into the storage medium; the storage adapter provides basic access to that medium (e.g. adapters to the file system or memory). Multiple storage adapters can exist for any particular medium – they may differ, for example, in the level of certainty that they provide about data storage.

- *Connection*
  - A connection provides its local queue manager with all the information it needs to establish communication *channels* with a remote queue manager. The name of a connection is the name of that remote queue manager. Only one connection definition can exist on a local queue manager for each remote queue manager name[5].
  - The information in a connection varies according to the nature of the connection (e.g. a direct connection – that goes straight to a target queue manager; an indirect connection – that goes via another queue manager). For a simple, direct connection, the information will typically include: the IP address of the machine hosting the target queue manager, the port number to be used, the *communications adapter* and the *channel* type.

- *Channel*
  - A channel is an entity created by a queue manager to move messages to another queue manager (and with the cooperation of that remote queue manager). Channels go from source to target queue manager; they do not go to target queues. Channels are not accessible to application programs. They have properties such as security, i.e. encryption and compression. Multiple channels can go from a local queue manager to the same target queue manager at any one time.
  - Different types of channels exist, for example a *client/server* channel allows only the client end to initiate data transfer across the channel (to push or pull data). A *peer-to-peer* channel allows either end to initiate data transfer across the channel.
  - Communications *adapters* are used by channels to provide basic access to the underlying communications infrastructure (e.g. TCP/IP, HTTP, UDP etc). Multiple communications adapters can exist for any particular underlying protocol support – they may differ, for example, in their performance, efficiency or footprint.

- Registry
  - The registry is the primary store for queue manager-related information; one exists for each queue manager. Every queue manager uses the registry to hold details of its properties and child objects. Optionally, other information (including security credentials) can be stored and, in certain configurations, multiple registries can exist for a single queue manager.

[5] This curious form of words allows for alias names of remote queue managers. Alias names allow multiple routes to be defined.

2.2 MQe naming rules

All objects in MQe must be named according to the following rules:

Names can be of unspecified length *(though recognize that, because names may have to be carried in messages, short names are to be preferred for efficiency).*

The following characters are permitted:

Numerics:	0 – 9
Characters:	a – z (lower and upper case)
Special:	_ (underscore), . (period), % (percent)

Restriction: The period character cannot be a leading or trailing character.

Capitalization is significant.

It is recommended that a subset of the MQe naming rules be adopted that restrict the MQe name space to one that is compatible with MQ. This results in the following rules:

Names can be up to 24 characters in length.

The following characters are permitted:

Numerics:	0 – 9
Characters:	a – z (lower and upper case)
Special:	. (period), % (percent)

Restriction: The period character cannot be a leading or trailing character.

Capitalization is significant.

2.3 The approach

The aim of this SupportPac is to provide the basic information to allow MQe queue managers and networks to be configured. This includes:

- *Creating and starting queue managers*
- *Defining connectivity between queue managers*
- *Establishing the routes taken by messages through an MQe network*
- *Exercising control over the protocols used*
- *Determining where messages are staged, if appropriate*
- *Configuring queue-level security*
- *Appreciating the MQe options available and their associated trade-offs*

It is important to appreciate that although MQe is like its sister – the MQ messaging product – at the highest level (i.e. in that it provides messaging services to applications with once-only, assured delivery), it is also different in many respects. These differences can be fundamental; beginning with the fact that MQe moves objects, whilst MQ messaging moves byte arrays. The differences have many consequences such that, at least initially, knowledge of MQ messaging may not be helpful. It is therefore important that assumptions made from analogy with MQ Series base messaging, are not applied indiscriminately to MQe. Although the two products are designed to operate seamlessly together – and indeed that is a great strength of MQe – function, configuration and application can be very different.

The approach taken here is first to describe the ways in which individual queue managers can be created and configured (this incidentally provides another example of differences with base MQ messaging). The Java version of MQe is essentially a set of Java classes[6] that can be deployed by application programmers to construct queue managers. These queue managers can be either:

- *Embedded within an application*
- *Have an independent existence from application(s), thus supporting one or more concurrent applications*

Utilities and samples provided with MQe show how queue managers can be created, or indeed create them. The easiest way of creating queue managers is to use MQe_Explorer, though this might not always be appropriate.

Here we first examine the use of MQe_Explorer and then examine the elements that comprise a queue manager. We look at the code necessary to launch queue managers (of various kinds) and then the additional code needed to create them from scratch.

Only after queue managers have been examined in detail, will we explore the networking aspects. It is impossible to understand MQe networks without a prior understanding of important queue manager details.

In this book the examples are all given in Java. MQe also provides a C interface to the Java, as well as a 100% native C implementation. However using both languages in one book is difficult and unduly repetitive. It is hoped that the Java used here is sufficiently simple that most readers will be able to extrapolate to the C code equivalent; assisted by the fact that the MQe C bindings and interfaces map closely to their Java counterparts.

[6] This statement ignores the existence of the 100% C code base implementation of MQe.

3 Client queue managers

3.1 Using MQe_Explorer

The simplest MQe queue manager is a client queue manager. Such a queue manager is intended to connect to servers, though it has full local function, i.e. it owns queues and other objects. In order to understand the elements of a queue manager we first create a client using the management tool MQe_Explorer and examine its constituents. Later we will achieve the same result through programming.

Run MQe_Explorer and create a new client queue manager as follows. Use the *File→New→Queue Manager* (or the equivalent button on the toolbar) to display the following dialog:

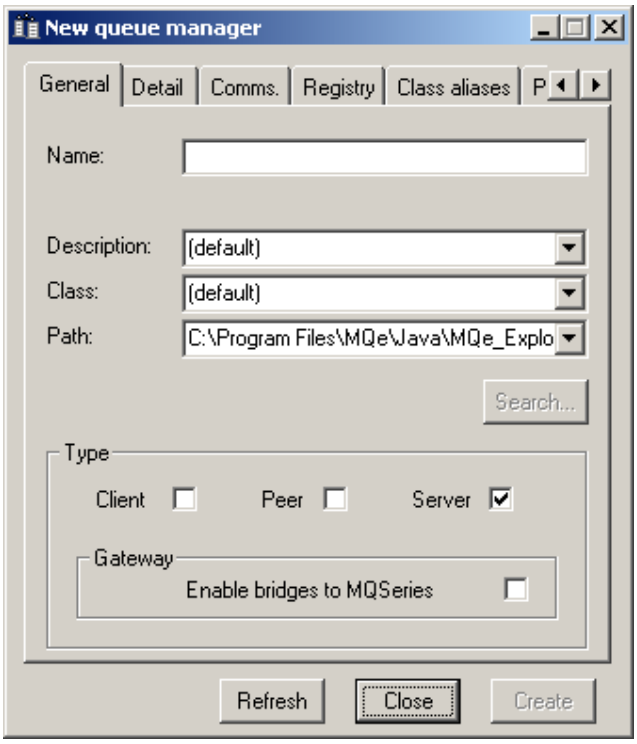

Figure 3-1: New queue manager panel

Edit the following fields on the *General* tab as follows:

***QMgr. name*:**	**myClientQMgr**
***Path*:**	**C:\Program Files\Java\MQe_Explorer**
***Type*:**	**Check the *Client* box**

Only the *Path* parameter is not self evident; the significance of this will be seen later.

Click the *Create* button to create the new queue manager. The following message is displayed:

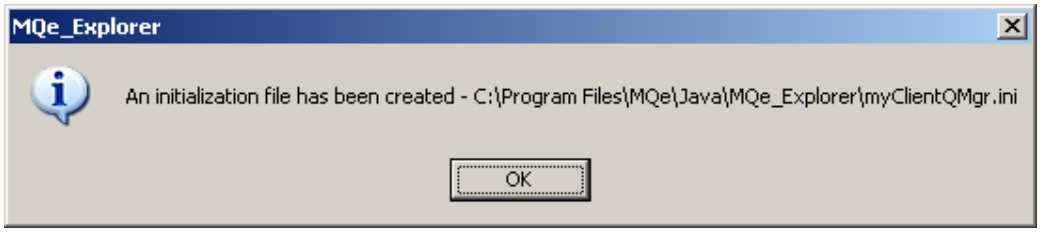

Figure 3-2: Queue manager creation message

The display changes and the new queue manager is seen to be running. Expand all the nodes in the tree and the display should be similar to:

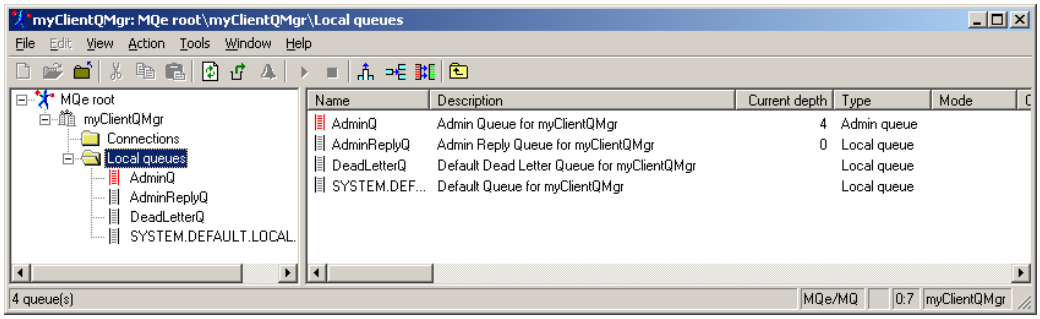

Figure 3-3: MQe_Explorer view of myClientQMgr

Display the property pages for the queue manager. The following window will appear:

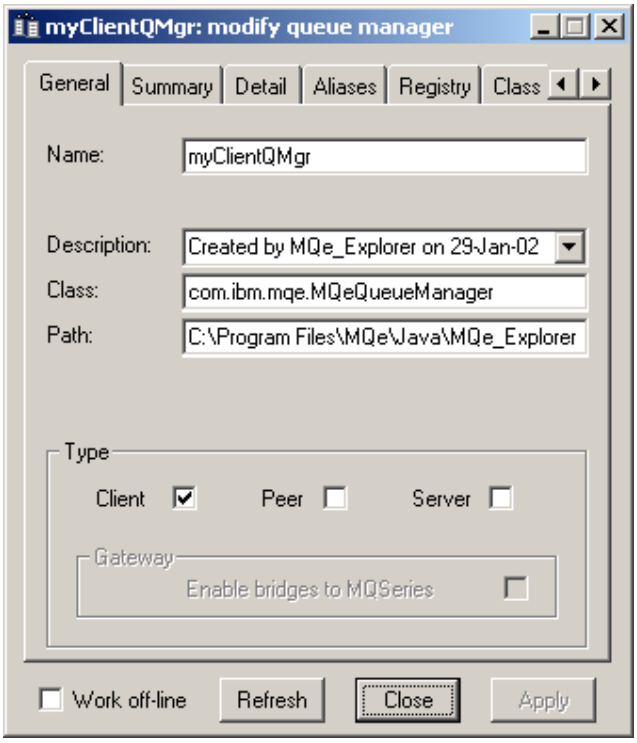

Figure 3-4: Property pages for myClientQMgr

Explore the various tabs – there are a great many properties, essentially set to default values by MQe_Explorer or chosen as a consequence of creating a client queue manager.

Finally close the queue manager and then restart it. In order to restart it you will have to open the initialization file – the name of which was displayed during the queue manager creation process. Opening this file restores the queue manager to exactly the state it was in just before it was closed. If you were to examine all the property pages you would find nothing had changed. Now exit from MQe_Explorer.

Much has happened in those few simple steps. In summary:

1. Starting MQe_Explorer created a new JVM; MQe_Explorer ran as an application in that JVM.
2. Information supplied by the user (*QMgr name*, *Path*, etc.) was supplemented with other values, i.e. relevant defaults, and stored by MQe_Explorer in an initialization file. Note that this file is the creation of MQe_Explorer – it has nothing to do with MQe.
3. MQe_Explorer then read in the file (in exactly the same way as it did later when you restarted the queue manager) and attempted to start the queue manager, passing relevant information to various MQe classes. We shall examine these classes later.
4. MQe was unable to start the queue manager because its registry did not exist, i.e. no registry had been configured for it in the file system (or elsewhere).
5. MQe_Explorer then used another set of MQe classes to create a queue manager registry with characteristics corresponding to those requested.
6. MQe_Explorer attempted again to start the queue manager – and this time it succeeded. Now the MQe queue manager and MQe_Explorer are both running in the same JVM.
7. MQe_Explorer then queried the queue manager to gets information on all of its properties. It was then able to display *Figure 3-3: MQe_Explorer view of myClientQMgr* and *Figure 3-4: Property pages for myClientQMgr*.
8. Closing the queue manager involved calls to MQe to shut it down. Afterwards only MQe_Explorer was running in the JVM.
9. Restarting the queue manager involved re-reading the initialization file and opening the queue manager, based on the information in that file.
10. Exiting from MQe_Explorer closed the queue manager, terminated the MQe_Explorer application and destroyed the JVM. The initialization file and the registry survived.

In order to write code to do the equivalent function, first the nature of the various properties of a queue manager must be understood. The next stage is to be able to start an existing queue manager, e.g. the *myClientQMgr* that MQe_Explorer has just created. The final step is to create a new queue manager, including a configured registry, from scratch.

The client queue manager is the simplest MQe queue manager that can be created; others have additional capabilities and consequently require additional features.

3.2 Property pages

The property pages displayed by MQe_Explorer for a client queue manager had the following tabs. The table below describes these in more detail and, crucially, shows where the associated data is stored.

Tab name	Description	Type	Location
General	Information that identifies the queue manager, class, type and registry location	*Name*: qMgr property *Class*: environmental info, registry, *Path*: environmental info *Type*: application concept	Application-defined Registry
Summary	An analysis of other properties	(Not applicable)	(Not applicable)
Detail	A collection of detailed properties	QMgr. properties	Registry
Aliases	Alternative names that resolve to this queue manager	Local connection property (see later)	Registry
Registry	Information describing how the registry is realized and protected	Registry properties	Registry
Class aliases	Alternative names that can be mapped to Java class names	Environmental info.	Application-defined
Pre-loads	A set of classes to be pre-loaded when the queue manager is loaded	Environmental info.	Application-defined
Permissions	Information describing which operations are to be permitted	Environmental info.	Application-defined

Figure 3-5: Client queue manager properties

From the table above it can be seen that information is stored in two places – by MQe in the *registry* and by the application in a place of its choosing. The application data is used to establish an *environment*.

The *registry* contains what MQe considers to be the queue manager configuration, including for example:

- Attributes of the queue manager object itself:
  - Name, class, description, channel timeout, …
- Objects owned by that queue manager:
  - Queues
    - Name, class, description, max depth, …
  - Connections
    - Name, description, channel class, adapter, address, …
  - Various bridge-related objects
    - Name, description, …

Details of these properties are discussed in the chapter *Queue manager properties* on page 38.

The *registry* does not contain the queues themselves, nor the messages that these queues might contain. We will see this later. The registry does however contain information about where those queues are and how they are to be accessed.

The *environment* is something that is created by an application before a queue manager is started (or created). For almost all purposes it can be considered to represent additional properties of the queue manager that happen not to be stored in the registry. The fact that they are not in the registry means that the application (or an application) must remember them between invocations of the queue manager. In principle, it also means that they can be changed independently of the information stored in the registry; however, with very few exceptions, this is bad practice. MQe_Explorer chooses to treat this environmental data just as though it was an extension of the registry – and uses an initialization file to store it[7]. The environmental data stored by MQe_Explorer for a client queue manager includes:

- Registry information:
    - Class, path, adapter class, registry type
- Class alias mappings
- Queue manager information:
    - Class, name
- Additional information
    - Queue manager type, registry type

The use of this environmental data is as follows:

> The registry content is sufficient to allow either an existing queue manager to be started (i.e. its registry located and accessed) or a new queue manager to have a registry created.
>
> The class alias mappings establish a default sets of names that can be used to refer to Java classes – and Java class names are used extensively throughout MQe to customize objects and properties. The long nature of these names means that shorthand references are useful and are widely used. Consequently, the default class alias mappings should always be established early when interfacing to MQe.
>
> Starting or creating a queue manager needs a class to instantiate and a name to be assigned.
>
> The additional information here is specific to MQe_Explorer, although other applications may need to do something similar. MQe_Explorer presents the concept of a queue manager type (client, peer, server or gateway) to the user and configures accordingly. It keeps track of the type (and associated properties); consequently it must be stored between invocations.

The registry information for *myClientQMgr* can be found at *C:\Program Files\Java\MQe_Explorer\myClientQMgr*. Since default values were used during creation, the registry type is *file registry* – though this cannot be deduced from an inspection of the file system. Registries and registry types are discussed further in the next chapter.

The environmental information is in the initialization file *C:\Program Files\Java\MQe_Explorer\myClientQMgr.ini.* MQe_Explorer used the path information supplied when the user requested that a new queue manager be created to (by default) generate these locations.

---

[7] In future releases of MQe it is likely that some of this environmental information will be moved inside the MQe registry.

Using the Windows File Explorer, the structure of the MQe registry for *myClientQMgr* can be seen as:

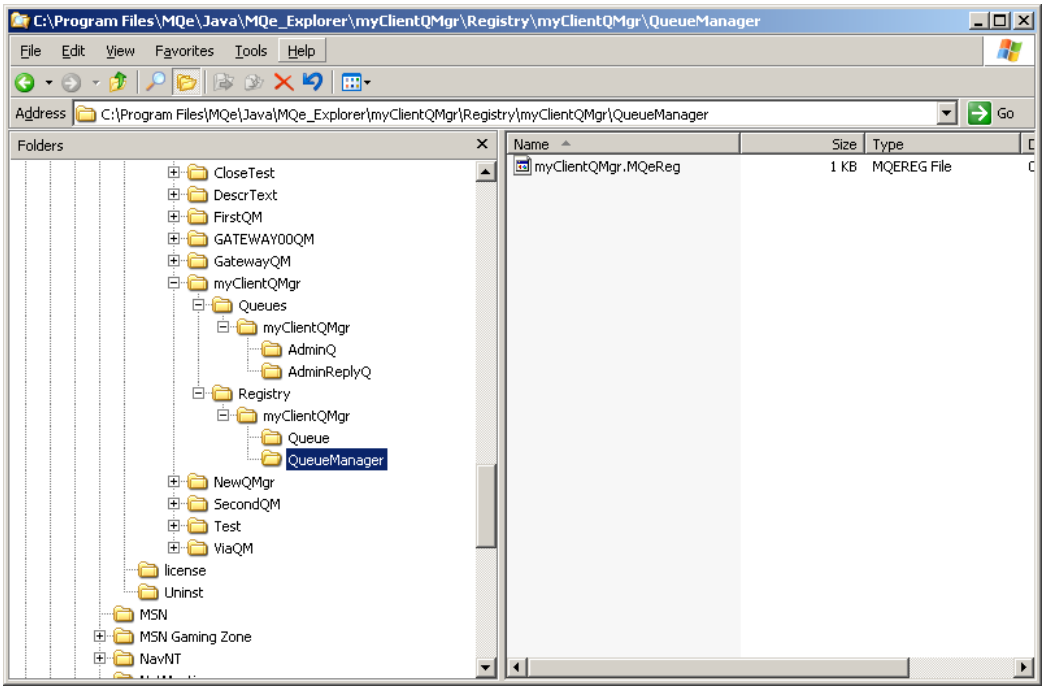

Figure 3-6: The myClientQMgr registry structure

Also shown above is the skeleton structure for the queue storage. Directories are not seen for the *DeadLetterQ* and the *SYSTEM.DEFAULT.LOCAL.QUEUE* because they have yet to receive a message. MQe will create their directories when the queues become active.

The environmental data, saved by MQe_Explorer in the initialization file, can be inspected with the Windows Notepad editor:

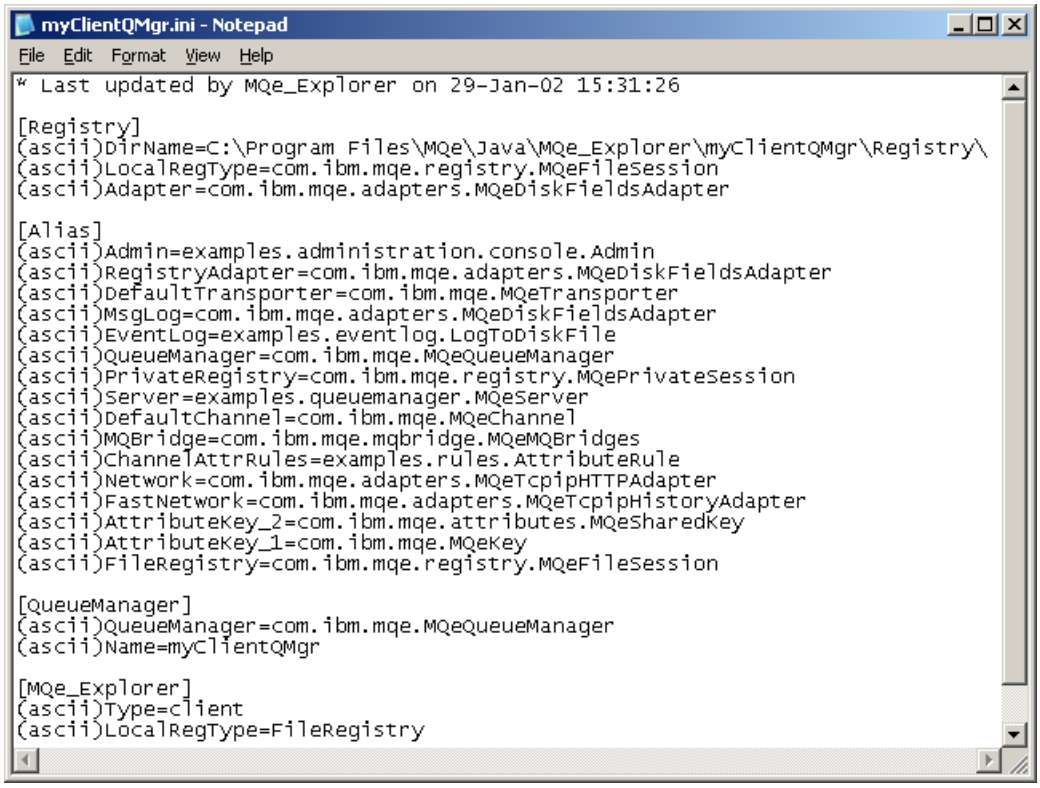

Figure 3-7: The myClientQMgr initialization data

3.3 Running a client queue manager

Setting up a client environment

In this section we will assume, like MQe_Explorer, that the environmental data has been arbitrarily stored in an initialization file. Many of the MQe utilities supplied with the product make the same assumption.

Reading, writing and processing such files is made easy by helper methods in the *examples.queuemanager.MQeQueueManagerUtils* class.

The code below sets up the client queue manager environment:

```
//read in the initialization file
MQeFields environment = MQeQueueManagerUtils.loadConfigFile(filename);

//process class aliases – [Alias] stanza
MQeQueueManagerUtils.processAlias(environment);

//process pre-loads – [PreLoad] stanza
MQeQueueManagerUtils.processPreLoad(environment);

//process permissions – [Permission] stanza
MQeQueueManagerUtils.processPermission(environment);
```

Example 3-1: Setting up a client queue manager environment

The *loadConfigFile*() method code above loads the contents of the initialization file into an MQeFields object. Each stanza name becomes the name of a nested MQeFields object; within each nested object each item name becomes a field name, its bracketed type prefix sets the data type, and its assignment value is the item value. The simple default client queue manager did not have pre-loads and permissions set – hence no corresponding stanzas are present in *Figure 3-7: The myClientQMgr initialization data*. Had other options been exercised before the queue manager had been created, or had changes been made later when the property pages were displayed, then additional stanzas would have been created by MQe_Explorer.

The various *processXxx*() methods take the entire fields object, select the appropriate nested fields object for processing (i.e. the relevant original stanza) and set up that aspect of the environment. Since the source of all the example classes is provided, you can examine the details of each *processXxx*() method if you wish to use only the relevant parts of the *MQeQueueManagerUtils* class, or to directly use the underlying MQe classes and methods.

Starting a client queue manager

Once the environment is established, the queue manager can be started. Again, using a method in the *examples.queuemanager.MQeQueueManagerUtils* class:

```
//start the queue manager
MQeQueueManager qMgr =
  MQeQueueManagerUtils.processQueueManager(environment, new Hashtable());
```

Example 3-2: Starting a client queue manager

The *processQueueManager*() method uses both the *[Registry]* and *[QueueManager]* stanzas. Again, examining the source of the method will show the details. The second parameter is set to a new Hashtable object if the queue manager is to be a client, as in this case (i.e. if it does not have a client/server channel listener and channel manager)[8].

[8] It can be seen that the method actually instantiates a queue manager of the class that corresponds to the alias "QueueManager". MQe_Explorer itself looks for the item named "QueueManager" in the "QueueManager" stanza; if present, it instantiates a queue manager of the class corresponding to that item's value; otherwise it behaves as the *processQueueManager()* method described above.

We now have a reference to the queue manager. We can use the methods in the *com.ibm.mqe.MQeQueueManager* class to get and put messages, browse queues, add a message listener to queues, and so on. Similarly, we could complete or change the configuration of the queue manager by changing queue manager properties, adding queues and connections and so on. These configuration changes will be described later.

Stopping a client queue manager

A client queue manager can be stopped by using the *close()* method in the *com.ibm.mqe.MQeQueueManager* class:

```
//get addressability to the local queue manager
MQeQueueManager qMgr = MQeQueueManager.getReference(null);

//close the queue manager
qMgr.close();
```

Example 3-3: Stopping a client queue manager

The *MQeQueueManager.getReference(null)* call gets a reference to the currently active queue manager. If a reference already exists, for example where the application has previously started the queue manager, this statement is obviously not needed.

Executing the sample – *Running a client queue manager*:

Use the command: *MQeConfigGuide 1*. The sample code is contained in the methods *startClientQMgr()* and *stopClientQMgr()*. By default, the MQe_Explorer-created *myClientQMgr.ini* file is used; by editing the sample code, the method *createClientEnvironment()* can be used instead.

3.4 Creating a client queue manager

The starting queue manager code above only works correctly if the queue manager exists (i.e. if the associated registry has already been created); if not, an exception will be thrown indicating that the queue manager is not configured. The code below configures a queue manager, i.e. it creates the queue directory, the registry and sets key parameters, and can be invoked in the exception handler.

```
//initialize strings
String queueDirectory =
        "C:\\Program Files\\MQe\\Java\\MQe_Explorer\\myClientQMgr\\Queues";

//make the queue directory
File qFile = new File(queueDirectory);
qFile.mkdirs();

//set the details for default queue storage
String msgStore = "com.ibm.mqe.messagestore.MQeMessageStore";
String queueAdapter = "com.ibm.mqe.adapters.MQeDiskFieldsAdapter";
String queueStorage = msgStore + ":"  + queueAdapter + ":" + queueDirectory;

//create a queue manager configurator
MQeQueueManagerConfigure qMgrConfig =
        new MQeQueueManagerConfigure(environment, queueStorage);

//set queue manager properties
qMgrConfig.setDescription("My queue manager");
qMgrConfig.setChnlAttributeRuleName("examples.rules.AttributeRule");
qMgrConfig.setQMgrRuleName("com.ibm.mqe.MQeQueueManagerRule");

//create the queue manager
qMgrConfig.defineQueueManager();

//create default queues
qMgrConfig.defineDefaultAdminQueue();                    //AdminQ
qMgrConfig.defineDefaultDeadLetterQueue();               //DeadLetterQ
qMgrConfig.defineDefaultSystemQueue();                   //SYSTEM.DEFAULT...
qMgrConfig.defineDefaultAdminReplyQueue();               //AdminReplyQ

//close the configurator
qMgrConfig.close();
```

Example 3-4: Creating a client queue manager

The code above is fairly clear; there is only a need to create a queue directory if queues are required on the client queue manager; otherwise set the queue storage string to *null*. The queue manager configurator process uses both the *[Registry]* and the *[QueueManager]* stanzas. In this case, a registry of the *file registry* type will be created.

The queue manager properties shown can be set before the queue manager is defined. After definition, but before being started, a limited number of queues can be created directly using the methods shown. It is very useful to create at least the *AdminQ* and the *AdminReplyQ* in this way; the existence of these two queues later allows admin messages to create and/or modify all the other MQe objects that may be needed on a queue manager. Once the configurator is closed, the queue manager can be started, as shown earlier.

Executing the sample – *Creating a queue manager*:

First delete the *myClientQMgr* created by MQe_Explorer, i.e. use the Windows File Explorer and delete the directory *C:|Program Files|MQe|Java|MQe_Explorer|myClientQMgr|* and all of its sub-directories. Do not delete the *myClientQMgr.ini* file in the parent directory.

Then use the command: *MQeConfigGuide 2*. The sample code is contained in the method *createClientQMgr().* By default, the MQe_Explorer-created *myClientQMgr.ini* file is used; by editing the sample code, the method *createClientEnvironment()* can be used instead.

If subsequently the queue manager needs further configuration, this should be done after the queue manager has been started, through admin messages. Either the messages can be generated programmatically (see *Configuration using admin messages* on page 40, or MQe_Explorer can be used). In either case, the configuration changes can be effected either locally or remotely.

3.5 Queue manager considerations

MQe requires that each queue manager in an MQe network is uniquely named; enforcing this is the responsibility of the user – since MQe itself has no way of knowing of the existence of all other queue managers. Strict adherence to this condition is essential. Naming and routing flexibility is available where necessary, through the provision of alias support (see later sections).

MQe itself poses no restrictions on the number of queue managers on any physical machine; indeed multiple queue managers per machine is a common way of exploiting MQe or of prototyping configurations. There is however a restriction of one instance of an MQe queue manager per JVM.

Queue managers use registry information; the registry gives the queue manager its name and properties. Consequently a single queue manager owns any particular registry and it is not permitted to instantiate more than one instance of that queue manager at any one time. Although MQe does not check for this occurrence, serious errors will occur if a registry is shared in this way. Such usage should be unnecessary since any single queue manager can support multiple concurrent applications, each using multiple concurrent threads to communicate with the queue manager if desired, and queue managers can have remote access to each other's queues.

4 Registry

4.1 Overview

The registry is the primary store for queue manager-related information; one (or more) exists for each queue manager and holds details of most of its properties and child objects. Optionally, user data (including security information) can be stored in the registry. Two kinds of registry are supported for these purposes:

(a) *File registry* – an unprotected registry held in the file system.[9]

(b) *Private registry* – a protected registry held in the file system, with PIN protected access. It provides additional services to those available in the file registry, for example credential storage (such that the entity's private key may be used for digital signature and RSA decryption, without the private credentials leaving the registry). A single private registry can be configured – associated with the queue manager itself, or additional further private registries can also be configured, each associated with an individual queue.

Additionally, a queue manager may be configured with facilities for the sharing of public credentials (mini-certificates), with secure storage, through:

(c) *Public registry* – a publicly accessible repository for mini-certificates, relevant only to the use of the message-level security features of MQe. This public registry is an active service; if accessed to provide a mini-certificate that is does not hold, and if configured with valid certificate server details, it automatically attempts to get the requested mini-certificate from the public registry of that server.

A typical usage scenario for the public registry is to build a store of the mini-certificates as they are used; alternatively the registry can be pre-loaded with the mini-certificates required.

The required registry type for a queue manager can be determined from the security requirements that must be satisfied:

Registry	Security requirements supported
File registry	No security, plus: Use of the compressors in message-level or queue-level security: *com.ibm.mqe.attributes.GZIPCompressor* *com.ibm.mqe.attributes.LZWCompressor* *com.ibm.mqe.attributes.RleCompressor* Use of the cryptors in message-level or queue-level security: *com.ibm.mqe.attributes.MQeXorCryptor* *examples.attributes.TableCryptor* Use of the authenticators in message-level or queue-level security: *examples.attributes.UseridAuthenticator* *examples.attributes.NTAuthenticator* *examples.attributes.UnixAuthenticator*

[9] The registry adapter determines the registry location; typically it is placed in the file system.

Registry	Security requirements supported
Private registry	Supports all requirements satisfied by *file registry*, plus: Use of the cryptors in message-level or queue-level security: *com.ibm.mqe.attributes.MQeDESCryptor* *com.ibm.mqe.attributes.MQe3DESCryptor* *com.ibm.mqe.attributes.MQeMARSCryptor* *com.ibm.mqe.attributes.MQeRC4Cryptor* *com.ibm.mqe.attributes.MQeRC6Cryptor* Use of the authenticator in queue-level security: *com.ibm.mqe.attributes.MQeWTLSCertAuthenticator* Local security using: *com.ibm.mqe.attributes.MQeLocalSecure* Message-level security using: *com.ibm.mqe.attributes.MQeMAttribute*
Private registry + Public registry	Supports all requirements satisfied by *file registry* alone, plus: Use of the authenticator in message-level security: *com.ibm.mqe.attributes.MQeWTLSCertAuthenticator* Message-level security using: *com.ibm.mqe.attributes.MQeMTrustAttribute*

Figure 4-1: Registry selection based on security requirements

The various security classes provided with MQe (and mentioned above) are described in *Security* chapter on page 109.

When configuring any registry there are three significant properties to set:

- *Registry type* (together with the associated class that implements that type)
- *Registry storage adapter class* (that maps the registry into a storage medium)
- *Registry location* (that determines where the registry will be stored)

Once the registry has been created, these properties cannot be changed; in particular a registry cannot be upgraded, say from *file registry* to *private registry*. Queue registries can however be added as required (see below).

The characteristics of the various MQe storage adapters are described in the chapter *Storage adapters* on page 116. The default class is *com.ibm.mqe.adapters.MQeDiskFieldsAdapter*, a class that stores the registry in the file system.

The configuration and use of a *public registry* is beyond the scope of this book; further details are provided in the *MQe Programming Guide*.

File registry

The MQe *file registry* is provided by the *com.ibm.mqe.registry.MQeFileSession* class. It provides full registry services, but with these characteristics:

- No PIN protection for registry access
- No ability to store credentials (i.e. certificates, keys etc)
- No ability to share credentials

There are very significant consequences arising from its inability to store and share credentials – these form much of the substance of the table in *Figure 4-1*. It is generally suitable for queue managers where security is not an important consideration.

Private registry – queue manager

The MQe *private registry* is provided by the *com.ibm.mqe.registry.MQePrivateSession* class. It does not (by itself) have the ability to share credentials, hence there are some restrictions on the message-level security functions; however it does support fully support queue-level security. It is PIN protected – in fact, there are three PINs/passwords that are important, though not all are relevant to all the uses of *private registry*:

- Registry PIN (needed to open the registry)
- Key ring password (used to protect the registry's private key)
- Certificate server request PIN (used to obtain mini-certificates)

None of the above are stored by MQe.

The *registry PIN* must be supplied each time a private registry is to be opened. The *key ring password* is normally supplied at the same time, though is relevant only when the registry holds credentials and MQe requires access to the private key. The *certificate server request PIN* is used only once for the queue manager private registry (but see the following section on queue private registries); it is used to obtain a credential (mini-certificate) from the certificate server. Once used, it is invalid for any further certificate requests for that credential.

There are two scenarios for the use of private registry, each assuming that queue-level security is important:

- Where mini-certificates are not required for authentication purposes, i.e. the *com.ibm.mqe.attributes.MQeWTLSCertAuthenticator* class is never to be used:

  In this case only the *registry PIN* is relevant. The various cryptors in the table above can be used. No certificate server is required.

- Where mini-certificates are required for authentication purposes, i.e. the *com.ibm.mqe.attributes.MQeWTLSCertAuthenticator* class is to be used:

  The queue manager must acquire two certificates; its own and that of the certificate server that issued its certificate. These certificates are obtained through an auto-registration process, which can take place whilst the queue manager is first being configured, or at any subsequent restart of the queue manager. The trigger for auto-registration is supplying MQe with the key ring password and the certificate server request PIN (along with addressing details of the server) at the same time as the other queue manager and registry section data. If MQe is passed these parameters and the private registry does not have the necessary credentials, it immediately attempts to acquire them by making requests to the certificate server.

Note that *private registry*, with or without credentials, does not in itself protect the queues in any way. Queue protection results from the setting of queue properties – however the properties that may be set are constrained by the nature and content of the registry.

Private registry – individual queues

A private registry for an individual queue can only be established if the queue manager itself has a private registry, and if that registry contains credentials (i.e. the certificates defined above). The significance of a private queue registry is that a queue has its own credentials and can use these instead of the queue manager's, i.e. it too can have its own certificate – and also that of the server that issued it. As previously, these certificates are only relevant to the use of the *com.ibm.mqe.attributes.MQeWTLSCertAuthenticator* class.

Queue private registries are created automatically as needed, i.e. when a queue has the *com.ibm.mqe.attributes.MQeWTLSCertAuthenticator* for the *authenticator* property set, as well as the 'Queue' value for the *target registry* property. In this case, if the queue does not have its own credentials, they are automatically acquired. A request is made to the certificate server using whatever *certificate server request PIN* was supplied to MQe when the queue manager was started – this is therefore very likely to be the same *certificate server request PIN* that was used to acquire credentials for the queue manager[10]. A queue private registry also has a *registry PIN* and a *key ring password*; MQe generates both these from a digest of the corresponding values for the queue manager registry[11]. Applications can do likewise should they ever need to work directly with the queue registry.

---

[10] This has great significance in certificate server administration. The certificate server request PIN is normally assigned by an administrator; in many cases it will be desirable that the PIN assigned for the queue credentials request be the same as that assigned for the queue manager credentials request. If not, it will be necessary to start MQe to configure the queue manager with one PIN; then restart MQe with a different PIN to configure the queue.

[11] This technique is used to ensure that the queue manager registry PIN does not persist in memory.

4.2 Creating a client queue manager with private registry

Using MQe_Explorer, a client queue manager with a private registry can be created by setting the new queue manager creation dialog tabs as follows:

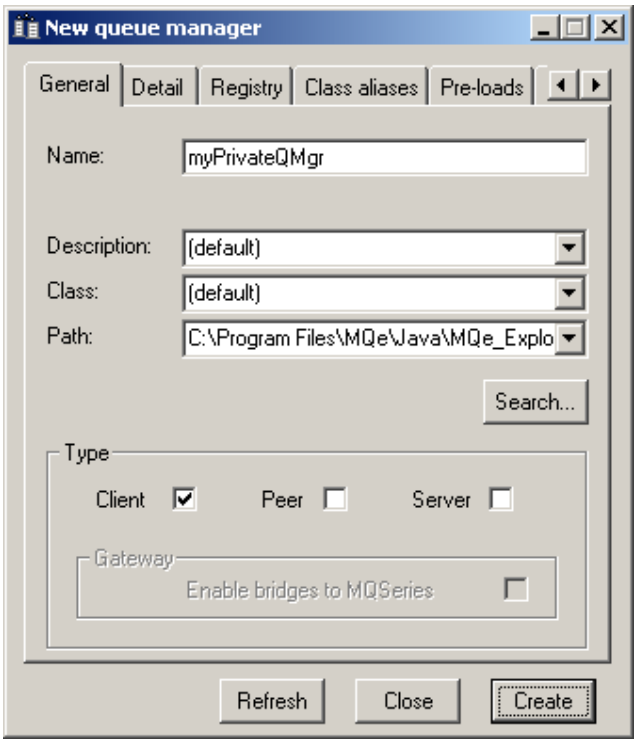

Figure 4-2: Create client queue manager (private registry) – General tab

The registry type is set to *Private registry*:

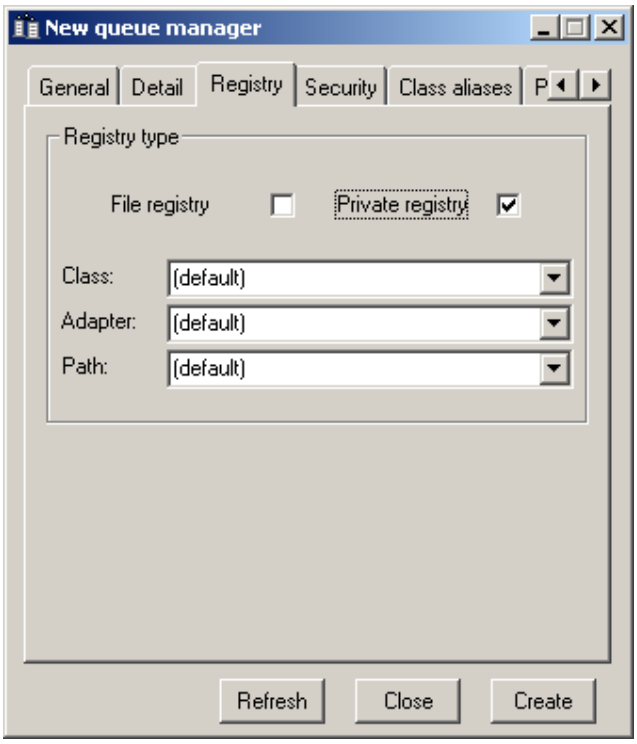

Figure 4-3: Create client queue manager (private registry) – Registry tab

The use of the *Prompt for passwords* option prevents passwords appearing in the initialization data:

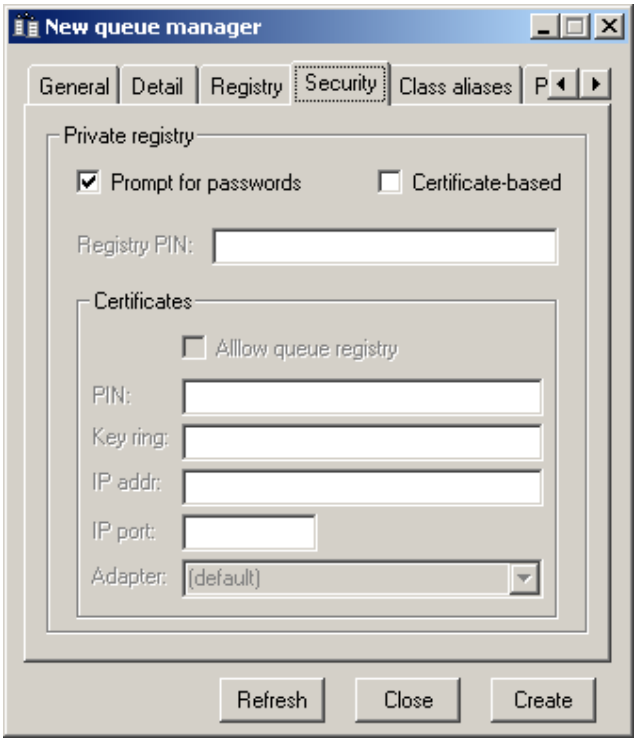

Figure 4-4: Create client queue manager (private registry) – Security tab

After the *Create* button is clicked, MQe_Explorer will prompt for a PIN that will be used to protect the private registry. This PIN is not stored and therefore must be remembered. Without the PIN the registry is inaccessible; the queue manager cannot be started, and messages cannot be retrieved.

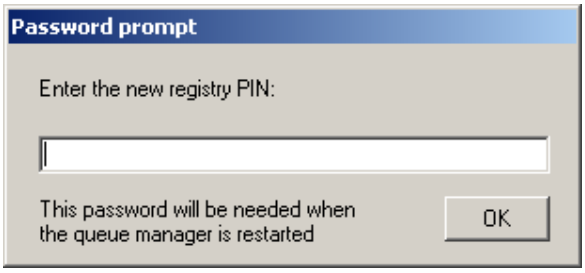

Figure 4-5: Create client queue manager (private registry) – Password prompt

The password is subject to the following checks by MQe_Explorer:

1. At least 6 characters in length.
2. Contain at least 4 different characters.
3. Not match a number of pre-defined (i.e. obvious) values.

Clicking the *Create* button creates the new queue manager.

The initialization data associated with the *myPrivateClientQMgr* is:

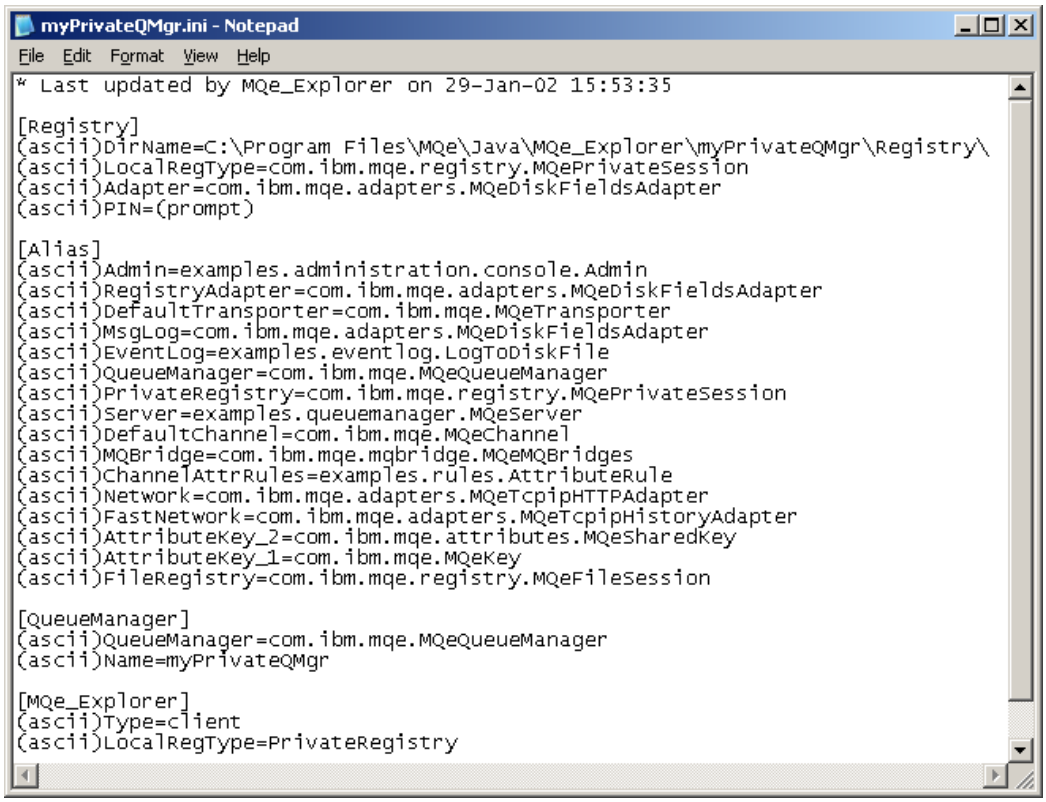

Figure 4-6: The myPrivateClientQMgr initialization data

Compare this with the equivalent client queue manager having a file registry, shown in *Figure 3-7: The myClientQMgr initialization data* on page 14. Excluding trivial name differences:

1. In the *[Registry]* section the registry type class has changed from *com.ibm.mqe.registry.MQeFileSession* to *com.ibm.mqe.registry.MQePrivateSession*.
2. A new entry for PIN has been added to the *[Registry]* section.

The code required to create a client queue manager, is identical to that given earlier for a queue manager with a file registry. The key call is the creation of *MQeQueueManagerConfigure* object, as shown in *Example 3-4: Creating a client queue manager* on page 17. This instantiation processes the content of the *[Registry]* section; it expects that the *PIN* entry contains the PIN value. It is passed the *environment* MQeFields variable containing the PIN, but removes it after using the value. MQe_Explorer implements the user prompt by pre-parsing the environment variable, requesting passwords from the user and substituting the respective value whenever it finds a "(prompt)" value.

4.3 Auto-registration with the mini-certificate server

If the *com.ibm.mqe.attributes.MQeWTLSCertAuthenticator* is to be used (i.e. mini-certificates) then the private queue manager registry must have its own credentials. This can most easily be achieved through *auto-registration*. The steps involved are:

1. Enable the mini-certificate server to issue a certificate to the authenticatable entity.
2. Create a queue manager with a secure registry, using mini-certificate based security.

The description that follows assumes that the MQe_MiniCertServer (from the MQSeries Everyplace ES03 SupportPac) is used to serve the certificates[12]. A more detailed version of the certificate-server aspects of this example is given in the MQe_MiniCertServer User Guide.

Enabling MQe_MiniCertServer to issue a certificate

To start the certificate server double click the icon on the desktop or run the *MQe_MiniCertServer.exe* from a command line. The following window appears:

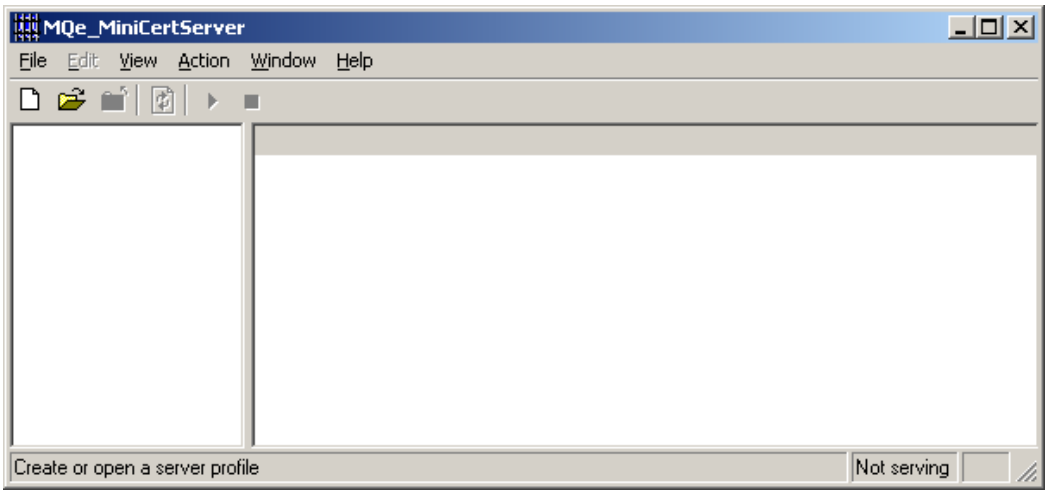

Figure 4-7: The initial MQe_MiniCertServer window

Click *View→Report* to activate and display the report window; this will display all recent server and significant operator activity.

[12] The MQSeries Everyplace SupportPac ES03 includes both a platform neutral, command-driven mini-certificate server (CommandConsole) and a Windows-only certificate server (MQe_MiniCertServer). Both of these are fully compatible and can (sequentially) share the same certificate database. The MQe_MiniCertServer is used here because it supports a graphical user interface.

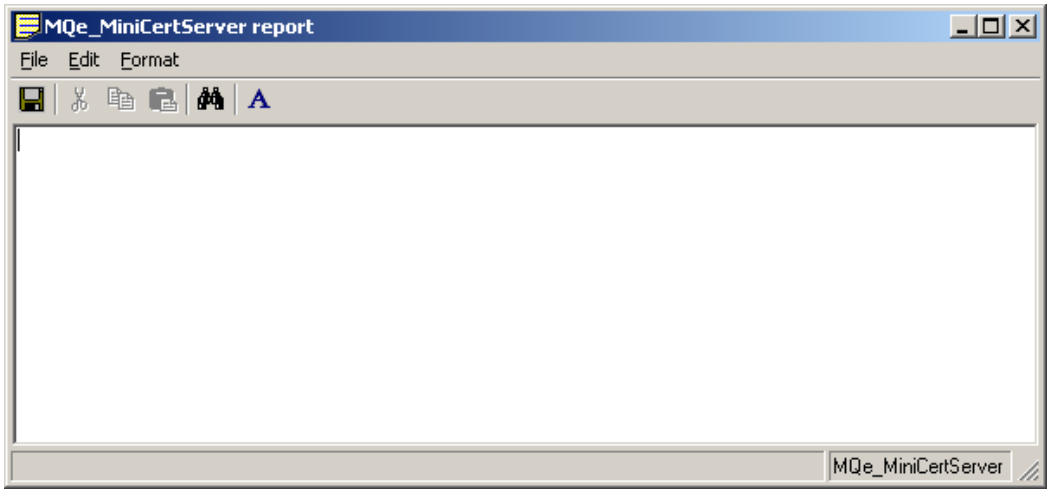

Figure 4-8: MQe_MiniCertServer report window

Assuming that this is the first time that MCS has been run, a new profile must be created; on future occasions an existing profile could be opened instead. To create a new profile, click the new profile icon on the toolbar. The form below will be displayed:

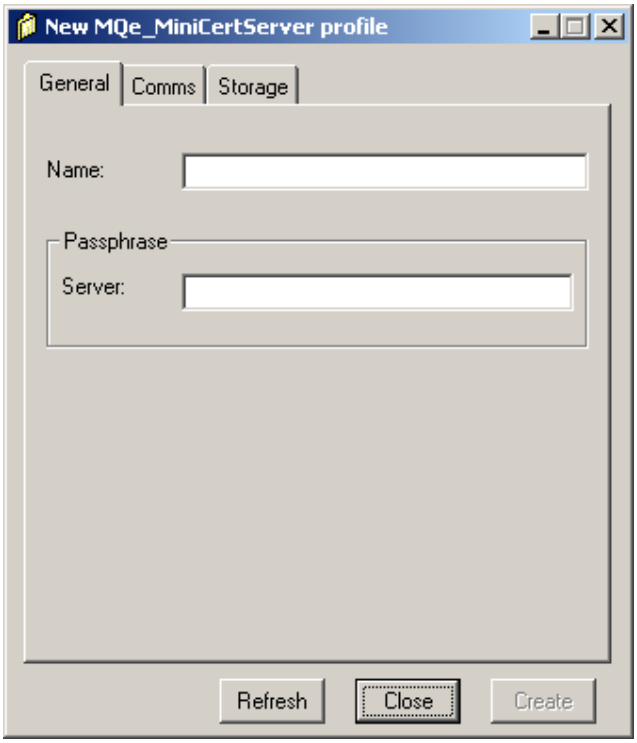

Figure 4-9: Create new profile – General tab

Enter the following data on the *General* tab:

1. **Name**: a unique name for this profile (e.g. "ConfigTests").
2. **Passphrase**: a password string to protect the registry (e.g. "abcdefghijk").

Enter the following data on the *Comms* tab:

1. **Port**: the IP port to receive incoming connection requests (e.g. "8085" is the expected default value).
2. **Adapter**: the adapter to be used (by default the *com.ibm.mqe.adapters.MQeTcpipHttpAdapter* is used – supporting incoming requests over the HTTP protocol).
3. **Timeout**: the channel time out value in seconds (by default "300" is proposed).
4. **Max. channels**: the maximum number of simultaneous channels (by default no maximum limit is set).

Enter the following data on the *Storage* tab:

1. **Adapter**: the storage adapter used to hold the server data (by default the adapter *com.ibm.mqe.MQeDiskFieldsAdapter* is used).
2. **Path**: the location of the directories and files used to hold the server data.
3. **Log file**: the fully qualified path of a file to be used to hold the audit log data (the file and associated directories will be created if they do not exist).

Click the *Create* button to save and load the profile. The passphrase will be requested again; after re-entering, the main window will then change in appearance:

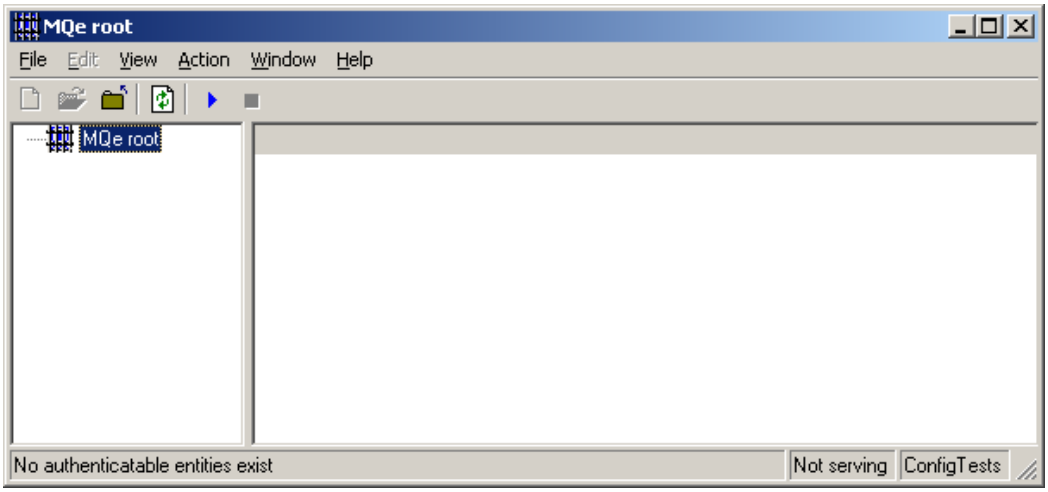

Figure 4-10: The running MQe_MiniCertServer window

The left hand pane holds the tree view of the authenticatable entities; since none exist only the root representing the MCS server (and associated profile) is shown. The right hand pane shows the children of the selected tree node in the left hand pane; again, for the same reason, the pane is empty.

To create a new queue manager authenticable entity, right click on the *MQe root* node in the tree pane and select *New Entity* (or select the *MQe root* node and use the *File→New→Entity* menu item). The new entity dialog appears.

Enter the following data on the *General* tab:

1. QMgr: the name of the queue manager (in this example "myCertClientQMgr")
2. PIN: the mini-certificate request pin to be used by the queue manager when retrieving its certificate (e.g. "1122334455")

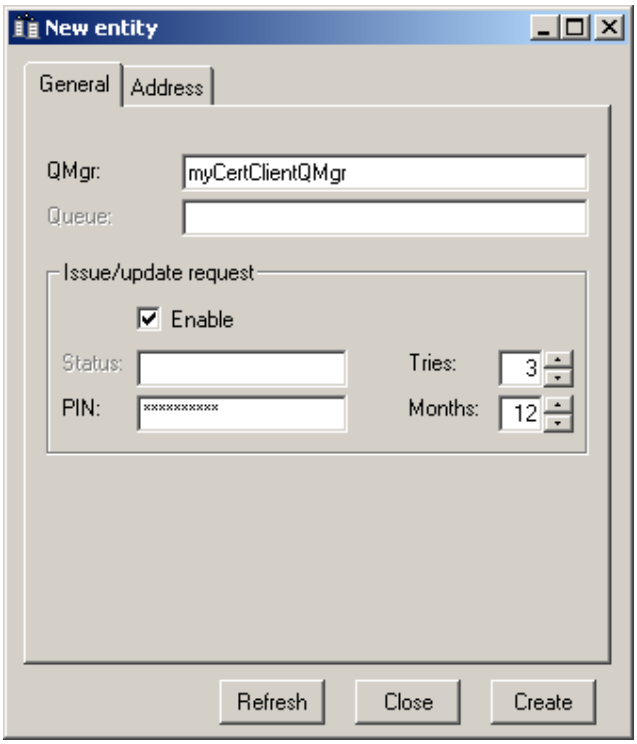

Figure 4-11: The new entity dialog – General tab, with data entered

The *Tries* and *Months* parameters have been left to their default values. *Tries* controls the number of attempts that are allowed for the authenticatable entity to retrieve its certificate; each time an attempt is made with the wrong PIN, one less attempt remains. *Months* controls the duration of the certificate; by default certificates are issued for 12 months, but the period can be set in the range 1 – 15 months.

Other identification data can be entered on the *Address* tab; each line can contain arbitrary text. MCS stores and retrieves this information, but does not use it.

Click on the *Create* button, then the *Close* button to remove the dialog. The tree and list view panes of the MQe_MiniCertServer will be updated; likewise the report window will have logged the activity.

Start the MCS server by clicking the ▶ icon on the toolbar (or by using the *Action→Start* menu item). The text in red changes to black and the status bar indicates that the server is now listening for incoming certificate requests. In this running state all administrative commands are still fully enabled. Leave the MQe_MiniCertServer running so that it can serve certificates on demand to requests received over the network.

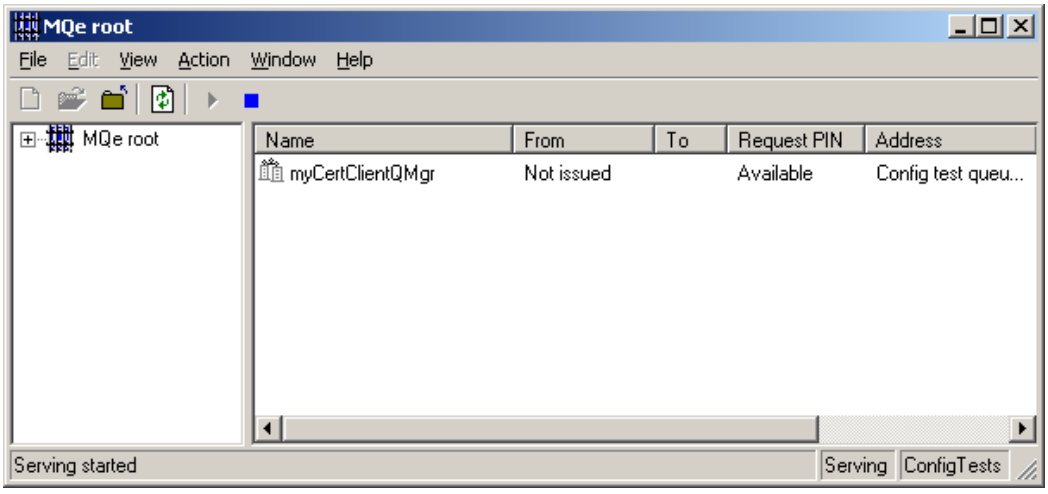

Figure 4-12: Main window before auto-registration

Creating a client queue manager using mini-certificates

Use MQe_Explorer to create a client queue manager with a private registry containing its own mini-certificate credentials. On the *General* tab complete the input fields as shown:

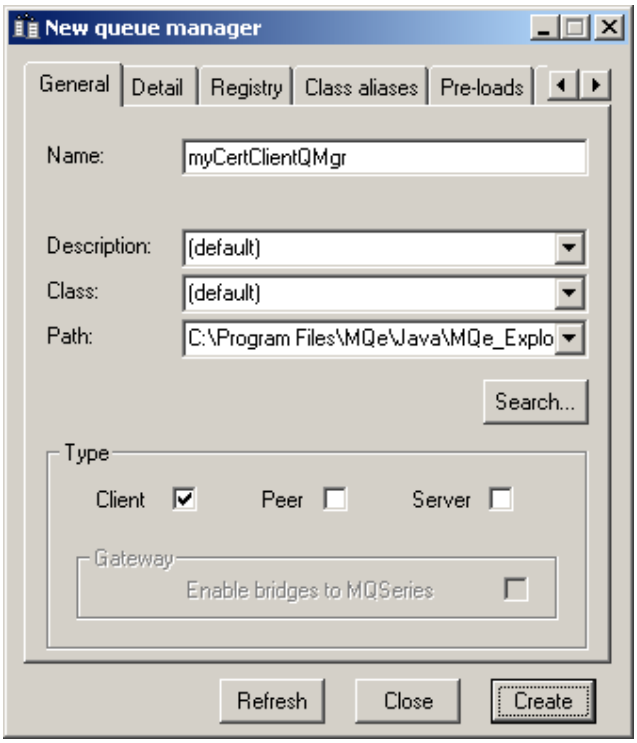

Figure 4-13: Create client queue manager (with credentials) – General tab

This *General* tab is used to enter the queue manager name and to set the type to client.

Using the *Registry* tab, set the registry type to *Private registry*:

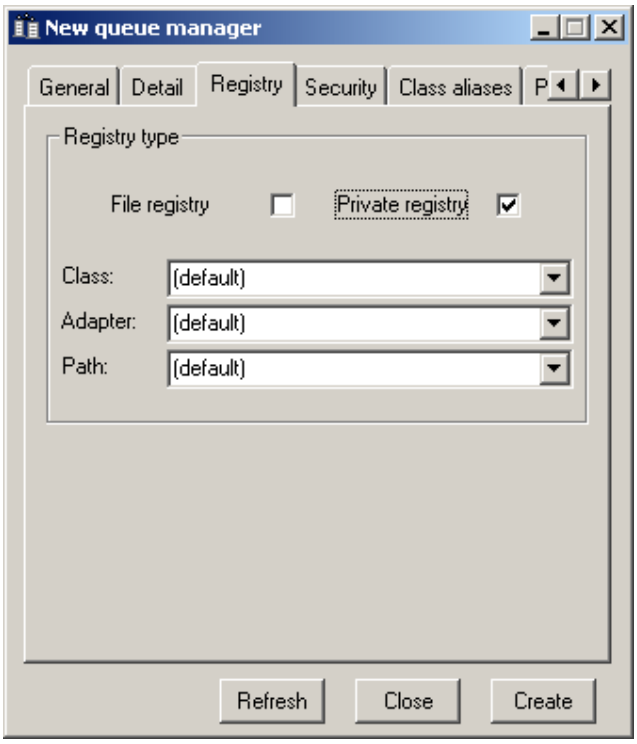

Figure 4-14: Create client queue manager (with credentials) – Registry tab

Switch to the *Security* tab.

As for *myPrivateQMgr*, the *Prompt for passwords* option is used. Additionally *Certificate-based* box is checked, and the *IP address* and *IP port* of the mini-certificate server added. Leave the *Allow queue registry* option unchecked – the use of this option is discussed in *Queue-based security* on page 109.

The default adapter in this case maps to *com.ibm.mqe.adapters.MQeTcpipHttpAdapter*.

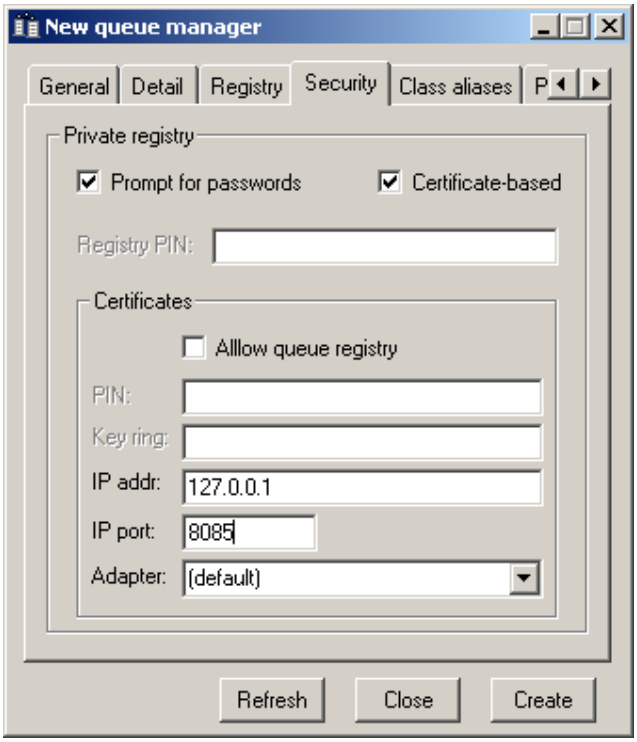

Figure 4-15: Create client queue manager (with credentials) – Security tab

After the *Create* button is clicked, MQe_Explorer will prompt for the passwords needed. As for *myPrivateClientQMgr*, a registry PIN is required:

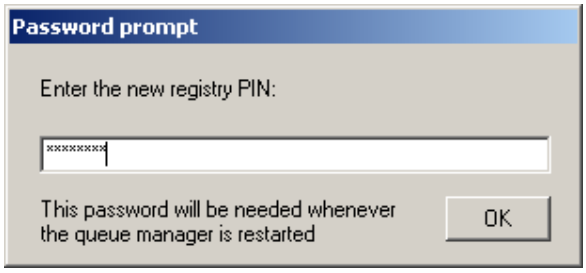

Figure 4-16: Create client queue manager (with credentials) – RegistryPIN prompt

This is subject to the same MQe_Explorer password validation as previously.

A key ring password is also required; used to protect the registry's private key. It is subject to the same MQe_Explorer validity checking as the registry PIN.

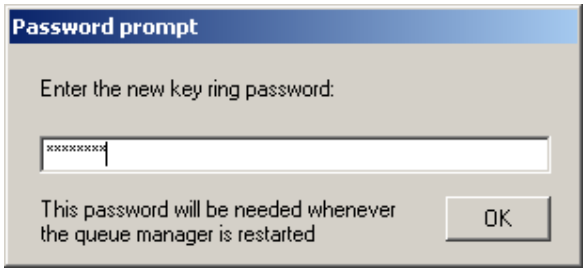

Figure 4-17: Create client queue manager (with credentials) – Key ring password prompt

Finally, the mini-certificate request PIN is required. This is the value that was entered into the mini-certificate server for this authenticatable entity (e.g. the string: 1122334455).

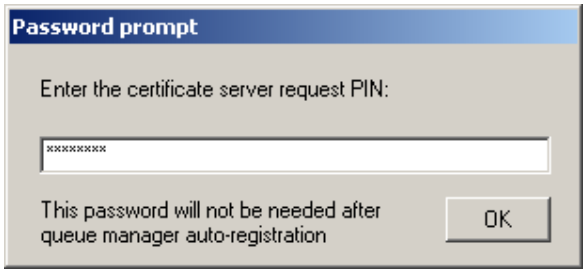

Figure 4-18: Create client qMgr (private registry) – Cert. server request PIN prompt

Clicking the *Create* button creates the new queue manager. During the creation it will obtain the necessary certificates and the details of this auto-registration can be seen in the MQe_MiniCertServer's report window. The summary in the list view pane will need to be manually refreshed to reflect the recent activity.

Details of the certificates acquired are shown in the *Certificates* tab, whenever the properties of the queue manager are displayed, for example:

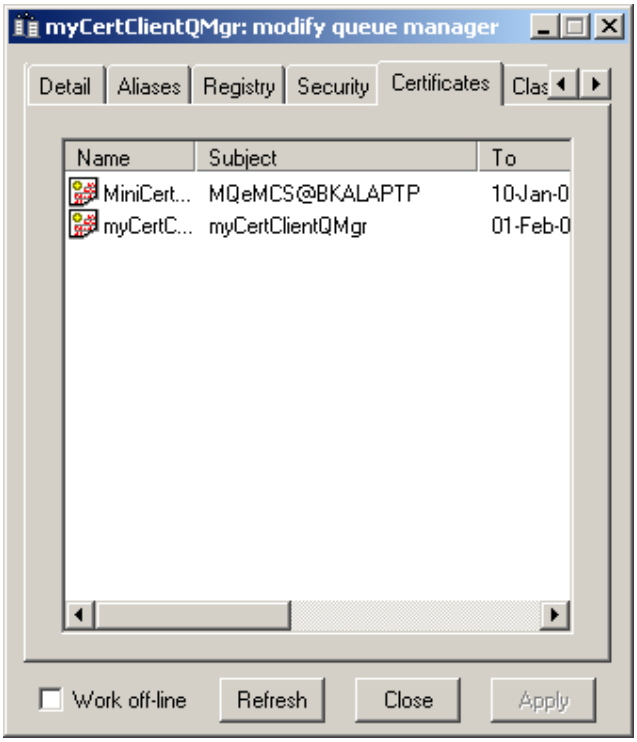

Figure 4-19: Queue manager certificate details

The panel shows two certificates, and gives the name, owner, issuer and validity dates for each.

The initialization data associated with the *myCertClientQMgr* is:

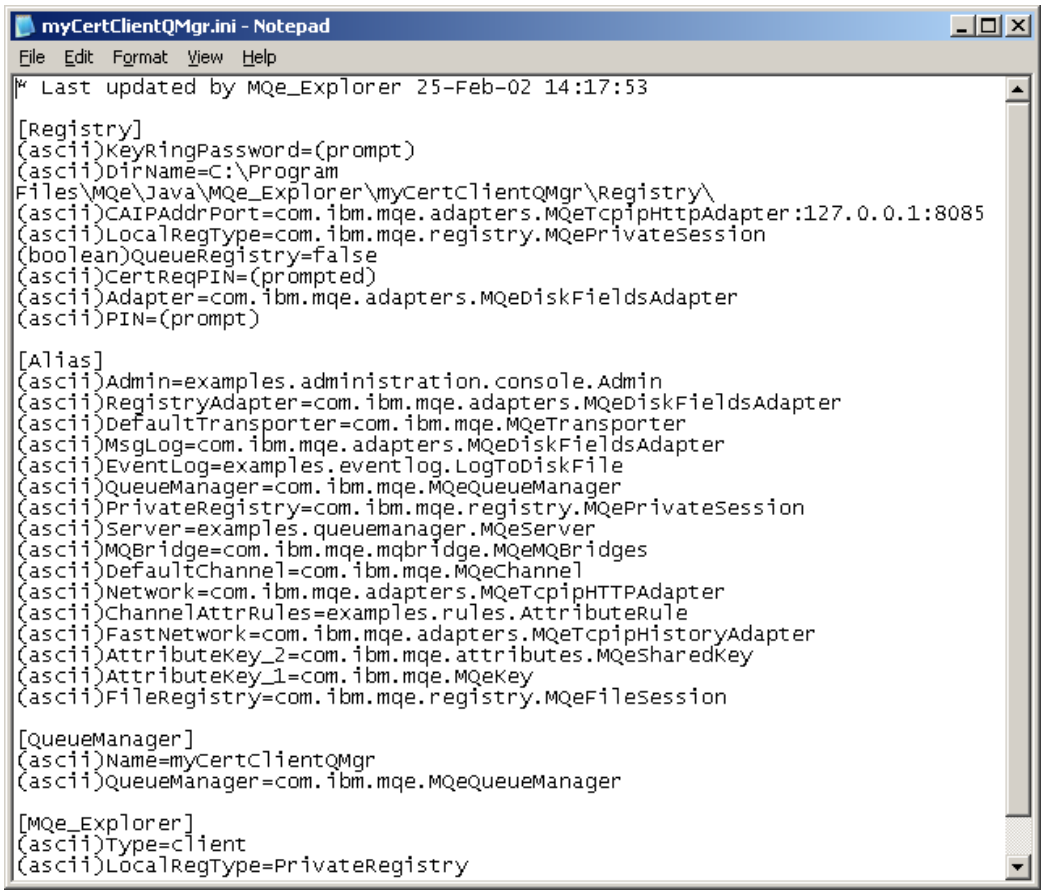

Figure 4-20: The myCertClientQMgr initialization data

Compare this with the same data for *myPrivateQMgr*, a client queue manager with a private registry without credentials, as shown in *Figure 4-6: The myPrivateClientQMgr initialization data* on page 26. Four new entries have appeared in the *[Registry]* section:

1. KeyRingPassword
2. CAIPAddrPort (with a composite value)
3. CertReqPIN
4. QueueRegistry

Thus the two additional password entries have now been added, to join the *PIN* registry password entry previously described. The *CAIPAddrPort* value contains all the information necessary to allow the queue manager to contact the certificate server. The fourth entry *QueueRegistry* is only used by MQe_Explorer and is used to retain the knowledge that the default option, not to allow future queue-based credentials, was selected.

As for a client queue manager with a private registry but no credentials, the code required to set up the environment and then run the queue manager, is the same as that for a queue manager with a file registry. The key call is the creation of *MQeQueueManagerConfigure* object, as shown in *Example 3-4: Creating a client queue manager* on page 17. This instantiation processes the content of the *[Registry]* section – and hence processes *KeyRingPassword*, *CAIPAddrPort* and *CertReqPIN*. However it expects that the *PIN*, *KeyRingPassword* and *CertReqPIN* entries all contain real passwords. It is passed the *environment* MQeFields variable containing these entries, but removes them after using the

values. MQe_Explorer implements the user prompt by pre-parsing the environment variable, requesting passwords from the user and substituting the respective value whenever it finds a "(prompt)" value[13].

It is not necessary that auto-registration take place at the time that the queue manager registry is created. It is equally possible to create a queue manager with a file registry and no credentials, and then to add the credentials later. This is equivalent to adding the relevant three extra *[Registry]* entries, after the queue manager has been created, but before re-starting. In this case the call to the *MQeQueueManagerUtils.processQueueManager()* method initiates the auto-registration – see *Example 3-2: Starting a client queue manager* on page 15.

After auto-registration, it is essential that the *PIN* and *KeyRingPassword* entries in the *[Registry]* section are passed whenever the queue manager is re-started. These entries are needed for access to the certificates. If queue-based authenticatable entities are to be used as well, then the *CertReqPIN* must also be present (see *Setting up a private registry for a queue* on page 112).

The examination and renewal of mini-certificates is discussed in the chapter *Certificate management* on page 133.

[13] The "(prompted)" value is used to identify passwords that are no longer required (e.g. the certificate request PIN is only used once for queue manager, certificate-based credentials).

5 Queue manager properties

In the previous chapter we saw that MQe stores many of the queue manager properties in the registry. Applications programs must remember other properties, and these may be stored in initialization files or in program code.

Queue manager properties in the registry are managed through admin messages; the mechanisms involved are discussed in *Configuration using admin messages* on page 40. Here we examine these properties in more detail.

5.1 Registry-held properties

A queue manager has the following properties held in the MQe queue manager registry:

Queue manager name:
Uniquely identifies the queue manager.

Description:
An arbitrary string describing the queue manager.

Aliases:
Alias names are optional alternative names that are mapped by MQe to this queue manager (see below).

Channel attribute rule:
The class (or alias) of the channel attribute rule associated with the queue manager (for more details see *Channel security* on page 65).

Channel timeout:
The class (or alias) of the rule associated with the queue manager.

Class:
The class (or alias) used to realize the queue manager.

Credentials:
Certificates and associated data (file registries only).

Queue adapter:
The default class (or alias) for a queue store (used as the default property value for queues).

Queue manager rule:
The class (or alias) of the rule associated with the queue manager; determines behavior when there is a change in state for the queue manager.

Queue message store:
The default class (or alias) for a queue store (used as the default property value for queues).

Queue path:
The default location of a queue store (used as the default property value for queues).

Additionally the registry holds details on child objects of the queue manager, such as connections, queues and bridge-related objects (i.e. the bridges object itself, bridges, MQ proxies, client connections and listeners). An admin inquiry enquiry message on the queue manager, as well as soliciting details of queue manager properties, also provides summary details on any child objects, i.e. the queue and connection names and the presence/absence of a bridges object[14].

[14] This feature applies to MQe version 1.27 and later releases.

5.2 Alias names

The queue manager name is used, inter alia, when MQe needs to deliver a message to the local queue manager. Such a message may have originated from either a local or remote queue manager. MQe requires that the target address in the message matches that of the queue manager to which it is delivered. In some cases however a degree of independence is needed between the queue manager addresses used by applications, and the actual names of queue managers in the network. The simplest example is where queue managers have been renamed after the application has been developed. This flexibility is provided through aliases.

Queue managers have associated *aliases*, which allow zero, one or more alias names to be associated with the local queue manager. These aliases can be regarded as changing a matching destination queue manager name in the message, such that it is replaced with the local queue manager name.

Queue manager aliases are not held in the registry as a property of the queue manager; instead they are held as properties of a local connection (see *Local connections* on page 72).

6 Configuration using admin messages

6.1 Introduction

Once a queue manager has been created, further configuration should take place through the sending of admin messages to the target queue manager's *AdminQ*. A queue manager that does not have an *AdminQ* cannot be administered. The intent behind the use of admin messages is that both local and remote administration is performed in an identical manner.

The mechanism is that an admin message is created and sent to the admin queue of the queue manager to be administered. Queue-based security attributes can be applied to that queue if access control is to be enforced (see *Queue-based security* on page 109). The admin message not only includes the details of the request but also indicates whether a response is required, and where that response is to be sent (in the form of an address identifying a queue manager and queue). The *AdminQ* itself acts upon the message and, if requested, results are returned. Admin messages can *inquire* on, *create*, *delete* or *update* objects. For a subset of the objects they can perform additional functions, such as *stop* and *start*.

To view admin messages in action, start MQe_Explorer and open the *myClientQMgr* created earlier. Then:

1. Clear any messages off the *AdminReplyQ* (if you have followed only the instructions above, then the queue should be empty).
2. Set demo mode (*Tools→Demo Mode*) – this will make MQe_Explorer keep a copy of all messages sent & received, on the *AdminReplyQ* .
3. Refresh the *myClientQMgr* icon.
4. Un-set demo mode.
5. Set to view off-line admin (*View→Offline Admin*).
6. Set to view other messages (*View→Others*).

The display will look like this (empty columns have been shrunk in width):

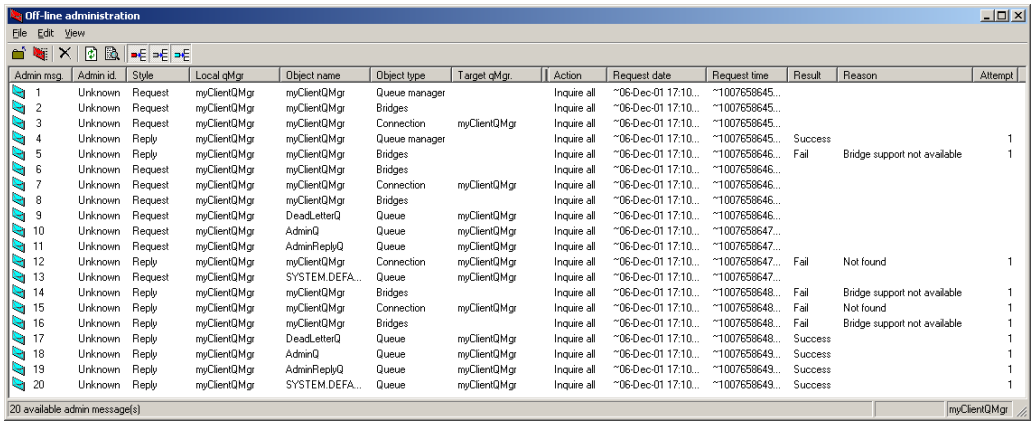

Figure 6-1: Queue manager refresh admin messages

Twenty messages are shown, all have the action type of 'Inquire all', and for every request there is a corresponding reply. The messages are sorted by message number and, in this case, this sequence also corresponds to the time the request (or response) was made. The list represents attempts by MQe_Explorer to get details on all the objects present on the

queue manager. In some cases details were requested on objects that did not exist, such as the bridges object, and a local connection[15].

Using the request (or response) time, the messages can also be viewed in another way; this time the message structure can be examined:

1. Select the *AdminReplyQ* node in the tree pane.
2. Explore the same messages, using the *Creation time* to correlate with the off-line admin window display.
3. Drill down into the messages.

The request message contains the admin instructions required; the corresponding response illustrates how MQe returns the data.

For example, the first request in *Figure 6-1* above is one to get all the details of the *myClientQMgr*. The figure below shows the first level of message structure (using the option to translate field names):

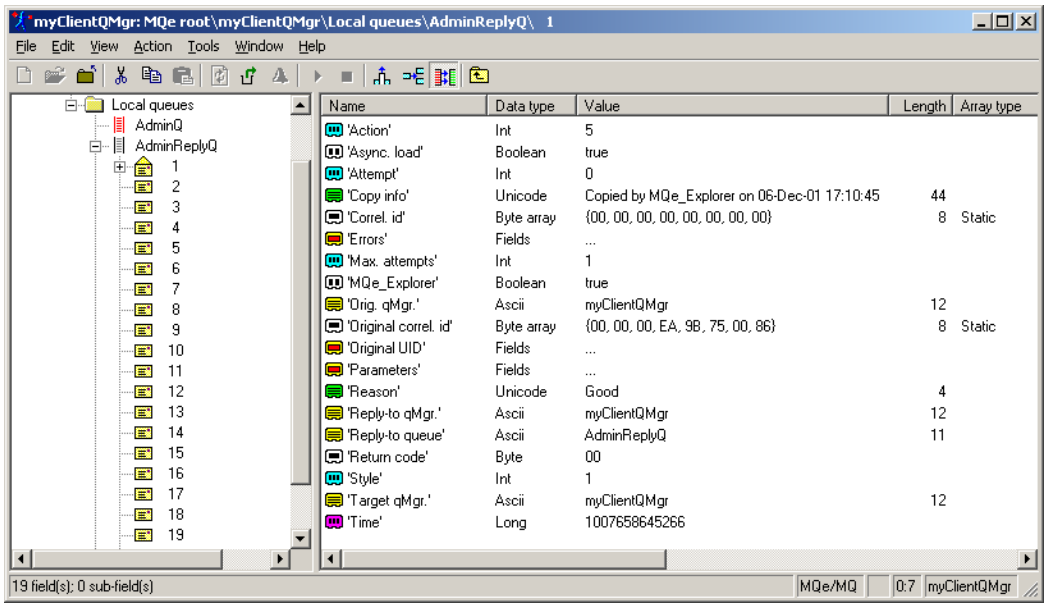

Figure 6-2: First level of a queue manager 'Inquire All' admin request

[15] This appears to be a connection of *myClientQMgr* to itself; local connections have special uses – this topic is discussed later.

The message shown is a copy of the messages actually sent and consequently certain fields have been added or changed by MQe_Explorer to avoid this message causing problems with 'real' admin messages. The differences are explained in detail the *MQe_Explorer User Guide*, but the important fields, from an admin perspective, are unaffected. The fields can be interpreted as:

Action:	5 - an 'Inquire All' request.
Max attempts:	1 – try once.
Target qMgr:	myClientQMgr – target of the request.
Parameters:	Details of the admin request.
Style:	1 – this is a request; send a response message.
Reply-to qMgr:	myClientQMgr – send response to this queue manager.
Reply-to queue:	AdminReplyQ – send response to this queue.

The following fields are present in the request for use in the response (they appear in the message automatically as a consequence of using the admin message constructor):

Attempt:	Used to return attempt number.
Errors:	Used to return errors.
Reason:	Used to return an error reason.
Return code:	Used to return an error code.

The remaining fields are either standard in all MQe messages, or are added by MQe_Explorer for its own purposes, but do not have to be present in admin messages:

Async. load:	Used by MQe_Explorer.
Copy info:	Used by MQe_Explorer for details of the copy.
Correl. id:	Used by MQe_Explorer to correlate request & response.
MQe_Explorer:	Used by MQe_Explorer.
Orig. qMgr:	Always present in an MQe message.
Original correl. id:	Used by MQe_Explorer for details of the original correl. id.
Original UID:	Used by MQe_Explorer for details of the original UID.
Time:	Always present in an MQe message.

The *Parameters* field has the structure:

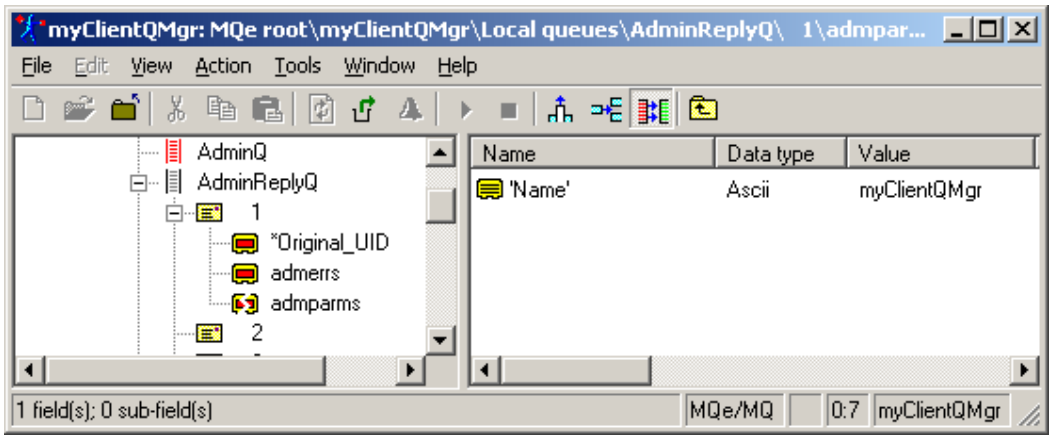

Figure 6-3: Second level of a queue manager 'Inquire All' admin request

The *Name* field present identifies the name of the object to be inquired upon.

The response to the inquiry is message number 4 (in this case) and shows how the information is returned:

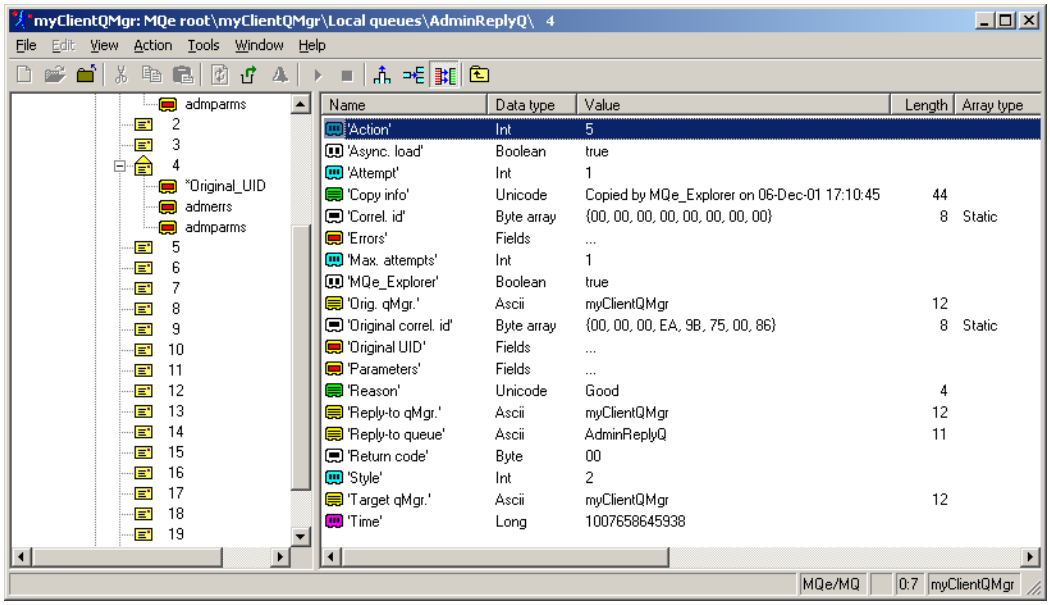

Figure 6-4: First level of a queue manager 'Inquire All' admin response

Only the two fields have changed, but this is because certain reply fields were pre-loaded with values. The important fields are:

Attempt:	1 – this is a reply to the first attempt.
Errors:	Used to return errors (none in this case).
Reason:	Good – the request was successfully processed.
Return code:	00 – the request was successfully processed.

If the *Errors* field is expanded it will be seen to have no constituent fields, i.e. there is no detailed error data. The *Parameters* field has changed considerably and now holds the requested queue manager details:

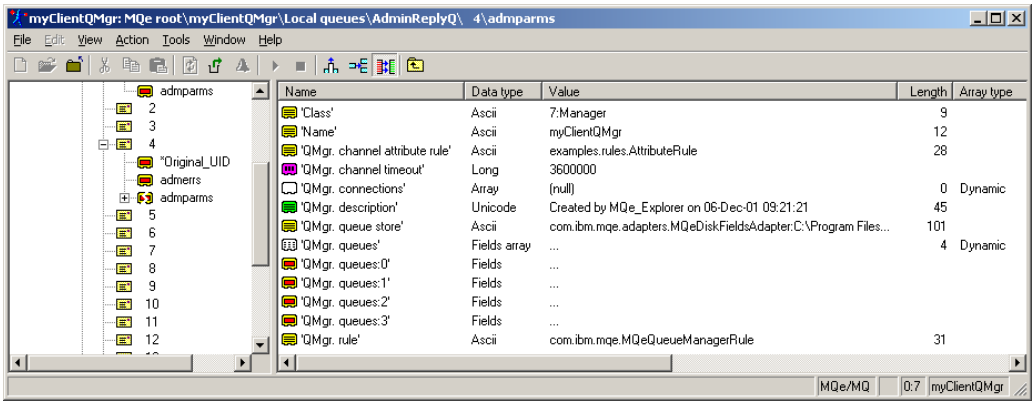

Figure 6-5: Second level of a queue manager 'Inquire All' admin response

This is more complicated to interpret. The fields are:

Class:	Queue manager class (or alias) – abbreviated format.
Name:	Queue manager name.
QMgr. channel attribute rule:	Queue manager channel attribute rule class (or alias).
QMgr. channel timeout:	Queue manager channel timeout.
QMgr. connections:	No connections are defined.
QMgr. description:	The descriptive text.
QMgr. queue store:	The default message store, queue adapter and path in a composite value.
QMgr. queues:	Indicates an MQeFields array present (4 queues).
QMgr. queues:0:	Details of queue 0.
QMgr. queues:1:	Details of queue 1.
QMgr. queues:2:	Details of queue 2.
QMgr. queues:3:	Details of queue 3.
QMgr. rule:	00 – request was successfully processed.

Drilling down into the first queue (queue 0) we see:

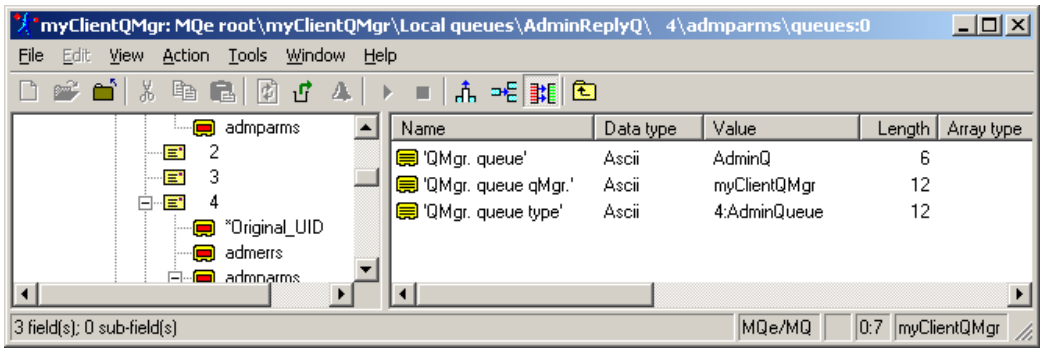

Figure 6-6: Third level of a queue manager 'Inquire All' admin response

This provides sufficient information to identify the queues:

QMgr. queue:	Queue name.
QMgr. queue qMgr.	The queue queue manager name.
QMgr. queue type:	The queue class (or alias) – abbreviated format.

More information on queues (or connections – had there been any) requires additional admin message exchange. In exactly the same way that we have just explored the 'Inquire All' command on a queue manager, the procedure can be repeated for 'Inquire All' on a queue.

Using MQe_Explorer to effect an inquire or a configuration change, followed an inspection of the messages sent and received, is a very useful way of determining what needs to be in an admin message to achieve a particular result. The same process would have shown messages that change property values, create new queues, stop and start bridges objects, and so on.

6.2 MQe_Explorer abstractions

The object and property view presented by MQe_Explorer does not map exactly to the underlying MQe objects and properties. The principal differences affect:

- Queue manager
- Queues
- Connections

In those cases where MQe_Explorer presents a property that does not exist at the MQe object level, MQe_Explorer holds it locally, almost always in its *.ini* file associated with that queue manager.

The queue manager property details are:

MQe_Explorer property	MQe property
Adapter (incoming connection)	Property does not exist (listener config. parameter)
Alias	Held as local connection alias
Bridges	MQe start-up parameter
Certificate-based security	Property does not exist
Certificate server adapter	Property does not exist

MQe_Explorer property	MQe property
Certificate server IP address	Property does not exist
Certificate server IP port	Property does not exist
Certificate server request PIN	Property does not exist
Channel (incoming connection)	Property does not exist (listener config. parameter)
Class aliases	MQe start-up parameter
Enc. parameters (incoming connection)	Property does not exist
IP address (incoming connection)	Property does not exist
IP port (incoming connection)	Property does not exist (listener config. parameter)
Key ring password	MQe start-up parameter
Max. channels (incoming connection)	Property does not exist (listener config. parameter)
Message store	Held as part of the default queue file descriptor
Options (incoming connection)	Property does not exist
Originator	Property does not exist
Parameters (incoming connection)	Property does not exist
Permissions	MQe start-up parameter
Pre-loads	MQe start-up parameter
Prompt for passwords	Property does not exist
Queue adapter	Held as part of the default queue file descriptor
Queue path	Held as part of the default queue file descriptor
Registry adapter	MQe creation parameter
Registry class	MQe start-up parameter
Registry path	MQe start-up parameter
Registry PIN	MQe start-up parameter
Registry type	Registry creation parameter
Rule data (incoming connection)	Property does not exist
Timeout (incoming connection)	Property does not exist (listener config. parameter)
Type	Property does not exist

Figure 6-7: MQe_Explorer queue manager abstractions

The queue property details are:

MQe_Explorer property	MQe property
Message store	Held as part of the queue file descriptor
Adapter	Held as part of the queue file descriptor
Path	Held as part of the queue file descriptor
Type	Property does not exist

Figure 6-8: MQe_Explorer queue abstractions

The connection property details are:

MQe_Explorer property	MQe property
Primary adapter	Held as part of the adapter file descriptor for the first adapter
Primary IP address	Held as part of the adapter file descriptor for the first adapter
Primary IP port	Held as part of the adapter file descriptor for the first adapter
Secondary properties	Properties not currently supported
Type	Property does not exist
Via queue manager	Held as a value of the adapter file descriptor for the first adapter

Figure 6-9: MQe_Explorer connection abstractions

6.3 Admin programming

General principles

Some underlying principles underpin MQe admin programming. In most cases these apply to all MQe objects (i.e. queue managers, queues, connections, MQ bridges, MQ queue manager proxies, MQ client connections and MQ listeners). However the bridge-related objects are treated differently in some important respects; these differences are identified in the next section. General principles:

1. The *AdminQ* queue only accepts messages that derive from the class *com.ibm.mqe.MQeAdminMsg*.
2. Every supplied MQe object that can be administered has an associated admin message class (that inherits from *com.ibm.mqe.MQeAdminMsg*) and this must be used (with rare exceptions).
    i. Queue manager:
    com.ibm.mqe.administration.MQeQueueManagerAdminMsg
    ii. Queue (according to queue type)[16]:
    com.ibm.mqe.administration.MQeAdminQueueAdminMsg
    com.ibm.mqe.administration.MQeHomeServerQueueAdminMsg
    com.ibm.mqe.administration.MQeQueueAdminMsg
    com.ibm.mqe.administration.MQeRemoteQueueAdminMsg
    com.ibm.mqe.administration.MQeStoreAndForwardQueueAdminMsg
    com.ibm.mqe.bridges.MQeMQBridgeQueueAdminMsg
    iii. Connection
    com.ibm.mqe.administration.MQeConnectionAdminMsg
    iv. Bridges object
    com.ibm.mqe.bridges.MQeBridgesAdminMsg
    v. MQ bridge
    com.ibm.mqe.bridges.MQeBridgeAdminMsg
    vi. MQ qMgr. proxy:
    com.ibm.mqe.bridges.MQeMQQMgrProxyAdminMsg

[16] On an inquiry the class *com.ibm.mqe.administration.MQeQueueAdminMsg* can be used irrespective of the target queue class.

vii. MQ client connection:
com.ibm.mqe.bridges.MQeClientConnectionAdminMsg

viii. MQ listener:
com.ibm.mqe.bridges.MQeListenerAdminMsg

3. Defined constants in *com.ibm.mqe.MQeAdminMsg* are used for generic field names and for certain of their standard values.
4. Field names (and standard values) peculiar to a particular object are defined constants in their associated admin message class (but see exception below).
5. Names are ASCII values (but see exception below).
6. Where an object has children, generally the identity of those children will be returned by an inquiry on the parent (but see exception below).
7. Returned abbreviated class names can be expanded by the method *com.ibm.mqe.MQe.abbreviate().*

There are a number of deviations from the general principles above:

1. A queue manager cannot be created by sending a *com.ibm.mqe.administration.MQeQueueManagerAdminMsg* message – local programming is required.
2. Queue manager aliases are handled through the admin messages of class *com.ibm.mqe.administration.MQeConnectionAdminMsg*.
3. A bridges object cannot be created by sending a *com.ibm.mqe.bridges.MQeBridgesAdminMsg* message – local programming is required.
4. A client/server listener cannot be created, modified or deleted through admin messages – local programming is required.
5. The administration of peer listeners is handled through the admin messages of class *com.ibm.mqe.administration.MQeConnectionAdminMsg*.

The general structure of an admin message is:

Action required

Attempt

Max. attempts

Style (request/response)

Reply-to qMgr.

Reply-to queue

Reason

Return code

Detailed parameters

Errors

The actions available are:

MQeAdminMsg.Action_Create: create an administered object.

MQeAdminMsg.Action_Delete: delete an administered object.

MQeAdminMsg.Action_Inquire: inquire on a specific administered object property.

MQeAdminMsg.Action_InquireAll: inquire on all administered object properties.

MQeAdminMsg.Action_Update: delete an administered object.

MQeConnectionAdminMsg.Action_AddAlias: add a connection alias.

MQeConnectionAdminMsg.Action_RemoveAlias: remove a connection alias.

MQeQueueAdminMsg.Action_AddAlias: add a queue alias.

MQeQueueAdminMsg.Action_RemoveAlias: remove a queue alias.

MQeStoreAndForwardQueueAdminMsg.Action_AddQueueManager: add a destination queue manager to a store and forward queue.

MQeStoreAndForwardQueueAdminMsg.Action_RemoveQueueManager: remove a destination queue manager from a store and forward queue.

MQeMQBridgesAdminMsg.Action_Stop: stop bridge-related object(s).

MQeMQBridgesAdminMsg.Action_Start: start bridge-related object(s).

Bridge object specifics

The following differences apply when managing bridge-related objects:

1. All field names (and standard values) that are used in the detailed parameters specification are defined in *com.ibm.mqe.mqbridge.MQeCharacteristicLabels*.
2. The object being managed is identified through an object-specific name, not a generic admin name.
3. Children are identified through a children/child concept.
4. Names are UNICODE values.
5. Bridge-related objects can be stopped and started.

The programming examples later in this chapter illustrate these points.

A gateway queue manager (i.e. a queue manager with a bridges object) can be identified from the value of the *bridgeCapable* property[17] returned from a *MQeQueueManagerAdminMsg* inquiry. If the value is true, then the object is present and the other gateway objects (MQ bridge, MQ proxy, client connection and listener) can be created and/or modified.

6.4 A queue manager inquiry

The following code implements an ‘Inquire All’ query on a remote queue manager.

Many of the admin message classes supply helper methods in order to simplify admin programming; in all the examples below however, these methods are not used. This has been done in order to make it clear exactly what is required in the message. In any serious

[17] This feature was added in MQe version 1.27. On earlier releases an MQeBridgesAdminMsg had to be sent; if a reply free from errors was received, then the queue manager was a gateway.

admin application it is generally useful to develop custom helper methods – admin programming being somewhat repetitive.

Creating and sending the message

The code below uses the local queue manager to create the message, and then sends it to the *AdminQ* of the target queue manager called 'myTargetQMgr'. Admin replies are returned to the *AdminReplyQ* on the local queue manager. The same code would be used to inquire on the local queue manager – the only change would be to insert the appropriate name in the *targetQMgrName* variable.

```
//get addressability to the local queue manager
MQeQueueManager qMgr = MQeQueueManager.getReference(null);

//set up strings
String thisQMgrName = qMgr.getName();                          //local qMgr name
String targetQMgrName = "myClientQMgr";                        //target qMgr name

//create the basic queue manager admin message
MQeQueueManagerAdminMsg adminMsg = new MQeQueueManagerAdminMsg();

//add command-neutral elements
adminMsg.putAscii(MQeAdminMsg.Admin_TargetQMgr, targetQMgrName);//set target qMgr
adminMsg.putInt(MQeAdminMsg.Admin_MaxAttempts, 1);             //set max tries
adminMsg.putInt(MQe.Msg_Style, MQe.Msg_Style_Request);         //need a reply
adminMsg.putAscii(MQe.Msg_ReplyToQMgr, thisQMgrName);          //reply queue manager
adminMsg.putAscii(MQe.Msg_ReplyToQ, MQe.Admin_Reply_Queue_Name);//reply queue

//add unique reference
byte[] adminKey = MQe.longToByte(System.currentTimeMillis());  //generate unique key
adminMsg.putArrayOfByte(MQe.Msg_CorrelID, adminKey);           //add key

//add inquiry command
adminMsg.putInt(MQeAdminMsg.Admin_Action, MQeAdminMsg.Action_InquireAll);

//add command parameters
MQeFields parms = new MQeFields();                             //new parms object
parms.putAscii(MQeAdminMsg.Admin_Name, targetQMgrName);        //admin object
adminMsg.putFields(MQeAdminMsg.Admin_Parms, parms);            //add parms

//send admin message
qMgr.putMessage(targetQMgrName, MQe.Admin_Queue_Name, adminMsg , null, 0);
```

Example 6-1: A queue manager 'Inquire All' query

This code can be mapped directly to the MQe_Explorer messages seen earlier. The only curious aspect of the code is the nature and use of the *adminKey* variable. This is added to the message in order that the reply can be correlated with the original. The use of the system time is an easy way to generate a suitable value (provided that only occasional messages are issued)[18]. Transforming it into an array of bytes is odd; it is done here because MQe_Explorer also chooses to do that. Use of a long value would be more natural, and any field name would suffice. The MQe examples also use the correlation id in this way – primarily to mirror MQ base messaging programming style.

[18] A more certain way of generating unique values is provided by the *MQe.uniqueValue()* method.

If the intent is synchronous administration, then no pre-configuration of queues is required. The standard defaults should suffice. If the remote *AdminQ* does not have a corresponding local remote queue definition, then MQe will generate one dynamically. Other error situations are considered later in this chapter.

Getting the reply

There are two ways in which the reply can be retrieved. The synchronous way is to wait, for example:

```
//create a filter
MQeFields msgFilter = new MQeFields();                          //new filter
msgFilter.putArrayOfByte(MQe.Msg_CorrelID, adminKey);           //add key

//wait for reply
int waitTime = 10000;                                           //wait 10 secs
MQeAdminMsg adminReply = (MQeAdminMsg) qMgr.waitForMessage(
        thisQMgrName , MQe.Admin_Reply_Queue_Name,
        msgFilter, null, 0, waitTime);                          //get reply
```

Example 6-2: Waiting for a reply

If no reply is received, an exception is thrown.

Alternatively a message listener can be set-up before the admin message is sent, as below:

```
//create a filter
MQeFields msgFilter = new MQeFields();                          //new filter
msgFilter.putArrayOfByte(MQe.Msg_CorrelID, adminKey);           //add key

//set up message listener
qMgr.addMessageListener(this, MQe.Admin_Reply_Queue_Name,
        msgFilter);
```

Example 6-3: Setting up a message listener

The following code will then be called when a reply arrives:

```
//called when admin reply arrives
public void messageArrived (MQeMessageEvent e) throws Exception
{
        //get the reply
        MQeAdminMsg adminReply = qMgr.getMessage(thisQMgrName ,
                MQe.Admin_Reply_Queue_Name, msgFilter, null, 0);
}
```

Example 6-4: Getting the reply when an event is raised

This code uses the same message filter in two places; firstly to get the event raised when the reply is received, and subsequently to retrieve it from the *AdminReplyQ*. More typically, a generic message listener would be established that is capable of getting replies to multiple messages (i.e. by using a less restrictive message filter); each response could then be retrieved by:

```
//called when one of a number of admin replies are received
public void messageArrived (MQeMessageEvent e) throws Exception
{
        //get a particular reply
        eventFilter = e.getMsgFields();                          //get message data
        MQeAdminMsg adminReply = qMgr.getMessage(thisQMgrName,
                MQe.Admin_Reply_Queue_Name, eventFilter, null, 0);
}
```

Example 6-5: Getting a specific reply when an event is raised

The *getMsgFields()* method returns the *MQeFields* object that caused the event. Using this to retrieve the message, ensures that the message that caused the event is removed from the queue monitored by the message listener.

Extracting the information

The requested data is returned in the command parameters. Originally only the name of the administered object was set in the inquiry, i.e. the queue manager name in this case. On a successful inquiry, all the properties have now been added to the response, e.g.

```
//extract the command parameters
parms = adminReply.getFields(MQeAdminMsg.Admin_Parms);

//extract each property
String descr = parms.getUnicode(
        MQeQueueManagerAdminMsg.QMgr_Description);          //description
long chTimeout = parms.getLong(
        MQeQueueManagerAdminMsg.QMgr_ChnlTimeout);          //channel timeout


//   etc.
```

Example 6-6: Extracting the responses to the 'Inquire All' query

Executing the sample – *Queue manager inquiry*:

This depends upon the existence of the configured *myClientQMgr* queue manager created earlier.

Then use the command: *MQeConfigGuide 3*. The sample code is contained in the method *inquireClientQMgr()*. By default, the MQe_Explorer-created *myClientQMgr.ini* file is used; by editing the sample code, the method *createClientEnvironment()* can be used instead.

Handling errors

The two likely places where errors might occur are either on the sending of the admin message, or on its receipt. Assuming synchronous administration, sending errors are likely to be the result of remote queue managers being unavailable, or of networking problems. These are presented as exceptions on the *putMessage()*, i.e.

```
//send admin message
try
{
        qMgr.putMessage(targetQMgrName, MQe.Admin_Queue_Name, adminMsg , null, 0);
}
catch (MQeException mqe)
{
        //handle MQe error
        int errCode = mqe.code();

}
catch (java.io.IOException ioe)
{
        //handle I/O error
}
catch (Exception e)
{
        //handle other errors
}
```

Example 6-7: Sending errors for synchronous admin

Error handling whilst processing the reply is more complicated. Whilst this particular example of an 'Inquire All' is unlikely to produce complex errors, the general philosophy is described below and is relevant when updates to MQe object properties are attempted. Error notification can happen:

1. As an exception on the *getMessage()* or *waitForMessage().*

   The normal reason for an exception here is that there is no admin reply on the *AdminReplyQ*.

2. In the *Reason* and *Return code* parameters.

   The *Reason* text parameter is used for overall admin errors, for example that the object being administered does not exist, or that an attempt is being made to create an object that already exists. The *Return code* has three possible values:

   i. *MQeAdminMsg.RC_Success* – complete success.
   ii. *MQeAdminMsg.RC_Fail* – complete failure.
   iii. *MQeAdminMsg.RC_Mixed* – partial success.

3. In the *Errors* array

   Errors is used to give error information on a per-property basis.

The example below shows a way in which error information can be handled:

```
//send admin message
try
{
        MQeAdminMsg adminReply = qMgr.getMessage(localQMgrName,
                MQe.Admin_Reply_Queue_Name, eventFilter, null, 0);

        //check for errors
        if (adminReply.getByte(MQeAdminMsg.Admin_RC) != MQeAdminMsg.RC_Success)
        {
                //errors occurred – get reason
                String reason = adminReply.getUnicode(MQeAdminMsg.Admin_Reason);

                //get detail
                MQeFields errors = adminReply.getFields(MQeAdminMsg.Admin_Errors);
                if (errors != null)
                {
                        Enumeration contents = errors.fields();
                        while (contents.hasMoreElements())
                        {
                                //get property name
                                String property = (String) contents.nextElement();
                                //get associated error
                                String error = errors.getAscii(property));
                        }
                }
        }
}
catch (MQeException mqe)          {//handle MQe errors....          }
catch (java.io.IOException ioe)   {//handle I/O errors ....        }
catch (Exception e)               {//handle other errors ....      }
```

Example 6-8: Error handling when processing admin replies

A successful return code indicates that no error information is present; any other value indicates that either or both of the *Reason* and *Errors* parameters contain error information.

6.5 Other admin examples

The above example was a simple inquiry; more complex examples below include the setting of property values and the handling of bridge-related objects. Only message creation is shown; extracting information and error handling is substantially the same as the inquiry case above.

Setting a queue manager property

The example below shows how to set the channel timeout property of a queue manager. The example uses the local queue manager to modify a remote queue manager called 'myTargetQMgr'. Admin replies are returned to the *AdminReplyQ* on the local queue manager.

```
//get addressability to the local queue manager
MQeQueueManager qMgr = MQeQueueManager.getReference(null);

//set up strings
String thisQMgrName = qMgr.getName();                          //local qMgr name
String targetQMgrName = "myClientQMgr";                        //target qMgr name

//create the basic queue manager admin message
MQeQueueManagerAdminMsg adminMsg = new MQeQueueManagerAdminMsg();

//add command-neutral elements
adminMsg.putAscii(MQeAdminMsg.Admin_TargetQMgr, targetQMgrName);//set target qMgr
adminMsg.putInt(MQeAdminMsg.Admin_MaxAttempts, 1);             //set max tries
adminMsg.putInt(MQe.Msg_Style, MQe.Msg_Style_Request);         //need a reply
adminMsg.putAscii(MQe.Msg_ReplyToQMgr, thisQMgrName);          //reply queue manager
adminMsg.putAscii(MQe.Msg_ReplyToQ, MQe.Admin_Reply_Queue_Name);//reply queue

//add unique reference
byte[] adminKey = MQe.longToByte(System.currentTimeMillis());  //generate unique key
adminMsg.putArrayOfByte(MQe.Msg_CorrelID, adminKey);           //add key

//add update command
adminMsg.putInt(MQeAdminMsg.Admin_Action, MQeAdminMsg.Action_Update);

//add command parameters
MQeFields parms = new MQeFields();                             //new parms object
parms.putAscii(MQeAdminMsg.Admin_Name, targetQMgrName);        //admin object
parms.putAscii(MQeAdminMsg.Admin_Class,
        "com.ibm.MQeQueueManager");                            //object class
parms.putLong(MQeQueueManagerAdminMsg.QMgr_ChnlTimeout,
        1000000L);                                             //new timeout
adminMsg.putFields(MQeAdminMsg.Admin_Parms, parms);            //add parms

//send admin message
qMgr.putMessage(targetQMgrName, MQe.Admin_Queue_Name, adminMsg , null, 0);
```

Example 6-9: Setting the channel timeout on a queue manager

Compared with the queue manager 'Inquire All' example, the points to note are:

1. The *update* command has been specified.
2. The object class has been added to the parameters.
3. The channel timeout has been added to the parameters.

Executing the sample – *Set a queue manager property*:

This depends upon the existence of the configured *myClientQMgr* queue manager created earlier.

Then use the command: *MQeConfigGuide 4*. The sample code is contained in the method *setPropertyClientQMgr().* By default, the MQe_Explorer-created *myClientQMgr.ini* file is used; by editing the sample code, the method *createClientEnvironment()* can be used instead.

Creating a new queue

The example below shows how to create a remote queue definition; only a subset of the possible parameters is specified – the rest will take their default values. The example uses the local queue manager to create a remote queue called 'myQueue', on a queue manager called 'myTargetQMgr'. The remote queue is a proxy for a local queue called 'myQueue' on a queue manager called 'remoteQMgr'. It implements asynchronous messaging, with a transporter that uses XOR compression. Admin replies are returned to the *AdminReplyQ* on the local queue manager.

```
//get addressability to the local queue manager
MQeQueueManager qMgr = MQeQueueManager.getReference(null);

//set up strings
String thisQMgrName = qMgr.getName();                          //local qMgr name
String targetQMgrName = "myClientQMgr";                        //target qMgr name
String queueQMgrName = "remoteQMgr";                           //remote qMgr name
String queueName = "myQueue";                                  //queue name - target
String descrText = "My remote queue";                          //description

//create the basic queue admin message
MQeRemoteQueueAdminMsg adminMsg = new MQeRemoteQueueAdminMsg();

//add command-neutral elements
adminMsg.putAscii(MQeAdminMsg.Admin_TargetQMgr, targetQMgrName);//set target qMgr
adminMsg.putInt(MQeAdminMsg.Admin_MaxAttempts, 1);             //set max tries
adminMsg.putInt(MQe.Msg_Style, MQe.Msg_Style_Request);         //need a reply
adminMsg.putAscii(MQe.Msg_ReplyToQMgr, thisQMgrName);          //reply queue manager
adminMsg.putAscii(MQe.Msg_ReplyToQ, MQe.Admin_Reply_Queue_Name);//reply queue

//add unique reference
byte[] adminKey = MQe.longToByte(System.currentTimeMillis());  //generate unique key
adminMsg.putArrayOfByte(MQe.Msg_CorrelID, adminKey);           //add key

//add create command
adminMsg.putInt(MQeAdminMsg.Admin_Action, MQeAdminMsg.Action_Create);

//add command parameters
MQeFields parms = new MQeFields();                             //new parms object
parms.putAscii(MQeAdminMsg.Admin_Name, queueName);             //queue name
parms.putAscii(MQeAdminMsg.Admin_Class,
        "com.ibm.MQeRemoteQueue");                             //object class
parms.putAscii(MQeQueueAdminMsg.Queue_QMgrName,
        queueQMgrName);                                        //queue qMgr. name
parms.putUnicode(MQeQueueAdminMsg.Queue_Description,
        descrText);                                            //queue description
parms.putByte(MQeQueueAdminMsg.Queue_Mode,
        MQeQueueAdminMsg.Queue_Asynchronous);                  //asynchronous mode
msgParms.putBoolean(MQeRemoteQueueAdminMsg.Queue_TransporterXOR,
        true);                                                 //XOR compress
adminMsg.putFields(MQeAdminMsg.Admin_Parms, parms);            //add parms

//send admin message
qMgr.putMessage(targetQMgrName, MQe.Admin_Queue_Name, adminMsg , null, 0);
```

Example 6-10: Creating a new remote queue

Compared with the queue manager ‘Inquire All’ example, the points to note are:

1. The admin message class has been changed to match the queue class being created.
2. The *create* command has been specified.
3. The remote queue object class has been added to the parameters.
4. The admin name is now the name of the new queue being created
5. The queue qMgr. name has been added (needed to identify the queue).
6. The queue description, mode and transporter XOR compress have been set.

Adding a connection alias

The example below shows how to add a connection alias to an existing connection. The example uses the local queue manager to add an alias 'aliasRemoteQMgr' to a connection called 'RemoteQMgr', on a queue manager called 'myTargetQMgr'. Admin replies are returned to the *AdminReplyQ* on the local queue manager.

```
//get addressability to the local queue manager
MQeQueueManager qMgr = MQeQueueManager.getReference(null);

//set up strings
String thisQMgrName = qMgr.getName();                          //local qMgr name
String targetQMgrName = "myClientQMgr";                        //target qMgr name
String connName = "RemoteQMgr";                                //connection name
String aliasName = "aliasRemoteQMgr"                           //alias name

//create the basic connection admin message
MQeConnectionAdminMsg adminMsg = new MQeConnectionAdminMsg();

//add command-neutral elements
adminMsg.putAscii(MQeAdminMsg.Admin_TargetQMgr, targetQMgrName);//set target qMgr
adminMsg.putInt(MQeAdminMsg.Admin_MaxAttempts, 1);             //set max tries
adminMsg.putInt(MQe.Msg_Style, MQe.Msg_Style_Request);         //need a reply
adminMsg.putAscii(MQe.Msg_ReplyToQMgr, thisQMgrName);          //reply queue manager
adminMsg.putAscii(MQe.Msg_ReplyToQ, MQe.Admin_Reply_Queue_Name);//reply queue

//add unique reference
byte[] adminKey = MQe.longToByte(System.currentTimeMillis());  //generate unique key
adminMsg.putArrayOfByte(MQe.Msg_CorrelID, adminKey);           //add key

//add add alias command
adminMsg.putInt(MQeAdminMsg.Admin_Action, MQeConnectionAdminMsg.Action_AddAlias);

//add command parameters
MQeFields parms = new MQeFields();                             //new parms object
parms.putAscii(MQeAdminMsg.Admin_Name, connName);              //connection name
parms.putAscii(MQeAdminMsg.Admin_Class,
        "com.ibm.mqe.MQeConnectionDefinition");                //object class
String[] aliasesArray = new String[1];                         //alias array
aliasesArray[0] = aliasName;                                   //load aliases
parms.putAsciiArray(MQeConnectionAdminMsg.Con_Aliases,
        aliasesArray);                                         //add aliases
adminMsg.putFields(MQeAdminMsg.Admin_Parms, parms);            //add parms

//send admin message
qMgr.putMessage(targetQMgrName, MQe.Admin_Queue_Name, adminMsg , null, 0);
```

Example 6-11: Adding a connection alias

Compared with the queue manager 'Inquire All' example, the points to note are:

1. The admin message class has been changed to reflect a connection definition.
2. The *add alias* command has been specified.
3. The connection object class has been added to the parameters.
4. The admin name is now the name of the connection being modified
5. The alias name has been set.

Inquiry on an MQ bridge

The code below uses the local queue manager to inquire on an MQ bridge called 'myMQBridge' hosted on a queue manager called 'myTargetQMgr'. Admin replies are returned to the *AdminReplyQ* on the local queue manager.

```
//get addressability to the local queue manager
MQeQueueManager qMgr = MQeQueueManager.getReference(null);

//set up strings
String thisQMgrName = qMgr.getName();                          //local qMgr name
String targetQMgrName = "myClientQMgr";                        //target qMgr name
String bridgeName = "myMQBridge";                              //MQ bridge name

//create the basic MQ bridge admin message
MQeMQBridgeAdminMsg adminMsg = new MQeMQBridgeAdminMsg();

//add command-neutral elements
adminMsg.putAscii(MQeAdminMsg.Admin_TargetQMgr, targetQMgrName);//set target qMgr
adminMsg.putInt(MQeAdminMsg.Admin_MaxAttempts, 1);             //set max tries
adminMsg.putInt(MQe.Msg_Style, MQe.Msg_Style_Request);         //need a reply
adminMsg.putAscii(MQe.Msg_ReplyToQMgr, thisQMgrName);          //reply queue manager
adminMsg.putAscii(MQe.Msg_ReplyToQ, MQe.Admin_Reply_Queue_Name);//reply queue

//add unique reference
byte[] adminKey = MQe.longToByte(System.currentTimeMillis());  //generate unique key
adminMsg.putArrayOfByte(MQe.Msg_CorrelID, adminKey);           //add key

//add inquiry command
adminMsg.putInt(MQeAdminMsg.Admin_Action, MQeAdminMsg.Action_InquireAll);

//add command parameters
MQeFields parms = new MQeFields();                             //new parms object
parms.putUnicode(MQeCharacteristicLabels.MQE_FIELD_LABEL_BRIDGE_NAME,
        bridgeName);                                           //MQ bridge name
adminMsg.putFields(MQeAdminMsg.Admin_Parms, parms);            //add parms

//send admin message
qMgr.putMessage(targetQMgrName, MQe.Admin_Queue_Name, adminMsg , null, 0);
```

Example 6-12: An MQ bridge 'Inquire All' query

Compared with the queue manager 'Inquire All' example, the points to note are:

1. The admin message class has been changed to reflect an MQ bridge object.
2. The admin name in the command parameters is replaced by the bridge name; the value is UNICODE instead of ASCII.

Setting an MQ listener property

The code below uses the local queue manager to set the description property of an MQ listener. The listener is called 'myMQListener', with a parent called 'myMQClientConnection', itself with a parent 'myMQProxy', on an MQ bridge called 'myMQBridge', hosted on a queue manager 'myTargetQMgr'. Admin replies are returned to the *AdminReplyQ* on the local queue manager.

```
//get addressability to the local queue manager
MQeQueueManager qMgr = MQeQueueManager.getReference(null);

//set up strings
String thisQMgrName = qMgr.getName();                          //local qMgr name
String targetQMgrName = "myTargetQMgr";                        //target qMgr name
String bridgeName = "myMQBridge";                              //MQ bridge name
String proxyName = "myMQProxy";                                //MQ proxy name
String clientConnName = "myMQClientConnection";                //MQ client conn name
String listenerName = "myMQListener";                          //MQ listener name
String descrText = "My listener";                              //description

//get addressability to the local queue manager
MQeQueueManager qMgr = MQeQueueManager.getReference(thisQMgrName);

//create the basic MQ listener admin message
MQeListenerAdminMsg adminMsg = new MQeListenerAdminMsg();

//add command-neutral elements
adminMsg.putAscii(MQeAdminMsg.Admin_TargetQMgr, targetQMgrName);//set target qMgr
adminMsg.putInt(MQeAdminMsg.Admin_MaxAttempts, 1);             //set max tries
adminMsg.putInt(MQe.Msg_Style, MQe.Msg_Style_Request);         //need a reply
adminMsg.putAscii(MQe.Msg_ReplyToQMgr, thisQMgrName);          //reply queue manager
adminMsg.putAscii(MQe.Msg_ReplyToQ, MQe.Admin_Reply_Queue_Name);//reply queue

//add unique reference
byte[] adminKey = MQe.longToByte(System.currentTimeMillis());  //generate unique key
adminMsg.putArrayOfByte(MQe.Msg_CorrelID, MQeWatcher.adminKey); //add key

//add update command
adminMsg.putInt(MQeAdminMsg.Admin_Action, MQeAdminMsg.Action_Update);

//add command parameters
MQeFields parms = new MQeFields();                             //new parms object
parms.putUnicode(MQeCharacteristicLabels.MQE_FIELD_LABEL_LISTENER_NAME,
        listenerName);                                         //MQ listener name
parms.putUnicode(MQeCharacteristicLabels.MQE_FIELD_LABEL_CLIENT_CONNECTION_NAME,
        clientConnName);                                       //MQ client conn name
parms.putUnicode(MQeCharacteristicLabels.MQE_FIELD_LABEL_MQ_Q_MGR_PROXY_NAME,
        proxyName);                                            //MQ proxy name
parms.putUnicode(MQeCharacteristicLabels.MQE_FIELD_LABEL_BRIDGE_NAME,
        bridgeName);                                           //MQ bridge name
parms.putUnicode(MQeCharacteristicLabels.MQE_FIELD_LABEL_DESCRIPTION,
        descrText);                                            //description
adminMsg.putFields(MQeAdminMsg.Admin_Parms, parms);            //add parms

//send admin message
qMgr.putMessage(targetQMgrName, MQe.Admin_Queue_Name, adminMsg , null, 0);
```

Example 6-13: Setting the description on an MQ listener

Compared with the setting of a queue manager property example, the points to note are:

1. The admin message class has been changed to reflect an MQ listener object.
2. The admin name in the command parameters is no longer present. It is replaced by multiple UNICODE parameters that collectively identify the listener.
3. All parameter field names are taken from the *com.ibm.mqe.mqbridge.MQeCharacteristicLabels* class.

7 Channels and transporters

Channels are the objects used by MQe to move messages between queue managers. The creation and management of channels is the responsibility of queue managers – there is no configuration or other user involvement in channel operation. Channels have security attributes reflecting the level of protection that must be given to their traffic – again MQe handles this automatically. Channels use communications adapters to provide access to the underlying communications infrastructure (e.g. TCP/IP, HTTP, UDP etc).

Given this – there would seem to be no obvious reason why channels need to be understood by administrators. However the reason that channels surface is that MQe offers two channel types – and the type to be used to communicate to any particular remote queue manager is selectable. It is specified as a parameter on the connection definition; hence it is controllable at the remote queue manager level. More details on connection definitions follow in the chapter – to begin, the two channel types will be described. These are:

- Client/server
- Peer-to-peer

The channel type in use has no implications for MQe application programming; applications are unaware of channel type. The only impact of channel type is to affect message flow initiation between queue managers.

7.1 Client/server channels

In client/server communication, the client and the server play two distinct roles:

Client: initiates transfers of data – transfers can be either to or from the server.

Server: responds to requests from the client.

The essence of client/server is that the client is in control of data transfers; the server cannot force data on to or off the client; the client must, in all cases, initiate data transfers. Transfers, when they take place may be uni- or bi-directional[19].

Although client/server behavior may seem unduly restrictive, this may be exactly what is required. Clients may not want unsolicited data transfer initiated by a server; or alternatively the presence of firewalls in the network may require precise control over which party initiated a data transfer.

The client/server characteristics are on a per-channel basis. So, for any one client-server channel the above is true; and indeed, all channels arising from a single connection definition will have the same characteristics. However, the following is also possible, and very common. Two queue managers, *A* and *B*, wish to exchange data. Queue manager *A* has a connection definition to *B* such that all channels instigated by *A* to *B* are client/server, where *A* is the client and *B* is the server. Likewise, queue manager *B* has a connection definition to *A* such that all channels instigated by *B* to *A* are client/server, where *B* is the client and *A* is the server. Now both *A* and *B* can be simultaneously clients and servers – and either can initiate freely data transfers to each other, and all data is being transferred over client/server channels.

In client/server communication, in order to set up a channel, the client must first make a connection request. This requires knowledge of the address of the server, and of the port being listened on by the server for such incoming connection requests. The server needs to have such a listener active but does not need any prior knowledge of the address or port being used by the client.

---

[19] How a server sends data to a client is discussed under *home server queues*.

MQe provides a client/server channel listener that listens on a single port for multiple incoming connection requests from clients. Each client request is then handed off to another port for the socket to be created; thus one port (with a fixed port number) can be continuously available to service incoming requests. The client/server listener is described in the section *Client/server listeners* on page 82.

7.2 Peer-to-peer channels

In peer-to-peer communication, both parties are equal and have identical capabilities. However two distinct roles can be identified, which either can play:

Master: initiates the establishment of the channel.

Slave: responds to the establishment request from the master.

Once a channel is established it is bi-directional and either peer can use it to send data to the other.

In peer-to-peer communication, in order to set up a channel, the master must first make a connection request. This requires knowledge of the address of the slave, and of the port being listened on by the slave for such incoming connection requests. The slave needs to have such a listener active but does not need any prior knowledge of the address or port being used by the master.

MQe provides a peer channel listener that listens on a single port for an incoming connection request from a master. It then uses that same port to create the sockets for the subsequent channel(s). The port is not free to service another incoming request whilst the channel(s) exist. The peer listener is described in the section *Peer listeners* on page 84.

7.3 Channel types compared

The principal differences between MQe channel types are summarized in the following table:

Property	Client/server	Peer-to-peer
Initiator	Client only	Either peer
Bi-directional data flows	Client can send data Client can pull data Server cannot send data	Either peer can send data Either peer can pull data
Channel duration	Timeout set by client	Timeout set by master
Addressability	Client needs address of server listener Server does not need addressability to client	Master needs address of slave listener Slave does not need addressability to client
Listener port	Server can listen for multiple incoming requests on a single port Client does not need a listener	Slave can listen for one incoming request at a time Master does not need a listener
Default class name	*com.ibm.mqe.MQeChannel*	*com.ibm.mqe.MQePeerChannel*

Figure 7-1: Channel type comparison

The principal disadvantages of peer-to-peer channels arise because the peer channel listener can only listen for one request at a time. The other problem with peer-to-peer channels is that the channel timeout is set by the master; if the slave is using a channel created by a master then there is no guarantee that the channel will be there when needed by the slave. This problem becomes irrelevant if the slave queue manager has a connection definition to the master queue manager, because then a new channel capable of sending to the master can be created when needed.

The principal disadvantage of client/server channels is that the server cannot use them to send data to the client. Arrangements must be made (see later) for the client to pull data from the server. Again this problem becomes irrelevant if the server queue manager has a connection definition to the client queue manager, because then a new channel capable of sending to the 'client' can be created when needed.

If in doubt, by default client/server channels should be configured in connection definitions – and peer-to-peer channels used only if their specific capabilities are known to be required.

7.4 Channel security

Channels have an associated security attribute that is managed automatically by MQe. However channel security behavior is controlled through an attribute rule; this rule is configured for a queue manager (see *Registry-held properties* on page 38); a related rule is set for queues (see *Queues* on page 102).

The process is as follows. When a local queue manager needs to send a message to a target queue on a remote queue manager, it first retrieves the security details of that target queue. This will typically be from a remote queue definition stored locally – though if the definition is not present it will attempt to discover the details from the remote queue manager. The local queue manager then attempts to re-use any existing channel to the target queue manager – with the local queue manager's attribute rule being used to determine whether such a channel can be re-used (or possibly upgraded). Depending upon the results from the rule, a channel is re-used or a new channel created. At the remote queue manager, the attribute rule associated with the target queue is consulted to determine whether this chosen channel is acceptable. If it is the message transfer takes place, if not an exception is thrown.

The sample attribute rule *examples.rules.AttributeRule* has the following characteristics:

(a) If the queue has an authenticator, the channel must have the same authenticator. If the queue does not have an authenticator, it does not matter whether the channel has one or not. If the channel has been authenticated it cannot be upgraded, but if it does not have one, an authenticator can be added to a channel.

(b) If the queue has a cryptor, the channel must have a cryptor that is the same as or better than that on the queue. If the queue does not have a cryptor it does not matter whether the channel has one or not. A cryptor can be added to a channel or strengthened. A cryptor cannot be removed from the channel or replaced with a weaker cryptor.

Better is defined as:

i. Any cryptor is the same as or better than XOR

ii. Any cryptor, except XOR, is the same as or better then DES

iii. The remaining cryptors (triple DES, RC4, RC6, and MARS) are considered equal to each other and all are better than XOR and DES.

(c) It does not matter what compressors are defined for the queue or channel. A compressor can be changed, added to, or removed from the channel.

It can be seen from the above description that it is not necessary that the remote queue definition security attributes match the target queue attributes, since negotiation takes place. This ability for mismatches to be tolerated can be exploited, for example, if it is desired that the messages flow from the source queue manager to the target queue manager over secure channels – even though the target queue is unprotected. This can be achieved by setting security attributes on the remote queue definition; even they do not exist on the target queue.

7.5 Transporters

Transporters convey messages to target queues, using the services of a channel. Thus channels are addressed to queue managers, whilst transporters are addressed to queues at queue managers. MQe automatically handles transporter management; however the transporter class to be used can be specified as a property of those queue types that move messages (i.e. remote queues, store and forward queues and home server queues) – for more details see *Queues* on page 102. The transporter can be regarded as doing puts and gets of messages on behalf of the sending (or receiving queue), ensuring that the conditions of once-only assured delivery are met; this is similar to the mechanisms used by an application to get and receive messages from a queue.

MQe offers a single transporter *com.ibm.mqe.MQeTransporter* as standard, which implements once only, assured delivery over MQe channels. It does allow certain aspects of its behavior to be set; again these parameters are presented as queue properties:

XOR compress:
: Indicates whether the transporter is to implement XOR compression of the field data with previous field data (this process also assists any compressor that may have been defined).

Close if idle:
: Indicates if the transporter is to be closed after transmitting all available messages.

In principle, other transporters could be developed which might, for example, offer increased performance at the expense of delivery assurance.

8 Communications adapters

Channels use communications adapters to provide access to the underlying communications infrastructure (e.g. TCP/IP, HTTP, UDP etc). Multiple communications adapters can exist for any particular underlying protocol support with different characteristics – they may differ, for example, in their performance, efficiency or footprint. MQe provides a number of communications adapters that may be specified in connection and listener definitions. In summary, the following are available:

8.1 TCP/IP adapters

com.ibm.mqe.adapters.MQeTcpipHistoryAdapter

This adapter implements an efficient protocol over TCP/IP. By default, this is the preferred adapter for TCP/IP, offering better performance than either *com.ibm.mqe.adapters.MQeTcpipLengthAdapter* or *com.ibm.mqe.adapters.MQeTcpipHTTPAdapter*. It takes options that control its operation:

<HISTORY> ensures that recently used data is cached to avoid re-transmission.

<NOHISTORY> ensures that recently used data is not cached.

<NOPERSIST> ensures that the connection does not beyond a single data transfers.

<PERSIST> ensures that the connection survives data transfers and is re-used.

<HISTORY><PERSIST> are the defaults if no options are specified.

Adapter options are specified in connection definitions (see *Communications adapters* on page 67).

com.ibm.mqe.adapters.MQeTcpipLengthAdapter

This adapter implements a simple, byte-efficient protocol over TCP/IP. The connection is not maintained between data transfers. For almost all purposes where TCP/IP is required, excepting where minimal storage is an important consideration, the *com.ibm.mqe.adapters.MQeTcpipHistoryAdapter* should be used in preference.

8.2 HTTP adapters

com.ibm.mqe.adapters.MQeTcpipHttpAdapter

This adapter implements communications over HTTP using the HTTP 1.0 protocol. Since HTTP adds additional overhead, where there is a choice of underlying protocol the *com.ibm.mqe.adapters.MQeTcpipHistoryAdapter* should be used in preference.

com.ibm.mqe.adapters.MQeWESAuthenticationAdapter

This adapter provides support for tunneling HTTP requests through WebSphere Everyplace Authentication and transparent proxies.

8.3 UDP adapters

com.ibm.mqe.adapters.MQeUdpipAdapter

This adapter provides support for assured data transfer using UDP/IP datagrams. It uses Java properties to set the values that control its behavior. The property names are:

MQeUdpipAdapter.Packet.Size
MQeUdpipAdapter.Timeout
MQeUdpipAdapter.Connection.Timeout
MQeUdpipAdapter.Retry.Interval

If the Java properties are not set by the user application then the default values are used. These are:

DefaultTimeOut	= -1
DefaultPacketSize	= 532
DefaultConnectionTimeout	= 10000
DefaultRetryInterval	= 250

These properties should be changed with care. However, the networks over which UDP is an appropriate protocol (such as mobile and satellite) vary greatly in operating characteristics (such as latency, error rate and bandwidth) and the properties should be set accordingly.

The *default timeout* of -1 means that no time out is set, which is appropriate under most circumstances.

The *packet size* should be altered to the optimum packet size for the network being used, minus 60 bytes for the IP header. The default packet size is set for a mobile network.

The *connection timeout* is divided by the *retry interval* to determine the number of times that the connection is tested before assuming the remote end is not going to respond to a request.

The *retry interval* is used variously to set the socket timeout and to set the retry limit with the connection timeout value.

These properties may be set by the command line parameter to the JVM using the -D option, for instance -DMQeUdpipAdapter.Timeout=500 or within a program using the *System.setProperties()* method, for instance:

System.setProperties("MQeUdpipAdapter.Timeout", "500").

9 Connections

A connection provides its local queue manager with all the information needed to establish communications with a remote queue manager. The name of a connection is the name of that remote queue manager. Detailed information, where appropriate, in the definition identifies the channel type and the communications adapter to be used.

Only one connection definition can exist on an individual local queue manager for any particular remote queue manager. MQe supports different types of connections; the two most common being:

- *Direct*

  A direct connection supplies the information needed for the local queue manager to create channels directly to another queue manager elsewhere in the MQe network. The connection name matches the name of that remote queue manager. Direct indicates that the channels go to the remote queue manager without passing through an intermediate queue manager.

- *Indirect*

  An indirect connection (or *via* connection) indicates that messages destined for a remote queue manager elsewhere in the MQe network are to be sent to an intermediate queue manager (for which a direct connection will also exist). The connection name matches the name of the remote queue manager.

MQe has two variants on the direct connection:

- *Alias-only*

  An alias-only connection is a specialized type of direct connection that is used to give alias names to remote queue managers that feature as destinations in store and forward queue definitions. No other parameters are required because the store and forward queue itself handles any associated delivery aspects.

- *MQ*

  This is a specialized type of direct connection that identifies a remote queue manager as an MQSeries queue manager, as opposed to an MQe queue manager. It can also assign aliases to the name of the MQ queue manager.

MQe also has the concept of a local connection, of which two variants exist:

- *Local (simple version)*

  This is a specialized type of connection that is actually created as a queue manager connection to itself, i.e. the name of the connection is the same as the name of the local queue manager. Local connections are best regarded as system definitions used by MQe to store local queue manager aliases and/or a peer listener definition (see below). It should not be necessary to explicitly create local connections when using the MQe_Explorer management tool; they are automatically created as required. By the same token, when using that tool, local connections should not normally be deleted.

- *Peer listener*

  This is a specialized type of connection that is used to define a peer channel listener and is simply an extended variant of the local connection above. For local queue managers created by the MQe_Explorer management tool it should not normally be necessary to define such a connection; MQe_Explorer will automatically create/delete one as required by the type setting of the local queue manager. The peer listener is described in the section *Client/server listeners* on page 82.

9.1 Direct connections

A direct connection definition typically has the following properties, as presented by the MQe_Explorer management tool:

Connection name:
: Identifies the name of the target queue manager.

Local qMgr:
: The name of the local queue manager owning the connection definition.

Aliases:
: Alias names are optional alternative names that are mapped by the local queue manager to this same connection name (see below).

Channel class:
: The channel class (or alias) used to realize the connection.

Description:
: An arbitrary string describing the connection.

Primary adapter class:
: The communication adapter class (or alias) that will be used to realize the connection. Adapters provide access to the underlying communications infrastructure (e.g. TCP/IP, HTTP, UDP etc). The various adapters are briefly described in *Communications adapters* on page 67.

Primary adapter encoded parameters:
: Adapter encoded parameters are a byte string passed to the adapter in order to realize the connection. Not all adapters take encoded parameters.

Primary adapter IP address:
: The IP address of the target queue manager.

Primary adapter IP port:
: The port that the target queue manager is listening on for an incoming connection request.

Primary adapter options:
Adapter options are an ASCII string passed to the adapter in order to control its operation (e.g. keep a socket open between data transfer requests). Not all adapters take options.

Primary adapter parameters:
Adapter parameters are an ASCII string passed to the adapter in order to realize the connection. For example, for an HTTP connection, the adapter parameter would be the name of the servlet. Not all adapters take parameters.

Secondary adapter properties:
Secondary adapters are not supported by current versions of MQe.

The essence of a direct connection definition is that it provides the network addressing information required to contact the remote queue manager. It also instructs MQe on the channel class to be used with that connection, and the adapter class. For every direct connection it is also necessary to ensure that the target queue manager has a listener configured in a totally compatible manner. Any mismatch is likely to mean that MQe cannot setup the channels between the source and target queue managers, and consequently that messages cannot be transferred.

MQe uses the connection definition at the channel level – not at the queue level. Consequently, the connection definition does not determine whether messages are sent synchronously or asynchronously. Control of this aspect is determined by queue definitions, and is discussed first in the chapter *Queues* on page 102 and then in subsequent chapters.

The property list above was carefully described as being that view presented by MQe_Explorer. At the MQe level the situation is subtly different; three of the above properties are combined into a composite property, as shown below:

Primary adapter file descriptor:

<primary adpt. class> **:** <primary adpt. IP address> **:** <primary adpt. IP port>

The chapter *Configuration using admin messages* on page 40 provides information on the use of admin messages to configure connections.

9.2 Indirect connections

Indirect connection definitions are another way to provide MQe with the information needed to contact a remote queue manager. They instruct MQe to go via another queue manager and thus enforce an indirect routing on the message flow.

An indirect connection definition is simple because MQe gets the detailed information it needs from the via queue manager connection definition.

An indirect connection definition has the following properties:

Connection name:
Identifies the name of the target queue manager.

Local qMgr:
The name of the local queue manager owning the connection definition.

Aliases:
Alias names are optional alternative names that are mapped by the local queue manager to this same connection name (see below).

Description:
An arbitrary string describing the connection.

Via queue manager:
The name of the queue manager through which the channel should be routed.

Indirect (or via connections) are a vital tool in messaging routing, along with other MQe features, such as store and forward queues and home server queues. The routing aspects are discussed in *Multi-hop and* advanced messaging on page 128.

9.3 Local connections

Local connections are not concerned with the setup of channels to remote queue managers. The simple form of a local connection is a means to give alias names to a local queue manager. Although it might seem more appropriate to have queue manager alias names as a property of the queue manager, MQe introduces the concept of a local connection, where the name of the connection matches that of the local queue manager. All connection definitions can have aliases; in this case the aliases become aliases for the local queue manager itself.

MQe_Explorer presents a more logical view of this situation and manages these aliases as queue manager properties. It creates and manages local connections as required.

A local connection has the following properties[20]:

Connection name:
The name of the local queue manager.

Local qMgr:
The name of the local queue manager.

Aliases:
Alias names are optional alternative names for the local queue manager (see below).

Description:
An arbitrary string describing the connection.

A more complex form of the local connection is described in *Client/server listeners* on page 82.

[20] For the reasons stated, MQe_Explorer does not present all these properties for a local connection.

9.4 Alias-only/MQ connections

Alias-only/MQ connections exist to inform MQe of the presence of a connection, even though it will not be necessary for MQe to use the associated data to establish connectivity.

They have the following properties:

Connection name:
The name of the target queue manager.

Local qMgr:
The name of the local queue manager.

Aliases:
Alias names are optional alternative names for the target queue manager (see below).

Description:
An arbitrary string describing the connection.

9.5 Connection alias names

Connection definitions are used when MQe needs to deliver a message to a remote queue manager. The queue manager destination in the message requires MQe to create a (or use an existing) channel to that destination. In some cases however a degree of independence is needed between the queue manager addresses used by applications, and the actual names of queue managers in the network. The simplest example is where queue managers have been renamed after the application has been developed. This flexibility is provided through aliases.

Connections have an *aliases* property, which allows zero, one or more alias names to be associated with the connection. These aliases are not just alternative names that map to the same connection; they also change the destination queue manager name in the message. This latter property is important, otherwise the message would be put on a channel destined for the target queue manager, but would be rejected when it arrived because of the name mismatch.

Consequently if a message is addressed to a queue manager name that is defined as an alias of a connection name, then MQe changes the queue manager address in the message to become the connection name.

Connection aliases also play a crucial role in allowing alternative routes to be defined through an MQe network.

9.6 Configuration

Configuration of connection definitions requires the use of admin messages. These messages can either be issued programmatically, or through the use of the MQe_Explorer management tool. This chapter describes the creation and/or modification of the most common connection definitions, first using MQe_Explorer, and then through programming. The programming description builds upon the principles and examples presented in the chapter *Configuration using admin messages* on page 40.

Using MQe_Explorer

Remote connections

Using the MQe_Explorer, a new connection to a remote queue manager is made through the *New connection* dialogue. The type can be set to any one of these relevant values:

- Direct connection
- Indirect connection
- Alias-only/MQ connection

For example, the two tabs below are used to enter all the details necessary for the *myClientQMgr* to connect to its server (here called *myServerQMgr*). The first tab collects the adapter-independent data:

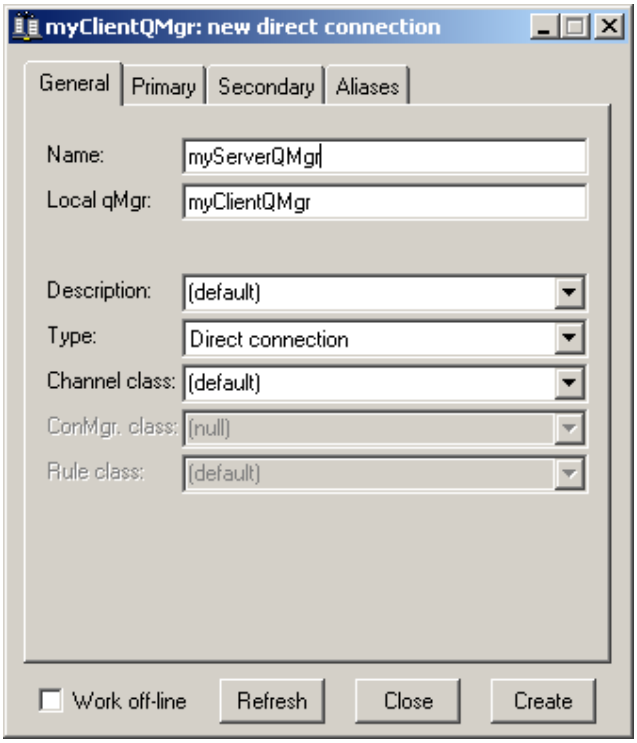

Figure 9-1: Creating a new direct connection - General properties

The default channel class for a direct connection is *com.ibm.mqe.MQeChannel*, i.e. a client/server channel.

The second tab collects the primary adapter-specific data:

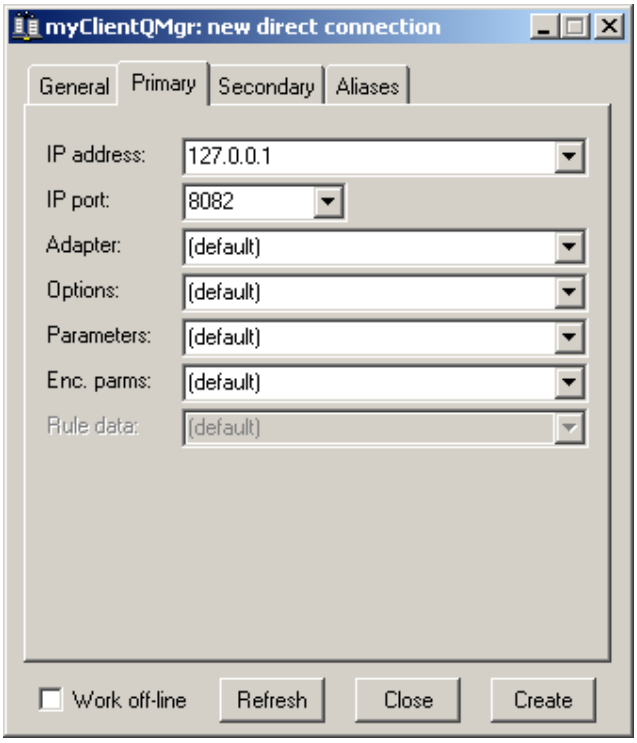

Figure 9-2: Creating a new direct connection - Primary adapter

The IP address is the loop-back address and thus assumes that *myServerQMgr* is running on the local machine; port 8082 has been identified as the port on which it is listening for incoming client/server connection requests. The default primary adapter class for a direct connection is *com.ibm.mqe.adapters.MQeTcpipHistoryAdapter*, i.e. the most efficient MQe exploitation of the TCP/IP protocol. The default primary adapter options for this adapter are *<PERSIST><HISTORY>,* i.e. to use a persistent connection and to record history to minimize data transmission. The default channel class is that for a client/server channel. The default parameters are "", assuming for example, that a servlet is not present. By default, no encoded parameter data is supplied.

Once a connection has been created it can be modified through the *Modify connection* dialogue – in this way any parameters can be changed, including the *Type* of the connection. Alternatively (but less efficiently) the connection can be deleted and a new one then created. A disadvantage of this delete/new approach is that any remote queue definitions that have been created for the original connection will be lost on its deletion.

Be aware that when any primary adapter property is changed, MQe requires that all other primary adapter properties be re-supplied and are reset to the new values. MQe_Explorer will handle this automatically as far as is possible when the administered queue manager is on-line; if off-line however, all the adapter property values must be explicitly set.

The following example illustrates how an indirect connection can be created. In this case it is assumed that a queue manager *myOtherQMgr* is reachable from *myClientQMgr* through *myServerQMgr*. The relevant MQe_Explorer property tab values are as follows. The first tab is almost as previously:

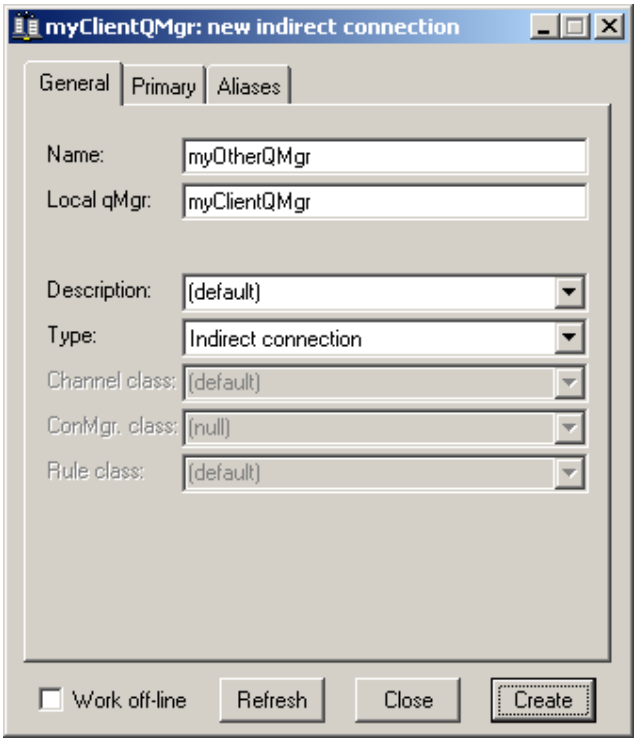

Figure 9-3: Creating a new indirect connection - General properties

The second tab has minimal, but vital, information:

Figure 9-4: Creating a new indirect connection - Primary adapter

Local connections

MQe_Explorer handles many aspects of local connection configuration automatically.

For all queue managers:

- A local connection is automatically created and/or updated, if needed to store queue manager aliases.

Additionally, only for the queue manager hosting MQe_Explorer:

- A peer listener form of the local connection may be created/updated/revoked depending upon the type setting of the queue manager:
  - true – if the queue manager is either a *peer* or *client.*
  - false – if the queue manager is either a *server* or *gateway.*

Otherwise the *New connection* dialogue can be used to create either a simple local connection or a peer listener; the *Modify connection* dialogue must be used to change between these forms, through setting the *Type* property.

Remote connections

The first example below shows how to programmatically create the remote direct connection shown above as an MQe_Explorer example. The properties are:

***Connection name*:**	**myServerQMgr**
***Description*:**	**Direct connection to myServerQMgr**
***Type*:**	**Direct**
***Channel class*:**	**com.ibm.mqe.MQeChannel**
***Primary IP address*:**	**127.0.0.1**
***Primary IP port*:**	**8082**
***Primary adapter*:**	**com.ibm.mqe.adapters.MQeTcpipHistoryAdapter**
***Primary adapter options*:**	**<PERSIST><HISTORY>**
Primary adapter parms:	***(none)***

A local queue manager is used, with admin replies returned to its *AdminReplyQ*.

```
//get addressability to the local queue manager
MQeQueueManager qMgr = MQeQueueManager.getReference(null);

//set up strings
String thisQMgrName = qMgr.getName();                              //local qMgr name
String targetQMgrName = "myClientQMgr";                            //target qMgr name
String connName = "myServerQMgr";                                  //connection name
String descrText = "Direct connection to myServerQMgr";            //description
String adapterText = "com.ibm.mqe.adapters.MQeTcpipHistoryAdapter"; //adapter class
String portText = "8082";                                          //IP port
String addressText = "127.0.0.1";                                  //IP address
String channelText = "com.ibm.mqe.MQeChannel";                     //channel class
String optionsText = "<PERSIST><HISTORY>";                         //adapter options
String parmsText = "";                                             //adapter parameters

//create the basic connection admin message
MQeConnectionAdminMsg adminMsg = new MQeConnectionAdminMsg();

//add command-neutral elements
adminMsg.putAscii(MQeAdminMsg.Admin_TargetQMgr, targetQMgrName);//set target qMgr
adminMsg.putInt(MQeAdminMsg.Admin_MaxAttempts, 1);                 //set max tries
adminMsg.putInt(MQe.Msg_Style, MQe.Msg_Style_Request);             //need a reply
adminMsg.putAscii(MQe.Msg_ReplyToQMgr, thisQMgrName);              //reply queue manager
adminMsg.putAscii(MQe.Msg_ReplyToQ, MQe.Admin_Reply_Queue_Name);//reply queue

//add unique reference
byte[] adminKey = MQe.longToByte(System.currentTimeMillis());     //generate unique key
adminMsg.putArrayOfByte(MQe.Msg_CorrelID, adminKey);               //add key

//add create command
adminMsg.putInt(MQeAdminMsg.Admin_Action, MQeAdminMsg.Action_Create);

//add command parameters
MQeFields parms = new MQeFields();                                 //new parms object
parms.putAscii(MQeAdminMsg.Admin_Name, connName);                  //connection name
parms.putAscii(MQeAdminMsg.Admin_Class,
        "com.ibm.mqe.MQeConnectionDefinition");                    //object class
```

```
parms.putUnicode(MQeConnectionAdminMsg.Con_Description,
        descrText);                                          //description
parms.putAscii(MQeConnectionAdminMsg.Con_Channel,
        channelText);                                        //channel class
MQeFields[] adapterArray = new MQeFields[1];                 //adapter array
adapterArray[0] = new MQeFields();                           //primary member
adapterArray[0].putAscii(MQeConnectionAdminMsg.Con_Adapter,
        adapterText + ":" + addressText + ":" + portText);   //file descriptor
adapterArray[0].putAscii(MQeConnectionAdminMsg.Con_AdapterOptions,
        optionsText);                                        //options
adapterArray[0].putAscii(MQeConnectionAdminMsg.Con_AdapterAsciiParm,
        parmsText);                                          //parameters
parms.putFieldsArray(MQeConnectionAdminMsg.Con_Adapters,
        adapterArray);                                       //adapter array
adminMsg.putFields(MQeAdminMsg.Admin_Parms, parms);          //add parms

//send admin message
qMgr.putMessage(targetQMgrName, MQe.Admin_Queue_Name, adminMsg , null, 0);
```

Example 9-1: Creating a new direct connection

This example follows on naturally from the cases described in *Admin programming* on page 47. The principal points of interest are:

1. The file descriptor parameter needed for the admin message is a composite string.
2. Adapter details are specified in an array of MQeFields objects.

Executing the sample – *Create a direct connection*:

This depends upon the existence of the configured *myClientQMgr* queue manager created earlier.

Then use the command: *MQeConfigGuide 5*. The sample code is contained in the method *createDirectConnection().* By default, the MQe_Explorer-created *myClientQMgr.ini* file is used; by editing the sample code, the method *createClientEnvironment()* can be used instead.

The sample cannot be re-run without first deleting the created direct connection – use MQe_Explorer to do this (or write code).

The second example here shows how to programmatically create the remote indirect connection shown above as an MQe_Explorer example. The properties are:

***Connection name*:**	**myOtherQMgr**
***Description*:**	**Indirect connection to myOtherQMgr via myServerQMgr**
***Type*:**	**Indirect**
***Channel class*:**	**com.ibm.mqe.MQeChannel**
***Via qMgr*:**	**myServerQMgr**

A local queue manager is used, with admin replies returned to its *AdminReplyQ*.

```
//get addressability to the local queue manager
MQeQueueManager qMgr = MQeQueueManager.getReference(null);

//set up strings
String thisQMgrName = qMgr.getName();                                   //local qMgr name
String targetQMgrName = "myClientQMgr";                                 //target qMgr name
String connName = "myOtherQMgr";                                        //connection name
String viaName = "myServerQMgr";                                        //via qMgr name
String channelText = "com.ibm.mqe.MQeChannel";                          //channel class
String descrText =
        "Indirect connection to myOtherQmgr via myServerQM";            //description

//create the basic connection admin message
MQeConnectionAdminMsg adminMsg = new MQeConnectionAdminMsg();

//add command-neutral elements
adminMsg.putAscii(MQeAdminMsg.Admin_TargetQMgr, targetQMgrName);//set target qMgr
adminMsg.putInt(MQeAdminMsg.Admin_MaxAttempts, 1);                      //set max tries
adminMsg.putInt(MQe.Msg_Style, MQe.Msg_Style_Request);                  //need a reply
adminMsg.putAscii(MQe.Msg_ReplyToQMgr, thisQMgrName);                   //reply queue manager
adminMsg.putAscii(MQe.Msg_ReplyToQ, MQe.Admin_Reply_Queue_Name);//reply queue

//add unique reference
byte[] adminKey = MQe.longToByte(System.currentTimeMillis());           //generate unique key
adminMsg.putArrayOfByte(MQe.Msg_CorrelID, adminKey);                    //add key

//add create command
adminMsg.putInt(MQeAdminMsg.Admin_Action, MQeAdminMsg.Action_Create);
```

```
//add command parameters
MQeFields parms = new MQeFields();                          //new parms object
parms.putAscii(MQeAdminMsg.Admin_Name, connName);           //connection name
parms.putAscii(MQeAdminMsg.Admin_Class,
        "com.ibm.mqe.MQeConnectionDefinition");             //object class
parms.putUnicode(MQeConnectionAdminMsg.Con_Description,
        descrText);                                         //description
parms.putAscii(MQeConnectionAdminMsg.Con_Channel,
        channelText);                                       //channel class
MQeFields[] adapterArray = new MQeFields[1];                //adapter array
adapterArray[0] = new MQeFields();                          //primary member
adapterArray[0].putAscii(MQeConnectionAdminMsg.Con_Adapter,
        viaName);                                           //file descriptor
parms.putFieldsArray(MQeConnectionAdminMsg.Con_Adapters,
        adapterArray);                                      //adapter array
adminMsg.putFields(MQeAdminMsg.Admin_Parms, parms);         //add parms

//send admin message
qMgr.putMessage(targetQMgrName, MQe.Admin_Queue_Name, adminMsg , null, 0);
```

Example 9-2: Creating a new indirect connection

This example follows on naturally from the cases described in *Admin programming* on page 47. The only additional point of interest is:

1. The file descriptor parameter needed for the admin message is the name of the via queue manager.

Executing the sample – *Create a direct connection*:

This depends upon the existence of both the configured *myClientQMgr* queue manager created earlier and of the connection to *myServerQMgr*.

Then use the command: *MQeConfigGuide 6*. The sample code is contained in the method *createIndirectConnection()*. By default, the MQe_Explorer-created *myClientQMgr.ini* file is used; by editing the sample code, the method *createClientEnvironment()* can be used instead.

The sample cannot be re-run without first deleting the created indirect connection – use MQe_Explorer to do this (or write code).

Local connections

The use of admin messages to create peer listeners is shown in the section *Using admin messages* on page 86.

10 Listeners

In the chapter *The basics* on page 4, four types of queue manager were defined: *client*, *peer*, *server* and *gateway*. The client queue manager has already been described in some detail in the chapter *Client queue managers* on page 8 and its fundamental characteristic is that it can only make (outgoing) connection requests; it cannot accept them. The essence of the other queue manager types is that they can all handle (incoming) connection requests. The peer queue manager handles them from other peers, the server (and the gateway) from client queue managers. Receiving such incoming requests requires the queue manager to have an active *listener*; such an active listener is protocol-specific and it listens for requests on a particular port. Making an outgoing connection request requires the initiating queue manager to have an appropriate *connection* definition.

MQe defines two classes of listener:

- **Client/server** – listens for incoming requests from a client queue manager, subsequently moving data over a client/server channel.
- **Peer** – listens for incoming requests from a peer queue manager, subsequently moving data over a peer channel.

The types of listeners are thus directly linked to the MQe channels types. For more information on channel types, see the chapter *Channels* on page 63.

10.1 Client/server listeners

MQe provides a client/server channel listener that listens on a single port for multiple incoming connection requests from clients. Each client request is then handed off to another port for the socket to be created; thus one port (with a fixed port number) can be continuously available to service incoming requests.

The client/server listener is not manageable object, i.e. it cannot be created, modified or deleted through admin messages; nor are its details stored in the MQe registry. The client/server listener must be created as part of the process of starting a queue manager.

Extending the earlier example given in *Running a client queue manager* on page 14, where a client queue manager was started, the additional code required is:

```
//create the queue manager environment
....                                                    //use MQeFields environment

//create the client/server channel manager
MQeChannelManager localChannelManager = new MQeChannelManager();

//set the maximum number of concurrent channels
localChannelManager.numberOfChannels(0);                //no limit

//start the queue manager
MQeQueueManager qMgr =
        MQeQueueManagerUtils.processQueueManager(environment,
        localChannelManager.getGlobalHashtable( ));

//create a client/server channel listener
String listenerAdapterText = "com.ibm.mqe.adapters.MQeTcpipHistoryAdapter";
String spawnAdapterText = "com.ibm.mqe.adapters.MQeTcpipHistoryAdapter";
String listenerPort = "8082";
MQeChannelListener localChannelListener = newMQeChannelListener(
        listenerAdapterText + "::" + listenerPort ,
        spawnAdapterText,
        localChannelManager));

//set the channel timeout interval for the listener
int timeInterval = 300;
localChannelListener.setTimer(timeInterval);
```

Example 10-1: Setting up a client/server listener

A client/server listener first requires the queue manager to have a channel manager object present; this gives the queue manager the ability to manager multiple concurrent channels. The channel manager is created before the queue manager is started, and can be configured to restrict the maximum number of concurrent channels.

In the code extract above it is assumed that an *MQeFields* variable *environment* contains the environmental parameters, exactly as shown previously for the *myClientQMgr* queue manager. Now however, starting the queue manager using the *processQueueManager()* method in the *examples.queuemanager.MQeQueueManagerUtils* class, requires that the channel manager's global hash table is passed to the queue manager. Previously for the client queue manager, a *null* was passed, indicating no channel manager.

After the queue manager is running, one or more client/server listeners can be created. For a listener, the adapter and port appropriate to the incoming connection request are specified. Additionally the adapter to be subsequently used for continuing communication must be identified; typically both adapters will be identical. The time interval specifies how long an idle channel is allowed to remain active before being destroyed.

From the code above it is easy to see how additional client/server listeners can be created, such that MQe can listen concurrently on multiple ports and/or protocols.

The MQe_Explorer management tool configures a client/server listener when a server or gateway queue manager is created or started; however v1.27 of the tool only supports one such listener. Creation of server queue managers is further described in the section *Server queue manager* on page 88.

10.2 Peer listeners

MQe provides a peer channel listener that listens on a single port for an incoming connection request from a master. It then uses that same port to create the sockets for the subsequent channel(s). The port is not free to service another incoming request whilst the channel(s) exist.

The peer channel listener is a manageable object, i.e. it can be created, modified or deleted through admin messages. Details are stored in the MQe registry. The only slightly unusual aspect of this support is that the client/server listener is managed as a local connection, i.e. a special case of a connection definition.

10.3 Configuration of peer listeners

Using MQe_Explorer

The MQe_Explorer management tool configures a peer channel listener when a peer queue manager is created or started. Clearly, this is restricted to local queue managers, i.e. the queue manager hosting MQe_Explorer. If a remote queue manager requires a new peer listener, the it is created through the *New connection* dialogue. The type of the connection is set to 'Peer listener'.

For example using the *myClientQMgr* queue manager created earlier, the two tabs below are used to enter all the details necessary for it to be configured with a peer listener. The first tab collects the adapter-independent data:

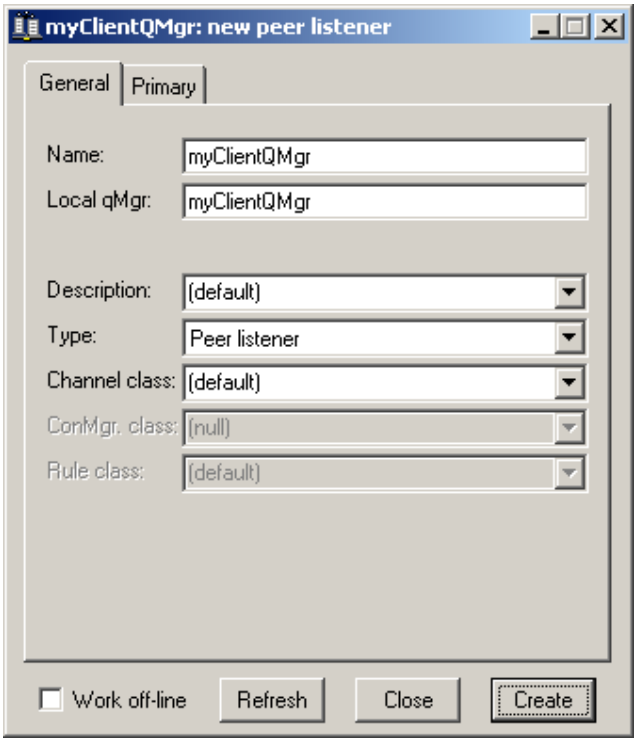

Figure 10-1: Creating a new peer listener - General properties

The default channel class for a peer listener is *com.ibm.mqe.MQePeerChannel*, i.e. a peer channel.

The second (primary) tab collects the communications-specific data:

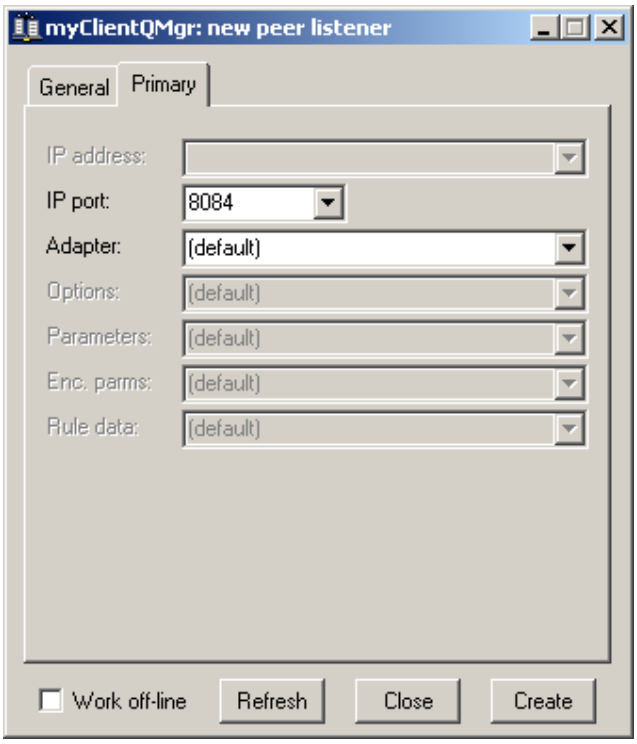

Figure 10-2: Creating a new peer listener - Primary tab

The port to be used is shown in the above example as '8084'. The default primary adapter class for a direct connection is *com.ibm.mqe.adapters.MQeTcpipHistoryAdapter*, i.e. the most efficient exploitation of the TCP/IP protocol. All other properties are disabled since they are not relevant.

The example shown above is somewhat contrived. We have used the *New connection* dialogue to create a peer listener for the local queue manager, when it is intended to be used for a remote queue manager[21]. MQe_Explorer explicitly handles the type property of local queue managers and here it has been set to *client*. When the queue manager is stopped and then restarted, MQe_Explorer will detect the presence of this peer channel listener and downgrade it to be just a simple local connection[22]. If it were allowed to remain, the queue manager would actually be a *peer*, yet it is supposed to be a *client*[23].

Once a peer listener has been created it can be modified via the *Modify connection* dialogue – in this way any parameters can be changed. Alternatively (but less efficiently) the connection can be deleted and a new one then created.

Since the creation of peer listeners is independent of client/server listeners, then it is obviously possible to create queue managers that listen for both client/server and peer incoming connection requests.

---

[21] We did this because so far, we have only created one queue manager, so there is no remote queue manager that we can use for this purpose.
[22] MQe_Explorer does not just delete it because the definition may also be being used to hold queue manager aliases – these would be lost if the definition is deleted.
[23] If the local queue manager were defined to be either a *server* or a *gateway*, then MQe_Explorer would not downgrade the peer listener definition to a simple local connection.

Using admin messages

The example below shows how to programmatically create the peer listener shown above as an MQe_Explorer example. The properties are:

***Description*:**	**Peer listener on port 8084**
***Adapter*:**	**com.ibm.mqe.adapters.MQeTcpipHistoryAdapter**
***IP port*:**	**8084**

A local queue manager is used, with admin replies returned to its *AdminReplyQ*:

```
//get addressability to the local queue manager
MQeQueueManager qMgr = MQeQueueManager.getReference(null);

//set up strings
String thisQMgrName = qMgr.getName();                                  //local qMgr name
String targetQMgrName = "myClientQMgr";                                //target qMgr name
String descrText = "Peer listener on port 8084";                       //description
String adapterText = "com.ibm.mqe.adapters.MQeTcpipHistoryAdapter";    //adapter class
String portText = "8084";                                              //IP port
String channelText = "com.ibm.mqe.MQePeerChannel";                     //channel class

//create the basic connection admin message
MQeConnectionAdminMsg adminMsg = new MQeConnectionAdminMsg();

//add command-neutral elements
adminMsg.putAscii(MQeAdminMsg.Admin_TargetQMgr, targetQMgrName);//set target qMgr
adminMsg.putInt(MQeAdminMsg.Admin_MaxAttempts, 1);                     //set max tries
adminMsg.putInt(MQe.Msg_Style, MQe.Msg_Style_Request);                 //need a reply
adminMsg.putAscii(MQe.Msg_ReplyToQMgr, thisQMgrName);                  //reply queue manager
adminMsg.putAscii(MQe.Msg_ReplyToQ, MQe.Admin_Reply_Queue_Name);//reply queue

//add unique reference
byte[] adminKey = MQe.longToByte(System.currentTimeMillis());          //generate unique key
adminMsg.putArrayOfByte(MQe.Msg_CorrelID, adminKey);                   //add key

//add create command
adminMsg.putInt(MQeAdminMsg.Admin_Action, MQeAdminMsg.Action_Create);
```

```
//add command parameters
MQeFields parms = new MQeFields();                          //new parms object
parms.putAscii(MQeAdminMsg.Admin_Name, targetQMgrName );    //connection name
parms.putAscii(MQeAdminMsg.Admin_Class,
        "com.ibm.mqe.MQeConnectionDefinition");             //object class
parms.putUnicode(MQeConnectionAdminMsg.Con_Description,
        descrText);                                         //description
parms.putAscii(MQeConnectionAdminMsg.Con_Channel,
        channelText);                                       //channel class
MQeFields[] adapterArray = new MQeFields[1];                //adapter array
adapterArray[0] = new MQeFields();                          //primary member
adapterArray[0].putAscii(MQeConnectionAdminMsg.Con_Adapter,
        adapterText + "::" + portText);                     //file descriptor
parms.putFieldsArray(MQeConnectionAdminMsg.Con_Adapters,
        adapterArray);                                      //adapter array
adminMsg.putFields(MQeAdminMsg.Admin_Parms, parms);         //add parms

//send admin message
qMgr.putMessage(targetQMgrName, MQe.Admin_Queue_Name, adminMsg , null, 0);
```

Example 10-2: Creating a new peer listener

This example follows on naturally from the cases described in *Admin programming* on page 47. The principal points of interest are:

1. The file descriptor parameter needed for the admin message is a composite string combining the adapter and the port number.

Executing the sample – *Create a peer listener*:

This depends upon the existence of the configured *myClientQMgr* queue manager created earlier.

Then use the command: *MQeConfigGuide 7*. The sample code is contained in the method *createPeerListener()*. By default, the MQe_Explorer-created *myClientQMgr.ini* file is used; by editing the sample code, the method *createClientEnvironment()* can be used instead.

The effect of running this sample is to change the client queue manager into a peer. The sample cannot be re-run without first deleting the created peer listener – use MQe_Explorer to do this (or write code). If using MQe_Explorer, note that when it starts the queue manager it will downgrade the peer listener to a simple connection, because the queue manager is defined to be a client. This simple connection must itself be deleted.

11 Server, gateway and peer queue managers

11.1 Server queue managers

The chapter *Client queue managers* on page 8 presented the information necessary to create and start client queue managers, either indirectly through MQe_Explorer or directly in code. Following that, channels, connections and listeners have been discussed. Together, all the information required to both create and start server queue managers has been described. In this section we will pull together just those elements that relate to server queue managers.

We have seen that a server queue manager can be regarded as just a client queue manager – but one that also has the additional capability of accepting incoming client/server channel connection requests, i.e. it has an active client/server listener. Moreover, it has been seen that the client/server listener needs the services of a channel manager. The configuration aspects of these various server-related elements are not stored in the registry, but are created and configured as the queue manager itself is started. So, a queue manager can, in principle, be started as a client, subsequently re-started as a server, and indeed re-started again as a client. In practice, such flexibility of behavior is unlikely to be needed.

Rather than start the client queue manager *myClientQMgr* that we have already created, as a server, it is useful now to create a second queue manager *myServerQMgr* that we can treat as a server – and indeed be capable of acting as an MQe server to *myClientQMgr*.

Using MQe_Explorer

Run MQe_Explorer and create a new server queue manager as below. On the *General* tab set the input fields as shown:

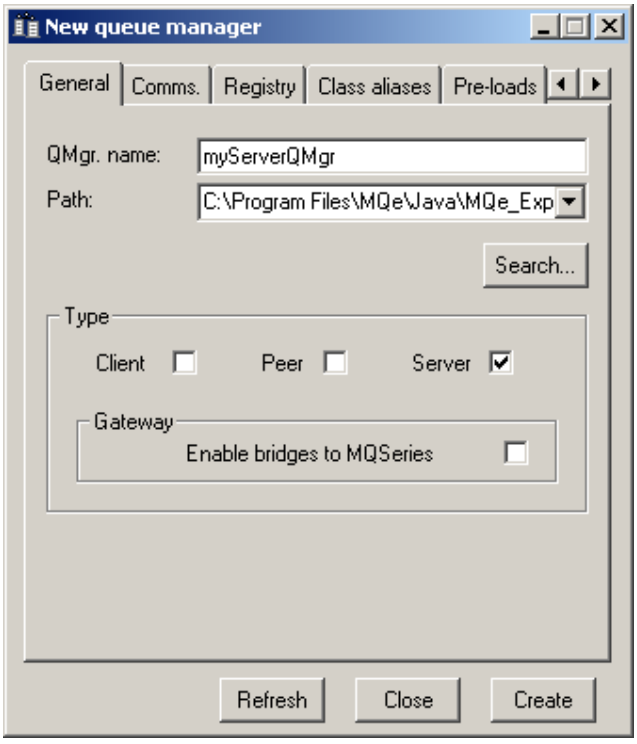

Figure 11-1: Creating a server queue manager – General tab

On the *Comms*. tab, set the client/server listener properties as shown:

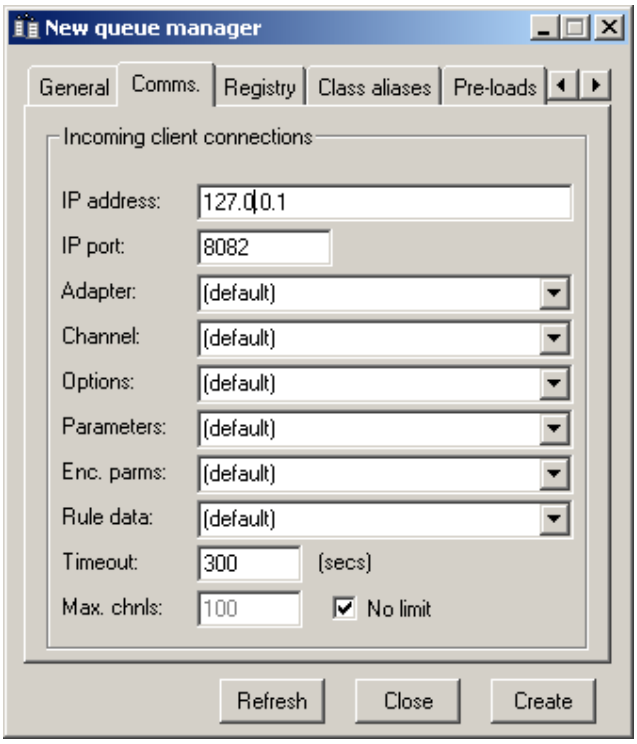

Figure 11-2: Creating a server queue manager – Comms. tab

This creates a server queue manager *myServerQMgr* that listens for incoming connections on the TCP/IP port 8082 and uses the *com.ibm.mqe.adapters.MQeTcpipHistoryAdapter* adapter. These choices intentionally match the properties used in the *myServerQMgr* connection definition held by *myClientQMgr* and created earlier in the connection section *Configuration* on page 73.

The associated initialization file has the contents:

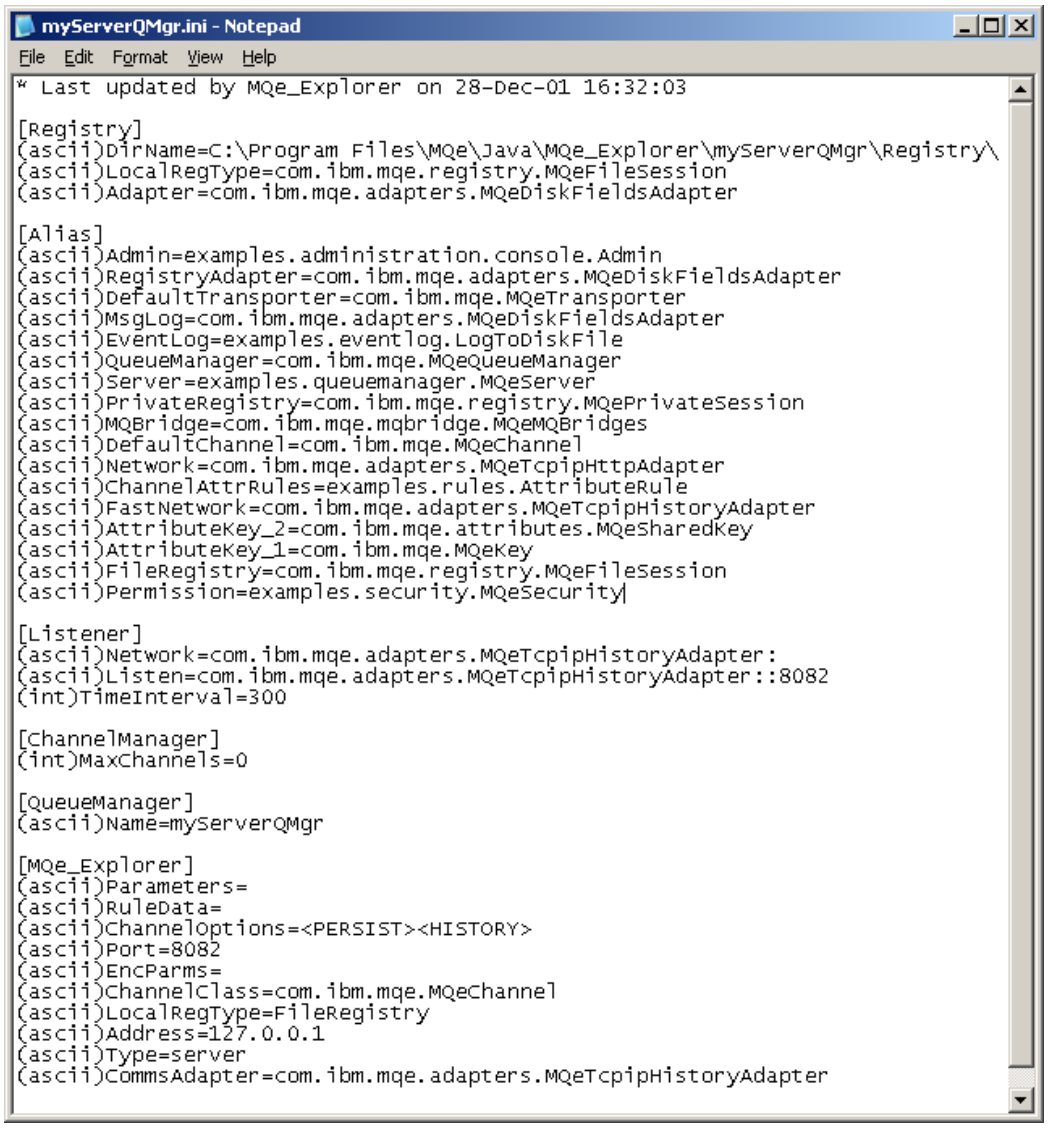

Figure 11-3: The myServerQMgr initialization data

It is interesting to compare this data with that in *Figure 3-7: The myClientQMgr initialization data* on page 14. The additional contents here are:

5. A *[Listener]* section containing configuration information for the client/server channel listener.
6. A *[ChannelManager]* section containing configuration information for the channel manager.
7. Additional and changed content in the *[MQe_Explorer]* section.

This allows MQe_Explorer to create and configure the client/server channel listener and channel manager, whilst starting the queue manager.

Using code

It follows from what we have seen earlier that the information in the section *Creating a client queue manager* on page 17 is equally applicable to creating a server queue manager – there being no difference. However the code shown both in *Setting up a client environment* on page 14 and in *Starting a client queue manager* on page 15, must be extended to cover the needs of a server.

Using the methods in the *examples.queuemanager.MQeQueueManagerUtils* class as before, but now processing a server initialization file – such as that just created by MQe_Explorer for *myServerQMgr* – the code is modified to become:

```
//read in the initialization file
MQeFields environment = MQeQueueManagerUtils.loadConfigFile(filename);

//process class aliases – [Alias] stanza
MQeQueueManagerUtils.processAlias(environment);

//process pre-loads – [PreLoad] stanza
MQeQueueManagerUtils.processPreLoad(environment);

//process permissions – [Permission] stanza
MQeQueueManagerUtils.processPermission(environment);

//process channel manager – [ChannelManager] stanza
MQeChannelManager localChannelManager =
        MQeQueueManagerUtils.processChannelManager(environment);
```

Example 11-1: Setting up a server queue manager environment

Thus a channel manager is created from the *[ChannelManager]* stanza; an alternative – and more direct way of doing this – was shown in the *Example 10-1: Setting up a client/server listener* on page 83.

Once the environment is established, the queue manager can be started. Again, using the methods in the *examples.queuemanager.MQeQueueManagerUtils* class:

```
//start the queue manager
MQeQueueManager qMgr =
        MQeQueueManagerUtils.processQueueManager(environment,
        localChannelManager.getGlobalHashtable( ));

//process the client/server channel listener
MQeChannelListener localChannelListener =
        MQeQueueManagerUtils.processListener(environment,
        localChannelManager);
```

Example 11-2: Starting a server queue manager

The differences here from the client queue manager case are:

1. Passing the channel manager global hash table for use when the queue manager is started.
2. Processing the *[Listener]* stanza to create the client/server channel listener.

An alternative – and more direct way of creating and configuring the client/server channel listener – was shown in the *Example 10-1: Setting up a client/server listener* on page 83.

A server queue manager can be stopped by using the various *close()* and *stop()* methods on the relevant objects:

```
//close the listener
localChannelListener.stop();

//close the queue manager
MQeQueueManager qMgr = MQeQueueManager.getReference(null);
 qMgr.close();
```

Example 11-3: Stopping a server queue manager

Executing the sample – *Running a server queue manager*:

Use the command: *MQeConfigGuide 8*. The sample code is contained in the methods *startServerQMgr()* and *stopServerQMgr()*. By default, the MQe_Explorer-created *myServerQMgr.ini* file is used; by editing the sample code, the method *createServerEnvironment()* can be used instead.

11.2 Gateway queue managers

A gateway queue manager is defined as a server queue manager – but one that also has the additional capability of exchanging messages with MQSeries queue managers. In other words, it is a server queue manager that is also has an active bridges object. The configuration aspect of the bridges object is not stored in the registry[24], but is created as the queue manager itself is started. So, a gateway queue manager can, in principle, be started as a server (or client) or as a gateway, depending upon the needs at the time. In practice, there are few occasions, if any, when this is appropriate.

Rather than start a queue manager such as *myClientQMgr* or *myServerQMgr* that we have already created, as a gateway, it is useful now to create a third queue manager *myGatewayQMgr* that is a gateway – and also capable of both acting as an MQe server to *myClientQMgr* and of connecting to *myServerQMgr*.

[24] Note that the children of a bridges object are stored in the MQe registry.

Using MQe_Explorer

Run MQe_Explorer and create a new gateway queue manager as below. On the *General* tab set the input fields as shown:

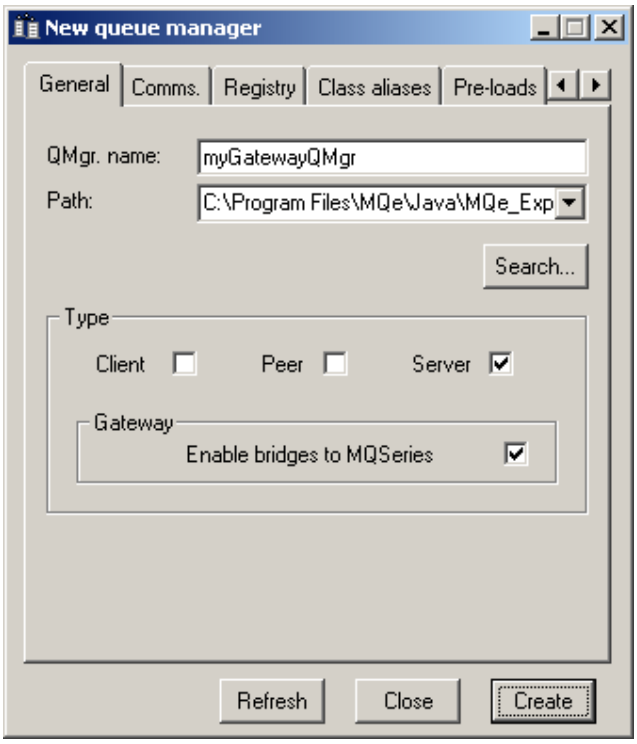

Figure 11-4: Creating a gateway queue manager – General tab

On the *Comms*. tab, set the client/server listener properties as shown:

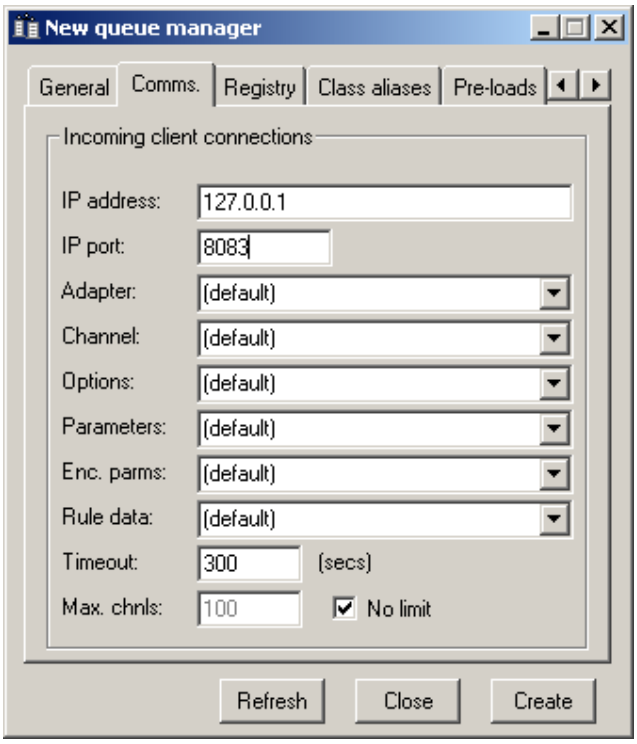

Figure 11-5: Creating a gateway queue manager – Comms. tab

This creates a gateway queue manager *myGatewayQMgr* that listens for incoming connections on the TCP/IP port 8083 and uses the *com.ibm.mqe.adapters.MQeTcpipHistoryAdapter* adapter. The port number does not conflict with that used by *myServerQMgr* to listener for incoming connection requests, created earlier in the section *Server queue managers* on page 88.

The associated initialization file has the contents:

Figure 11-6: The myGatewayQMgr initialization data

If this is compared with the file in *Figure 11-3: The myServerQMgr initialization data* on page 90, the differences are:

1. A *[MQBridge]* section containing configuration information for the bridges object and its children.
2. Changed content in the *[MQe_Explorer]* section.

This allows MQe_Explorer to create and configure the bridges object and its dependencies, whilst starting the queue manager itself.

Using code

It follows from what we have seen earlier that the information in the section *Creating a client queue manager* on page 17 is equally applicable to creating a gateway queue manager (as it was to a server). Likewise the code shown in *Example 11-1: Setting up a server queue manager environment* on page 91; the gateway requirements here are identical[25] to those of a server (although a superset of what was needed for a client). The *Example 11-2: Starting a server queue manager* on page 91 however, must be extended to cover the needs of a gateway.

The server code is modified to become:

```
//start the queue manager
MQeQueueManager qMgr =
        MQeQueueManagerUtils.processQueueManager(environment,
        localChannelManager.getGlobalHashtable( ));

//process the client/server channel listener
MQeChannelListener localChannelListener =
        MQeQueueManagerUtils.processListener(environment,
        localChannelManager);

//start the bridges object
MQeMQBridges localBridges = new MQeMQBridges();
localBridges.activate(environment);
```

Example 11-4: Starting a gateway queue manager

The differences from the server queue manager case are:

1. A bridges object is instantiated.
2. The bridges object is activated, and passed a rule that describes which child MQ bridges are to be loaded (using the default rule, all are loaded – however for a new gateway queue manager, none will exist).

A gateway queue manager can be stopped by using the various *close()* and *stop()* methods on the relevant objects:

```
//close the listener
localChannelListener.stop();

//close the bridges object
localBridges.close();

//close the queue manager
MQeQueueManager qMgr = MQeQueueManager.getReference(null);
 qMgr.close();
```

Example 11-5: Stopping a gateway queue manager

[25] Although the gateway environment set-up is identical to server environment set-up, be aware that in the examples used here, the *environment* variable for a gateway includes an additional *[MQBridge]* stanza that is used when the gateway queue manager is being started.

Executing the sample – *Running a gateway queue manager*:

Use the command: *MQeConfigGuide 9*. The sample code is contained in the methods *startGatewayQMgr()* and *stopGatewayQMgr().* By default, the MQe_Explorer-created *myGatewayQMgr.ini* file is used; by editing the sample code, the method *createGatewayEnvironment()* can be used instead.

The gateway queue manager is not able to exchange messages with MQSeries until various child objects of the bridges object are created and configured. This can only be done through admin messages; for more details of the mechanisms to be used, see *Admin programming* on page 47. The objects that must be created form two families, i.e.

Connection definition (to the MQSeries queue manager)

Remote queue definition (of type Bridge queue)

MQ bridge (child of the bridges object)

MQ queue manager proxy

MQ client connection

MQ listener

For a detailed example describing these objects, along with their required properties, see the *Gateway Configuration and Usage* script in the *MQe_Explorer User Guide*.

11.3 Peer queue managers

A peer queue manager is simply a queue manager that has the ability to listener for incoming peer connection requests, i.e. it has an active peer channel listener. The peer listener is created and configured through admin messages as a special case of a local connection – this was discussed in detail in the chapter *Listeners* on page 82.

Creating a peer queue manager is therefore identical to creating a client queue manager, with an additional step of creating a peer channel listener, once the queue manager is up and running.

This can be demonstrated by creating a peer queue manager *myPeerQMgr*. Note that this queue manager will be unable to receive messages from the other queue managers created so far (i.e. *myClientQMgr*, *myServerQMgr* and *myGatewayQMgr*) unless they are each further configured with appropriate connection definitions (specifying the correct port, address, adapter and the peer channel type). Moreover, if these queue managers are to act as slaves and respond to connection requests from *myPeerQMgr*, then they must each also have a peer channel listener defined.

Using MQe_Explorer

Run MQe_Explorer and create a new peer queue manager as below. On the *General* tab set the input fields as shown:

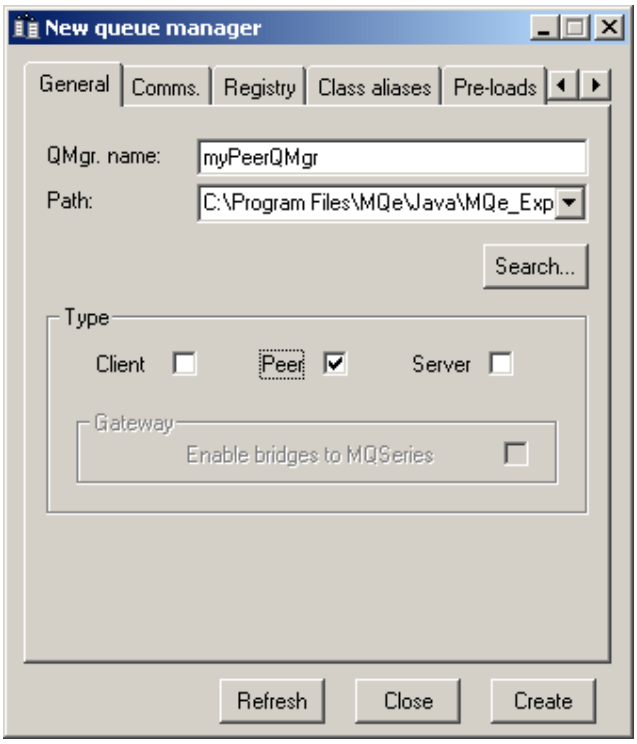

Figure 11-7: Creating a peer queue manager – General tab

On the *Comms*. tab, set the peer listener properties as shown:

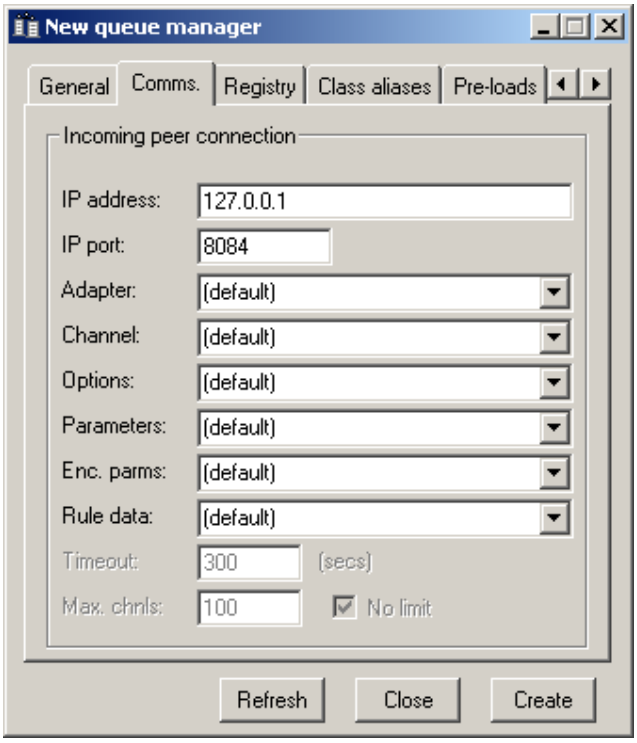

Figure 11-8: Creating a peer queue manager – Comms. tab

This creates a peer queue manager *myPeerQMgr* that listens for incoming connections on the TCP/IP port 8084 and uses the *com.ibm.mqe.adapters.MQeTcpipHistoryAdapter* adapter.

The associated initialization file has the contents:

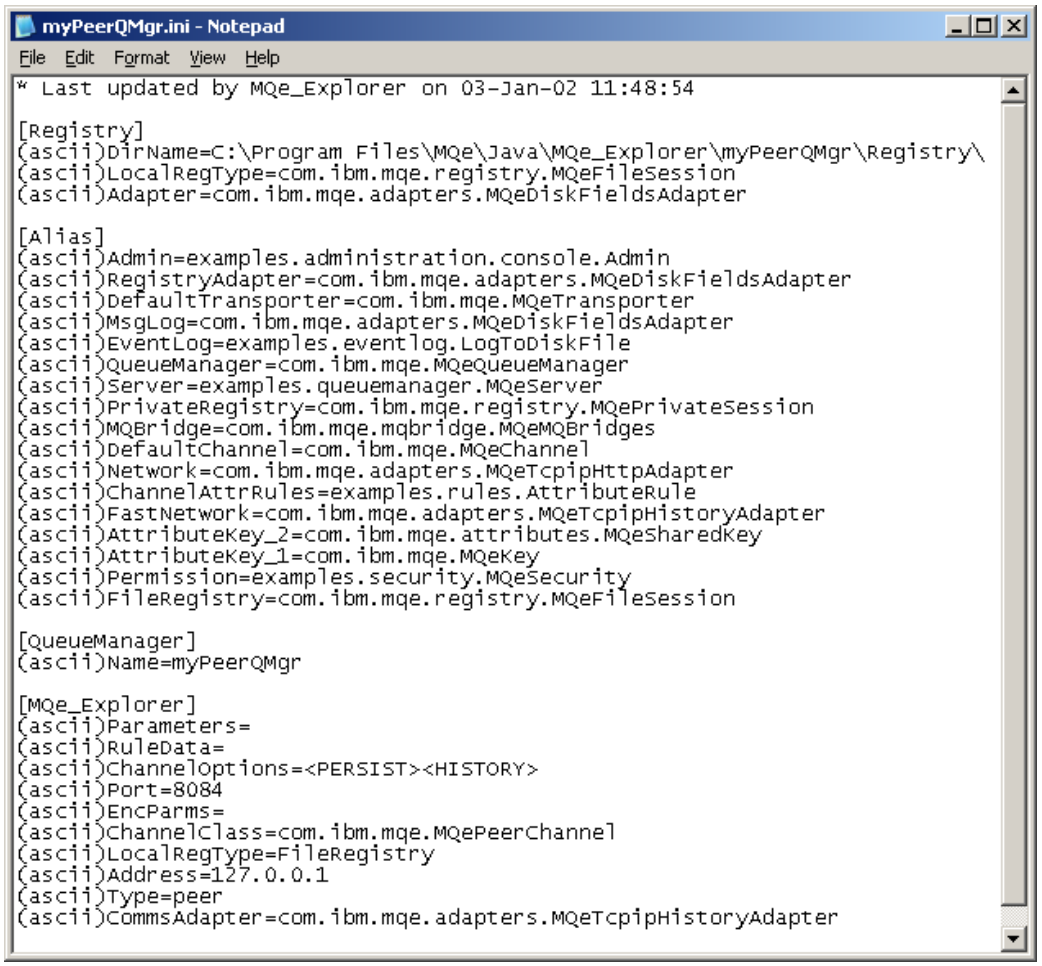

Figure 11-9: The myPeerQMgr initialization data

Comparing this data with that in *Figure 3-7: The myClientQMgr initialization data* on page 14:

1. All sections, excepting *[MQe_Explorer]*, are identical (apart from the queue manager name itself).
2. The *[MQe_Explorer]* includes, inter alia, the information necessary for MQe_Explorer to configure a peer channel listener.

Using code

The information in the sections: *Creating a client queue manager* on page 17, *Setting up a client environment* on page 14, and in *Starting a client queue manager* on page 15, is equally applicable to creating a peer queue manager.

The peer channel listener can be created with the code in the *Example 10-2: Creating a new peer listener* on page 87.

Executing the sample – *Running a peer queue manager*:

Use the command: *MQeConfigGuide 10*. The sample code is contained in the methods *startPeerQMgr()* and *stopPeerQMgr().* By default, the MQe_Explorer-created *myPeerQMgr.ini* file is used; by editing the sample code, the method *createPeerEnvironment()* can be used instead.

The *myPeerQMgr* queue manager already has a peer listener created by MQe_Explorer. If the queue manager type is changed to be a client and then MQe_Explorer is used to restart the queue manager, the peer listener will be replaced by a simple local connection definition. The code in *Sample 7* can then be edited to refer to *myPeerQMgr* instead of *myClientQMgr* and used to re-create the peer listener; note that the create command for the local connection must be changed to an update command (alternatively the local connection can first be deleted). If MQe_Explorer is subsequently used to start this queue manager it will convert it back to a client queue manager, in accordance with its queue manager type.

12 Queues

Queues can play an important part in determining message flow across MQe networks. MQe supports the following queue types:

- *Admin*

  A queue of type *admin queue* is a *local queue* (see below) that accepts admin messages. It processes those messages, executing their contained instructions

- *Bridge*

  A queue of type *bridge queue* is a proxy for a remote queue on an MQSeries queue manager. Bridge queues are only defined on MQe gateways.

- *Home server*

  A queue of type *home server queue* is a queue that points to a store and forward queue elsewhere in the MQe network. Home server queues pull messages destined for their local queue manager from those remote queues and deliver them to the relevant local queues. Home server queues should be regarded as system queues – messages are never addressed to home server queues and their contents are not accessible to applications.

- *Local*

  A queue of type *local queue* stores messages. Its contents are accessible to applications – and most messages are addressed to local queues.

- *Remote*

  A queue of type *remote queue* is a proxy for a local queue on a remote queue manager elsewhere in the MQe network. This remote queue manager is known as the *queue queue manager*. The proxy sends messages to the remote local queue; additionally, if the proxy is defined as asynchronous, then it will temporarily store the messages locally until it is able to send them. Applications are not able to view messages awaiting transmission; attempts to browse remote queue definitions are interpreted as a browse of the remote local queue itself.

- *Store and forward*

  A queue of type *store and forward queue* collects messages on its local queue manager that are destined for elsewhere in the MQe network. A single store and forward queue can collect messages addressed to one or more remote queue managers – these queue managers are referred to as *destination queue managers*. A store and forward queue can optionally send the messages it has collected to a remote queue manager – this named queue manager is called the *target queue manager*, i.e. the queue is forwarding messages to the target queue manager. In some cases it is desirable that the messages are just collected and not forwarded – the queue is now just a store, and there is no target queue manager defined. Store queues normally have messages pulled from them by home server queues. Store (and forward) queues should be regarded as system queues – messages are never addressed to store and forward queues and their contents are not accessible to applications.

From an MQe networking perspective therefore, the important queue types are: *local*, *remote*, *home server* and *store and forward*. Queues of type *local* are simply the destinations to which messages are addressed; the other queue types affect the way in which MQe flows the messages across the network to those destinations. Message flow is the subject of a later chapter.

12.1 Local queues

A local queue definition typically has the following basic properties, as presented by the MQe_Explorer management tool. Other advanced properties may be relevant, depending upon the values set for the basic properties.

Queue name:
Identifies the name of the local queue.

Local qMgr:
The name of the local queue manager owning the queue.

Adapter:
The class (or alias) of a *storage adapter* that provides access to the message storage medium (see *Storage adapters* on page 116).

Aliases:
Alias names are optional alternative names for the queue (see below).

Attribute rule:
The attribute class (or alias) associated with the security attributes of the queue (for more details see later in this chapter).

Authenticator:
The authenticator class (or alias) associated with the queue (for more details see later in this chapter).

Class:
The class (or alias) used to realize the local queue.

Compressor:
The compressor class (or alias) associated with the queue (for more details see later in this chapter).

Cryptor:
The cryptor class (or alias) associated with the queue (for more details see later in this chapter).

Description:
An arbitrary string describing the queue.

Expiry:
The time after which messages placed on the queue expire.

Max. depth:
The maximum number of messages that may be placed on the queue.

Max. message length:
The maximum length of a message that may be placed on the queue.

Message store:
The class (or alias) that determines how messages on the local queue are stored.

Path:
The location of the queue store.

Priority:
The default priority associated with messages on the queue.

Rule:
The class (or alias) of the rule associated with the queue; determines behavior when there is a change in state for the queue.

Target registry:
The target registry to be used with the authenticator class (i.e. “None”, “Queue”, or “Queue manager”).

12.2 Remote queues

A remote queue definition typically has the following basic properties, as presented by the MQe_Explorer management tool. Other advanced properties may be relevant, depending upon the values set for the basic properties. In the text below the phrase *remote queue* refers to the queue proxy; the phrase *target queue* refers to the local queue on the remote queue manager.

Queue name:
Identifies the name of the remote queue (which must be the same as the name of the target queue).

Local qMgr:
The name of the queue manager owning the remote queue.

Queue qMgr:
The name of the queue manager owning the target queue.

Adapter:
Only valid if the mode is asynchronous: The class (or alias) of a *storage adapter* that provides access to the message storage medium (see *Storage adapters* on page 116).

Aliases:
Alias names are optional alternative names for the remote queue (see below).

Attribute rule:
The attribute class (or alias) associated with the security attributes of the remote queue (for more details see later in this chapter).

Authenticator:
The authenticator class (or alias) associated with the remote queue (for more details see later in this chapter).

Class:
The class (or alias) used to realize the remote queue.

Close if idle:
Indicates if the transporter is to be closed after transmitting all available messages.

Compressor:
The compressor class (or alias) associated with the remote queue (for more details see later in this chapter).

Cryptor:
The cryptor class (or alias) associated with the remote queue (for more details see later in this chapter).

Description:
An arbitrary string describing the remote queue.

Expiry:
The time after which messages placed on the remote queue expire.

Max. depth:
Only valid if the mode is asynchronous: the maximum number of messages that may be placed on the remote queue.

Max. message length:
The maximum length of a message that may be placed on the remote queue.

Message store:
Only valid if the mode is asynchronous: The class (or alias) that determines how messages awaiting transmission on the remote queue are stored.

Mode:
Indicates whether the remote queue can store messages pending transmission (*asynchronous*) or cannot (*synchronous*).

Path:
Only valid if the mode is asynchronous: the location of the queue store for the remote queue.

Priority:
The default priority associated with messages on the remote queue.

Rule:
The class (or alias) of the rule associated with the remote queue; determines behavior when there is a change in state for the remote queue.

Target registry:
The target registry to be used with the authenticator class (i.e. “None”, “Queue”, or “Queue manager”).

Transporter:
The class (or alias) that flows messages over the channel to the target queue.

XOR compress:
Indicates whether the transporter is to implement XOR compression of the field data with previous data (this process also assists any compressor that may have been defined).

12.3 Home server queues

A home server queue definition typically has the following basic properties, as presented by the MQe_Explorer management tool. Other advanced properties may be relevant, depending upon the values set for the basic properties.

Queue name:
Identifies the name of the home server queue (which must be the same as the name of the target store and forward queue).

Local qMgr:
The name of the queue manager owning the home server queue.

Queue qMgr:
The name of the queue manager owning the target store and forward queue.

Attribute rule:
The attribute class (or alias) associated with the security attributes of the queue (for more details see later in this chapter).

Authenticator:
The authenticator class (or alias) associated with the queue (for more details see later in this chapter).

Class:
The class (or alias) used to realize the home server queue.

Close if idle:
Indicates if the transporter is to be closed after transmitting all available messages.

Compressor:
The compressor class (or alias) associated with the queue (for more details see later in this chapter).

Cryptor:
The cryptor class (or alias) associated with the queue (for more details see later in this chapter).

Description:
An arbitrary string describing the home server queue.

Target registry:
The target registry to be used with the authenticator class (i.e. “None”, “Queue”, or “Queue manager”).

Time interval:
The time between the attempts to get messages from the target store and forward queue.

Transporter:
The class (or alias) that flows messages over the channel from the store and forward queue.

XOR compress:
Indicates whether the transporter is to implement XOR compression of the field data with previous data (this process also assists any compressor that may have been defined).

12.4 Store and forward queues

A store and forward queue definition typically has the following basic properties, as presented by the MQe_Explorer management tool. Other advanced properties may be relevant, depending upon the values set for the basic properties.

Queue name:
Identifies the name of the store and forward queue.

Local qMgr:
The name of the queue manager owning the store and forward queue.

Target qMgr:
Only valid if the queue is to forward messages: the name of the queue manager to which messages are forwarded.

Message store:
The class (or alias) of a *storage adapter* that provides access to the message storage medium (see *Storage adapters* on page 116).

Attribute rule:
The attribute class (or alias) associated with the security attributes of the queue (for more details see later in this chapter).

Authenticator:
The authenticator class (or alias) associated with the queue (for more details see later in this chapter).

Class:
The class (or alias) used to realize the store and forward queue.

Close if idle:
Indicates if the transporter is to be closed after transmitting all available messages.

Compressor:
The compressor class (or alias) associated with the queue (for more details see later in this chapter).

Cryptor:
The cryptor class (or alias) associated with the queue (for more details see later in this chapter).

Description:
An arbitrary string describing the store and forward queue.

Destinations:
The list of remote queue manager names for which the store and forward queue will store messages.

Expiry:
The time after which messages placed on the queue expire.

Max. depth:
The maximum number of messages that may be placed on the store and forward queue.

Max. message length:
The maximum length of a message that may be placed on the queue.

Message store:
The class (or alias) that determines how messages awaiting transmission on the store and forward queue are stored.

Path:
The location of the queue store for the store and forward queue.

Priority:
The default priority associated with messages on the queue.

Rule:
The class (or alias) of the rule associated with the store and forward queue; determines behavior when there is a change in state for the store and forward queue.

Target registry:
The target registry to be used with the authenticator class (i.e. “None”, “Queue”, or “Queue manager”).

Transporter:
The class (or alias) that flows messages over the channel to the target queue manager (if required).

XOR compress:
Indicates whether the transporter is to implement XOR compression of the field data with previous data (this process also assists any compressor that may have been defined).

12.5 Queue alias names

Queue definitions are used when MQe needs to deliver a message to either a remote or local queue; the queue name identifies the target queue. In some cases however a degree of independence is needed between the queue name used by sending applications, and the actual names of queues at the target queue manager. This flexibility is provided through aliases.

Queues have an *aliases* property, which allows zero, one or more alias names to be associated with the queue. These aliases are not just alternative names that map to the same queue; they also change the destination queue name in the message. This latter property is important for remote messaging, otherwise the message would be put on a channel destined for the target queue manager, but would be rejected when it arrived because of the queue name mismatch.

Consequently if a message is addressed to a queue name that is defined as an alias of a queue name, then MQe changes the name in the message address to that queue name.

13 Security

MQe provides two security mechanisms directly concerned with the transport of messages:

- Message-based security:
  - Messages are protected by the application, using MQe services, and passed to MQe for transport in a fully protected state. MQe delivers the messages to a target queue, from which they are removed by an application and subsequently unprotected, again using MQe services. Since the messages are fully protected when being directly handled by MQe, they can be flowed over clear channels and held on unprotected intermediate queues.
- Queue-based security
  - Messages are not protected by the application and therefore passed to MQe for transport in an unprotected state. MQe delivers the messages to a target queue, from which they are removed by an application, being delivered by MQe in an unprotected state. MQe protects the messages on receipt and flows them over secure channels; they are also held protected on any intermediate queues and on the destination queue.

Message-based security is beyond the scope of this book.

13.1 Queue-based security

Introduction

The level of queue-based security to be used is determined through the setting of attributes on queues. As a consequence of these attributes, MQe uses appropriate secure channels, cryptors and compressors and controls access (if required) through authenticators. The relevant queue properties are:

- Compressor
- Cryptor
- Authenticator
- Attribute rule

These properties can be set on all queue definitions; the effect they have depends upon the kind of queue definition involved, i.e.

- **Local queue**: determines how the data is stored and whether the incoming channel characteristics are acceptable.
- **Remote queue**: determines how the data is stored pending transmission (if applicable) and how the outgoing channel is negotiated.
- **Store & forward queue**: determines how the data is stored pending transmission and how the outgoing channel is negotiated (if applicable).
- **Home server queue**: determines how the outgoing channel is negotiated.

The *compressor* determines whether the data should be compressed. The choice of compressor is one of:

(*null*)
com.ibm.mqe.attributes.GZIPCompressor
com.ibm.mqe.attributes.LZWCompressor
com.ibm.mqe.attributes.RleCompressor

Details of the compressor characteristics are given Compressor classes on page 114.

The *cryptor* determines whether the data should be encoded to hide the significance of the contents. The choice of cryptor is one of:

(*null*)
com.ibm.mqe.attributes.MQe3DESCrypto
com.ibm.mqe.attributes.MQeDESCryptor
com.ibm.mqe.attributes.MQeMARSCryptor
com.ibm.mqe.attributes.MQeRC4Cryptor
com.ibm.mqe.attributes.MQeRC6Cryptor
com.ibm.mqe.attributes.MQeXorCryptor
examples.attributes.TableCryptor

Details of the cryptor characteristics are given in *Cryptor* on page 114. The specification of certain cryptors is only allowed if the queue manager registry is of the *private registry* type; see *Figure 4-1: Registry selection based on security requirements* on page 20.

The *authenticator* determines whether the data access should be controlled. The choice of authenticator is one of:

(null)
com.ibm.mqe.attributes.MQeWTLSCertAuthenticator
examples.attributes.UseridAuthenticator
examples.attributes.NTAuthenticator
examples.attributes.UnixAuthenticator

Details of the authenticator characteristics are given in *Authenticator* on page 115. The specification of certain authenticators is only allowed if the queue manager registry is of the *private registry* type (see *Figure 4-1: Registry selection based on security requirements* on page 20).

If the *com.ibm.mqe.attributes.MQeWTLSCertAuthenticator* class authenticator is used then an additional parameter *target registry* must also be set. This parameter determines which registry is to supply the credentials for authentication, and can have the value of either "Queue manager" or "Queue".

If "Queue manager" is specified, then the credentials used are those of the queue manager owning the queue, and come from the private registry of the queue manager. The queue manager originally obtains these credentials through auto-registration with the mini-certificate server (see the section *Auto-registration with the mini-certificate server* on page 27). This option is the recommended default.

If "Queue" is specified, then the credentials used are those of the queue itself, and come from the private registry of the queue. The queue originally obtains these credentials through auto-registration with the mini-certificate server (see below).

The *attribute rule* determines whether a channel is allowed access to the queue. It does this by comparing the security properties of the queue, with those of the channel. The attribute rule is therefore important when the channel attributes and the queue attributes are not identical. MQe ships an example attribute rule:

examples.attributes.AttributeRule

Details of the behavior enforced by this rule, and the way in which attributes rules are used, is given in *Channel security* on page 65.

Channel security considerations

For efficiency in queue-based security, an MQe channel uses symmetric cryptors (e.g. DES, 3DES, MARS, RC4, RC6); a consequence of which is that the two queue managers at either end must use the same encryption key. When such a channel is established, a protocol, called the Diffie Hellman key exchange, is used to establish a secret key that only the two queue managers know. This protocol is susceptible to a "man in the middle" attack, but for that to be successful, the "man in the middle" must know some of the data that is fed into the Diffie Hellman protocol. This data is held in the *com.ibm.mqe.attributes.MQeDHk* class. It is possible for an attacker to get hold of this data, by examining the shipped MQe classes. However, this data can be changed by running the *com.ibm.mqe.attributes.MQeGenDH* utility; it generates a new java source file *com.ibm.mqe.attributes.MQeDHk.java*. This file can then be compiled into a replacement *com.ibm.mqe.attributes.MQeDHk.class* file.

Setting up a private registry for a queue

A private registry for a queue is only relevant where:

1. Queue-based security is to be used.
2. The authenticator required is: *com.ibm.mqe.attributes.MQeWTLSCertAuthenticator*.
3. The target registry property of the queue has been set to "Queue" (i.e. credentials owned by the queue itself are to be used during authentication, rather than credentials owned by the queue manager).

In order to establish such a private registry, the following conditions must be met:

1. The owing queue manager must itself have a registry of type *private registry* (see *Registry* on page 19).
2. The owning queue manager must have previously auto-registered with the mini-certificate server (see *Auto-registration with the mini-certificate server* on page 27.
3. In starting the queue manager, the parameters *CertReqPIN*, *KeyRingPassword*, and *CAIPAddrPort* were passed whilst the opening the registry (these parameters are described in the section *Auto-registration with the mini-certificate server)*.
4. The mini-certificate server is running when the authenticator property is being set (or modified) and has been primed to issue a mini-certificate for the queue.

Where MQe_Explorer is used to create and launch the queue manager, the following conditions apply:

1. The owing queue manager must itself have a registry of type *private registry* (see *Registry* on page 19).
2. The option *Allow queue registry* (on the *Security* tab) must have been checked at creation time.
3. The owning queue manager must have previously auto-registered with the mini-certificate server (this should have happened automatically).
4. The mini-certificate server is running when the authenticator property is being set (or modified) and has been primed to issue a mini-certificate for the queue.

Setting up the MQe_MiniCertServer to issue queue credentials is very similar to the queue manager case described in *Enabling MQe_MiniCertServer to issue a certificate* on page 27. The only difference is that to create a new queue authenticable entity, right click on the parent queue manager in the tree pane and select *New Entity* (or select the node and use the *File→New→Entity* menu item). The new entity dialog appears:

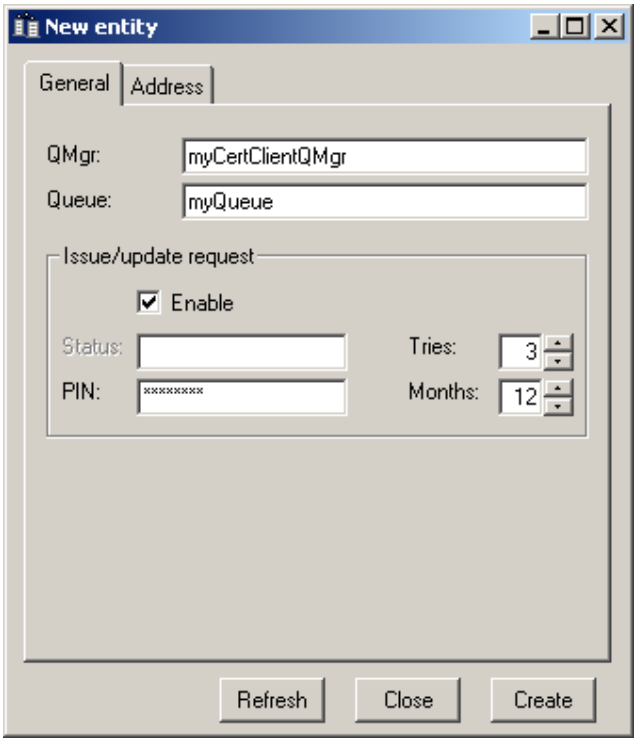

Figure 13-1: The new queue entity dialog – General tab

Enter the following data on the *General* tab:

1. *Queue: the name of the queue*
2. *PIN: the mini-certificate request pin to be used by the queue when retrieving its certificate*

When the queue is created (or modified) a process exactly analogous to queue manager auto-registration takes place, with the queue acquiring a registry containing its own mini-certificate. There is no specific programming required to accomplish queue auto-registration.

Queue credential examination and renewal is discussed in the chapter *Certificate management* on page 133.

13.2 Security classes

Security classes are used in both message-level and queue-level security. Where queue-level security is in use, MQe automatically associates security classes with the channels used to move messages between queue managers. In summary, the following classes are available:

Compressor classes

com.ibm.mqe.attributes.MQeGZIPCompressor

Compresses and decompresses a byte array using the GZIP algorithm. This likely to be effective where the data has frequently repeating words or byte patterns; the algorithm uses a reference back to the first occurrence of a pattern.

com.ibm.mqe.attributes.MQeLZWCompressor

Compresses and decompresses a byte array using the LZW algorithm. The algorithm uses a dictionary structure and is likely to be effective where the data has frequently repeating words or byte patterns.

com.ibm.mqe.attributes.MQeRleCompressor

Compresses and decompresses a byte array using a simple run length encoding algorithm. This is effective where the data to be compressed contains strings of repeated bytes.

Cryptor classes

com.ibm.mqe.attributes.MQe3DESCryptor

Provides an *MQeCryptor* object that uses triple DES CBC to encrypt/decrypt a byte array.

com.ibm.mqe.attributes.MQeDESCryptor

Provides an *MQeCryptor* object that uses DES CBC to encrypt/decrypt a byte array.

com.ibm.mqe.attributes.MQeMARSCryptor

Provides an *MQeCryptor* object that uses MARS to encrypt/decrypt a byte array.

com.ibm.mqe.attributes.MQeRC4Cryptor

Provides an *MQeCryptor* object that uses RC4 to encrypt/decrypt a byte array.

com.ibm.mqe.attributes.MQeRC6Cryptor

Provides an *MQeCryptor* object that uses RC6 to encrypt/decrypt a byte array.

com.ibm.mqe.attributes.MQeXorCryptor

Provides an *MQeCryptor* object that uses a simple exclusive OR algorithm to disguise the data. This cryptor does not provide a high level of data protection but obscures the data sufficiently to prevent visual recognition.

examples.attributes.TableCryptor

Provides an example of an *MQeCryptor* object that uses a simple data substitution table to encrypt/decrypt a byte array. The source code is provided.

Authenticator classes

com.ibm.mqe.attributes.MQeWTLSCertAuthenticator

Manages mutual authentication using WTLS size-optimized certificates.

examples.attributes.NTAuthenticator

A sample authenticator, specific to Windows NT/2000/XP systems, that presents a simple GUI interface for the local user to supply his/her Windows user credentials (user id/password/domain). The source code is provided.

examples.attributes.UnixAuthenticator

Provides an example of an authenticator, specific to Unix systems, that presents a simple GUI interface for the local user to supply his/her user credentials (user id/password). The source code is provided.

examples.attributes.UseridAuthenticator

Provides an example of an authenticator using a user id and password file that presents a simple GUI interface for the local user to supply his/her user credentials. The source code is provided.

14 Storage adapters

MQe uses storage adapters to provide access to underlying storage media. In summary, the following are available:

com.ibm.mqe.adapters.MQeDiskFieldsAdapter

Provides a persistent store for data, using the file system. Typically this is the default adapter for queues and the registry, since it offers the greatest assurance that data has not been lost. It does not rely on the operating system to complete lazy writes.

com.ibm.mqe.adapters.MQeCaseInsensensitiveDiskAdapter

A subclass of the com.ibm.mqe.adapters.MQeDiskFieldsAdapter that performs its comparison of filenames in a case insensitive fashion. This is required on some JVM/OS combinations (e.g. IBM 4690 OS) where the case of the name of the file created is not the same as that reported from a *File.list()* method call; in these cases the standard disk fields adapter will not match the files.

com.ibm.mqe.adapters.MQeMemoryFieldsAdapter

Provides a non-persistent, temporary store for data, using memory. Typically this adapter is used to store queues where fast access is required and where the messages need not survive the queue manager, nor survive system failure. In the current release, a registry service provider cannot use this adapter.

com.ibm.mqe.adapters.MQeReducedDiskFieldsAdapter

Provides a persistent store for data, using the file system. This adapter is a higher-speed alternative to the *com.ibm.mqe.adapters.MQeDiskFieldsAdapter*. However, it does introduce a dependency on the operating system staying up long enough to complete lazy writes.

15 Single hop messaging

In previous chapters we have seen the various types of queue manager (client, server, gateway etc.) and understood that messages flow between queue managers over channels, created and managed by queue managers. Moreover, the nature of these channels are determined by connection definitions, with each queue manager having a set of connection definitions that define how channels are to be created to remote queue managers[26].

None of these considerations are visible to application programs. In this chapter we will consider message flow from both an application and from an administrative perspective, using just a pair of queue managers.

The code required to create and send a message is very simple[27]:

```
//get addressability to the local queue manager
MQeQueueManager qMgr = MQeQueueManager.getReference(null);

//set strings
String targetQMgrName = "myServerQMgr";
String targetQName = "myTargetQueue";

//create the message
MQeMsgObject message = new MQeMsgObject();

//send the message
qMgr.putMessage(targetQMgrName, targetQName, message, null, 0);
```

Example 15-1: Sending a message

The key method call in the example above is *putMessage()*. This behaves as follows:

1. It attempts to send the message *on its way* to the final destination (i.e. a queue of type *local* on the destination queue manager). If it can do so it will return normally. Such a normal return means that the message has either reached its destination or is held in some intermediate queue. If held in an intermediate queue it is the responsibility of MQe to move the message progressively to its destination. Network errors or incorrect configuration may mean that such a message cannot be delivered – but it will not be inadvertently lost or discarded.
2. If it cannot be sent *on its way* to its final destination then MQe will throw an exception – the reason will be found in the exception data. MQe does not have the message and is not responsible for its delivery – that remains with the application.

[26] This is a slight simplification that fails to take into account: (a) peer channels established by the remote queue manager – but then used by the local queue manager; (b) home server queues on a remote queue manager – that pull messages from the local queue manager over channels established by the remote queue manager; (c) store and forward queues – that send messages destined to remote queue managers through intermediate queue managers. These exceptions are detailed later.

[27] In this book we are neither concerned with *attribute* objects on messages, nor with the use of the *confirm id*.

MQe is frequently described as supporting either *synchronous* or *asynchronous* delivery. The situation is much more complicated; in fact:

1. Messages may be delivered over multiple hops from the sending queue manager to the target queue manager.
2. Each hop can be regarded as having the potential to be synchronous or asynchronous:
    i. An asynchronous hop is one where the message is queued at the queue manager sending over the hop.
    ii. A synchronous hop is one where the message cannot be queued at the queue manager sending over the hop.
3. Messages are always delivered to a target queue (i.e. they are queued until collected by an application).

Where only one hop is involved, it can be seen that the above description reflects either synchronous or asynchronous messaging over that hop.

The simplest MQe network that can be envisaged is one that comprises just two queue managers (*SourceQMgr* and *TargetQMgr*) and where each may freely communicate with the other. By freely is meant that each can establish channels to the other whenever data needs to be sent. This condition is satisfied by:

1. Both *SourceQMgr* and *TargetQMgr* are server queue managers, each with an appropriate connection definition to the other[28].
2. Both *SourceQMgr* and *TargetQMgr* are peer queue managers, each with an appropriate connection definition to the other.

The case where *SourceQMgr* is a client and *TargetQMgr* is a server will be described later. For the purpose of the next examples, two server queue managers will be used.

[28] As a reminder – this configuration of server queue managers is acceptable because a server automatically has client capabilities. So each can act both as a client and as a server to the other.

15.1 Synchronous operation

The figure below shows an application on *SourceQMgr* sending a message to the *TargetQ* queue on *TargetQMgr*.

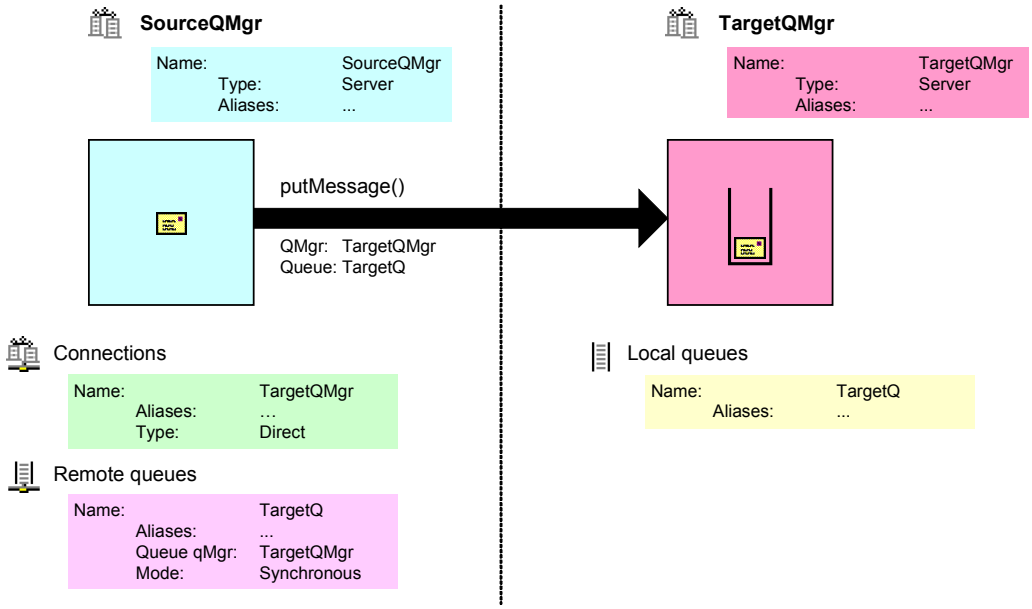

Figure 15-1: Direct, synchronous messaging

The objects present on each queue manager are shown, together with their key properties. *TargetQMgr* only has the destination queue defined, which is of type *local queue*. The sending queue manager, *SourceQMgr*, has two relevant objects: a connection definition to he target and a remote queue definition.

The connection definition on *SourceQMgr* allows MQe to set up a channel to *TargetQMgr*. If this were not present, then the application would get an exception on attempting the *putMessage()* call.

The remote queue definition on *SourceQMgr* gives MQe the details of the destination queue[29]. In this case the important property of the remote queue definition is the *mode*, which has the value of *synchronous*. It is this value that determines that the delivery will be synchronous over the hop. Another way of viewing the situation is that the remote queue on *SourceQMgr* does not have the ability to store messages awaiting transmission, and so synchronous messaging is the only option.

[29] In this case the details are trivial. However if the destination queue had security attributes set then these would be reflected in the remote queue definition on *SourceQMgr* and would be used by MQe to ensure that a secure channel was used to move the message.

If the remote queue definition on *SourceQMgr* were not present, in this simple case[30] MQe would itself attempt to determine details of the *TargetQ* on *TargetQMgr* – this is known as *queue discovery*. If it were able to do so, it would then create a synchronous remote queue definition on *SourceQMgr* (exactly as shown in the figure), and then synchronously transfer the message. If it could not, then the application would receive an exception on the *putMessage()*. The following characteristics of the target queues are discovered (and thus used in the remote queue definition[31]):

Attribute rule
Authenticator
Compressor
Cryptor
Description
Expiry
Max. depth
Max. message size
Priority
Target registry

If for any reason, such as network failure, the synchronous *putMessage()* cannot be completed, the application receives an exception. If the application needs to determine the exact state of message transfer, e.g. if the message got sent just before the error occurred, then the more advanced form of *putMessage()* is used, exploiting *confirm ids*. This topic is outside the scope of this book.

If application(s) at *SourceQMgr* need to send to multiple queues on *TargetQMgr*, then multiple queue definitions are needed at *SourceQMgr*, one remote queue definition for each target queue.

The above configuration is equally applicable to the situations where:

(a) *SourceQMgr* and *TargetQMgr* are both peers.

(b) *SourceQMgr* is a client and *TargetQMgr* is a server.

---

[30] Queue discovery would not take place if, for example, *SourceQMgr* had a store and forward queue that collected messages for *TargetQMgr*.

[31] This default behavior may not always be desirable, e.g. the presence of MQe-created remote queue definitions may prevent user-created definitions being established (until deleted). Certain properties of the remote queue definition need not match the target queue (e.g. description, max. queue depth etc).

15.2 Asynchronous operation

The figure below shows an application on *SourceQMgr* sending a message to the *TargetQ* queue on *TargetQMgr*; unlike the previous example, the remote queue definition on *SourceQMgr* has the *mode* property set to *asynchronous*.

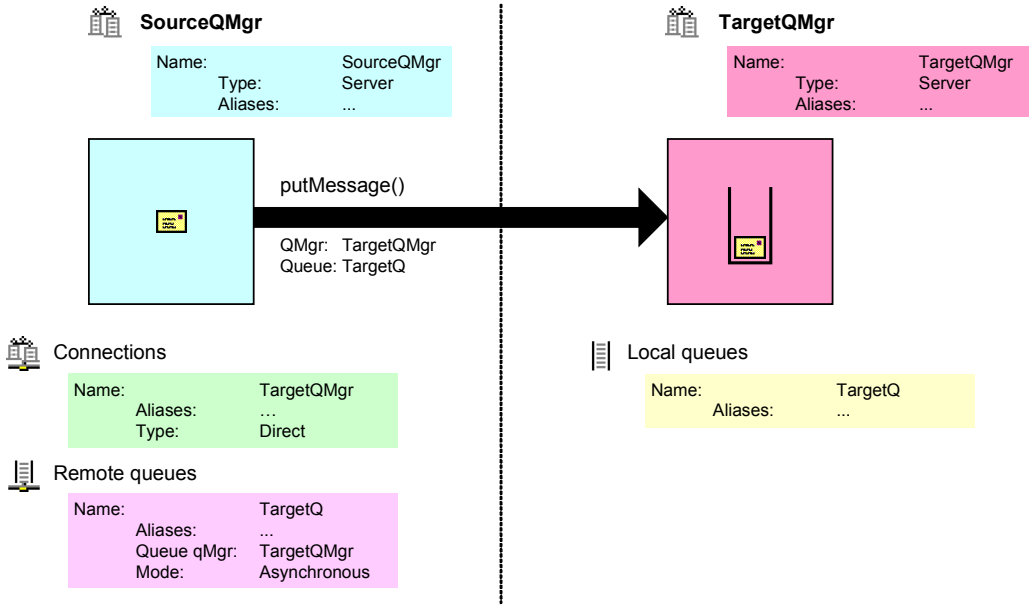

Figure 15-2: Direct, asynchronous messaging

Remote queue definition *TargetQ* now has the ability to store messages awaiting transmission. When the application does the *putMessage()* the message will first be stored on the remote queue definition on *SourceQMgr*. An attempt will be made to send the messages when the queue manager is triggered; by default this happens when a message is put on a queue for delivery. A discussion of triggering is beyond the scope of this book.

The messages will therefore be sent asynchronously over the hop. The application is very unlikely get an exception from the *putMessage()* call, certainly network or target queue manager unavailability would not be a reason.

If application(s) at *SourceQMgr* need to send to multiple queues on *TargetQMgr*, then multiple queue definitions are needed at *SourceQMgr*, one remote queue definition for each target queue.

The above configuration is equally applicable to the situations where:

(a) *SourceQMgr* and *TargetQMgr* are both peers.

(b) *SourceQMgr* is a client and *TargetQMgr* is a server.

15.3 Synchronous and asynchronous operation

In some cases it may be necessary to support simultaneous (or sequential) synchronous and asynchronous communication from a queue manager to a target queue. The figure below shows the configuration required:

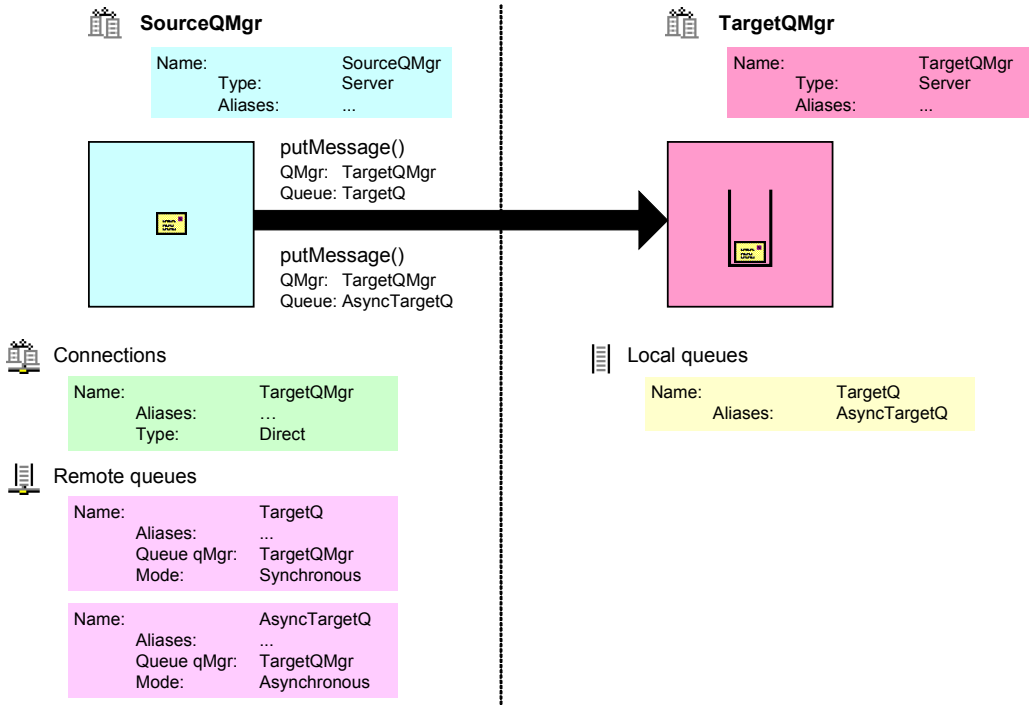

Figure 15-3: Direct, synchronous and asynchronous messaging

The key to this configuration is the alias name of *AsyncTargetQ* set for the *TargetQ* on *TargetQMgr*. Messages that arrive on queue manager *TargetQMgr* addressed to either *TargetQ* or *AsyncTargetQ* will be resolved to the same local queue.

On the *SourceQMgr* two remote queue definitions have been defined; remote queue *TargetQ* has the mode set to *synchronous*; remote queue *AsyncTargetQ* has mode set to *asynchronous*.

Two *putMessage()* calls are shown; the first will be synchronous because it specifies the target queue as *TargetQ*; the second will be asynchronous because it specified the target queue as *AsyncTargetQ*. Both calls will result in their messages arriving at the same local queue on *TargetQMgr*.

The above configuration is equally applicable to the situations where:

(a) *SourceQMgr* and *TargetQMgr* are both peers.

(b) *SourceQMgr* is a client and *TargetQMgr* is a server.

15.4 Source aliases

Aliases have many uses. For example in the configuration below they are used at *SourceQMgr* to redirect messages from an (old) application to the correct target queue and queue manager. Synchronous messaging has been used but the principles apply equally to asynchronous messaging.

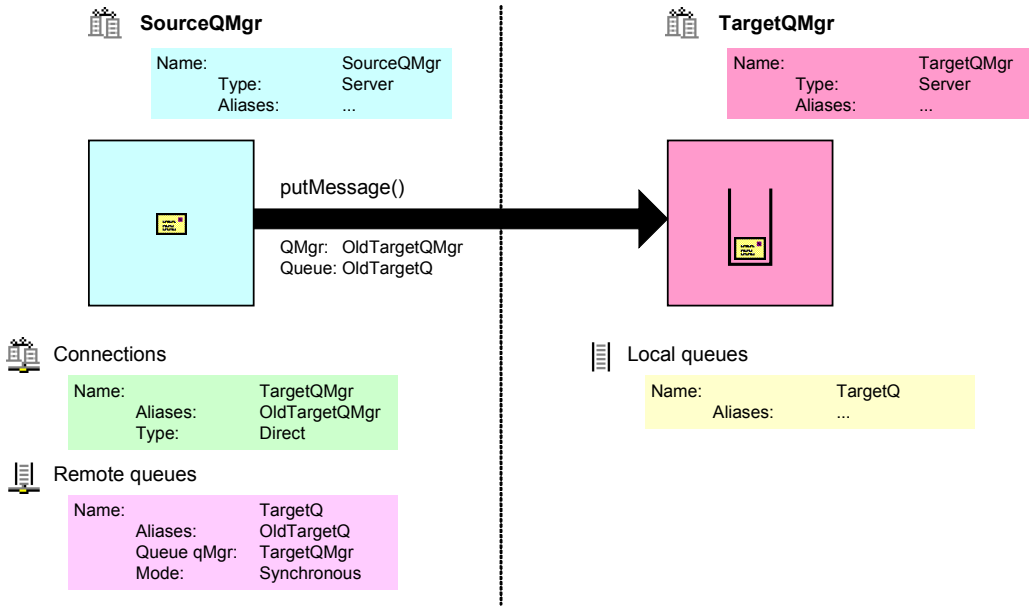

Figure 15-4: Direct, synchronous messaging with source aliasing

The application consequences of this configuration are identical to that shown in *Figure 15-1: Direct, synchronous messaging* on page 119. In this case the application refers to the target queue and queue manager as *OldTargetQ* and *OldTargetQMgr* respectively; however these are substituted in the message with the names *TargetQ* and *TargetQMgr*. At *TargetQMgr* there is no discernable difference from the earlier example.

The above configuration is equally applicable to the situations where:

(a) *SourceQMgr* and *TargetQMgr* are both peers.

(b) *SourceQMgr* is a client and *TargetQMgr* is a server.

15.5 Destination aliases

Similarly, aliases can be used at the destination (or indeed elsewhere in the network) to resolve naming differences. For example:

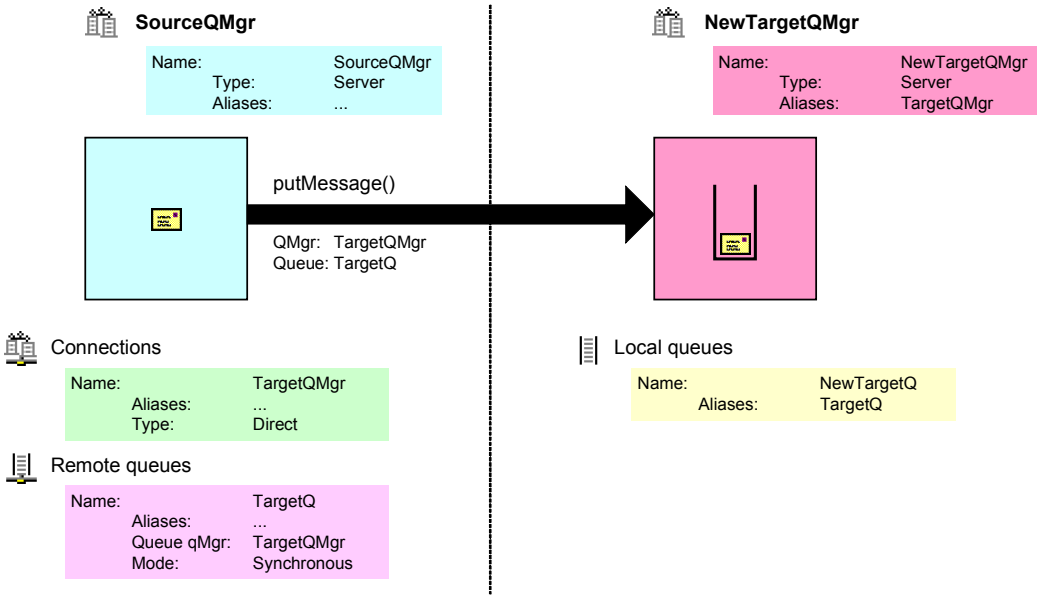

Figure 15-5: Direct, synchronous messaging with destination aliasing

Here the destination queue manager and queue has been changed from that expected by both the application and configuration objects at *SourceQMgr*. However use of a queue manager alias and a local queue alias at *NewTargetQMgr* means that the messages are accepted onto the target queue.

The above configuration is equally applicable to the situations where:

(a) *SourceQMgr* and *TargetQMgr* are both peers.

(b) *SourceQMgr* is a client and *TargetQMgr* is a server.

15.6 Client/server operation

The examples so far have shown a source queue manager pushing messages to a target queue manager. If *SourceQMgr* were a server queue manager and *TargetQMgr* a client queue manager then this would not be possible, because the client cannot accept an incoming channel request. Consequently, in this case *TargetQMgr* must pull the messages from the source.

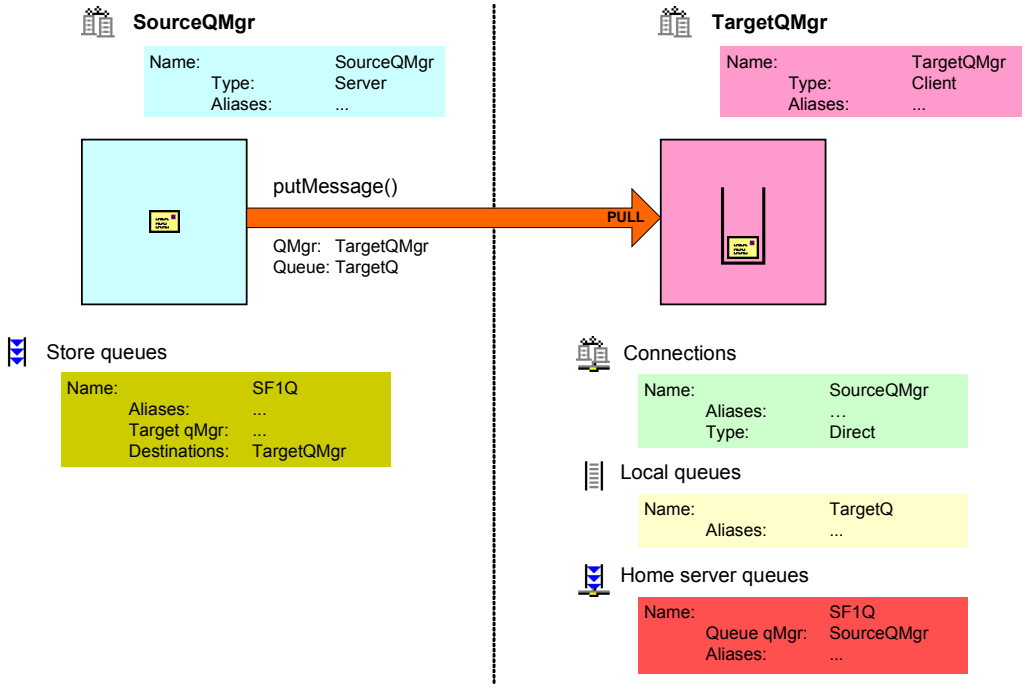

Figure 15-6: Client/server messaging

In this configuration the server queue manager has no connection definition to the client; instead it has a store queue (i.e. a store and forward queue with no target queue manager) that collects all messages bound for the client. This message collection embraces all queue destinations on the client.

The client pulls the messages from the store queue using a home server queue pointing at the store queue on the client. The home server queue never stores messages itself – it collects them from the store queue and delivers them to their destinations on the client. The client makes the connection request to the server using its connection definition.

The above configuration is equally applicable to the situations where:

(a) *SourceQMgr* and *TargetQMgr* are both peers.

(b) *SourceQMgr* and *TargetQMgr* are both servers.

15.7 Use of store and forward queues

An attractive feature of the configuration above, where messages are pulled, is that the server object definitions apply independently of the specific queues that must be accessed on the client. A similar result can be achieved for cases where messages are pushed.

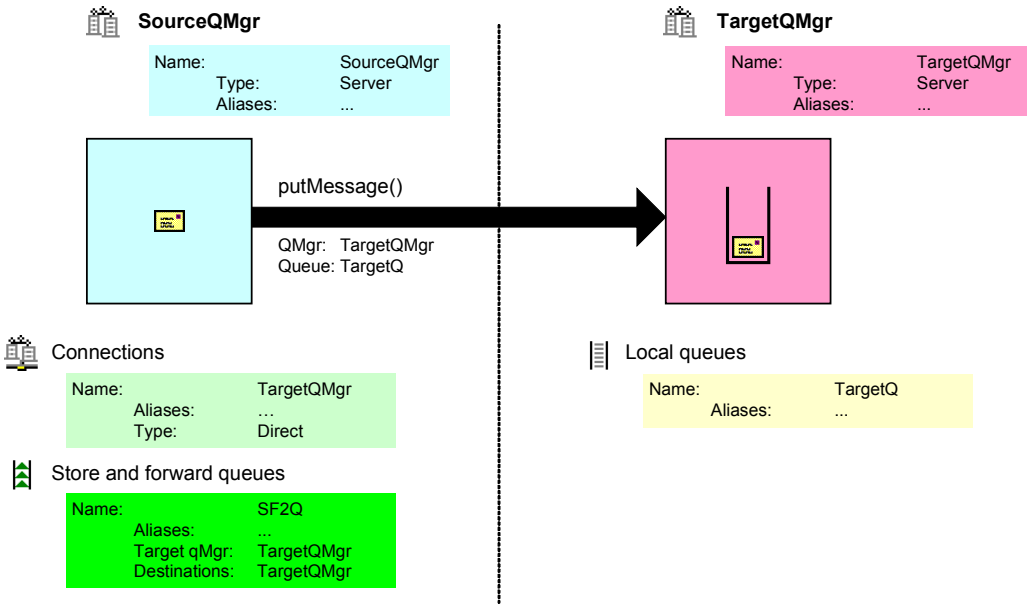

Figure 15-7: Store and forward queues in messaging

The store and forward queue *SF2Q* collects all messages on *SourceQMgr* bound for the *TargetQMgr* queue manager and forwards them, irrespective of their target queue.

The above configuration is equally applicable to the situations where:

(a) *SourceQMgr* and *TargetQMgr* are both peers.

(b) *SourceQMgr* is a client and *TargetQMgr* is a server.

15.8 Using both remote and store & forward queues

In the configuration below, both a store and forward queue, and a remote queue are defined on *SourceQMgr*. The *TargetQMgr* has two local queues, *TargetQ* and *TargetQ1*, to which messages are put by an application on *SourceQMgr*.

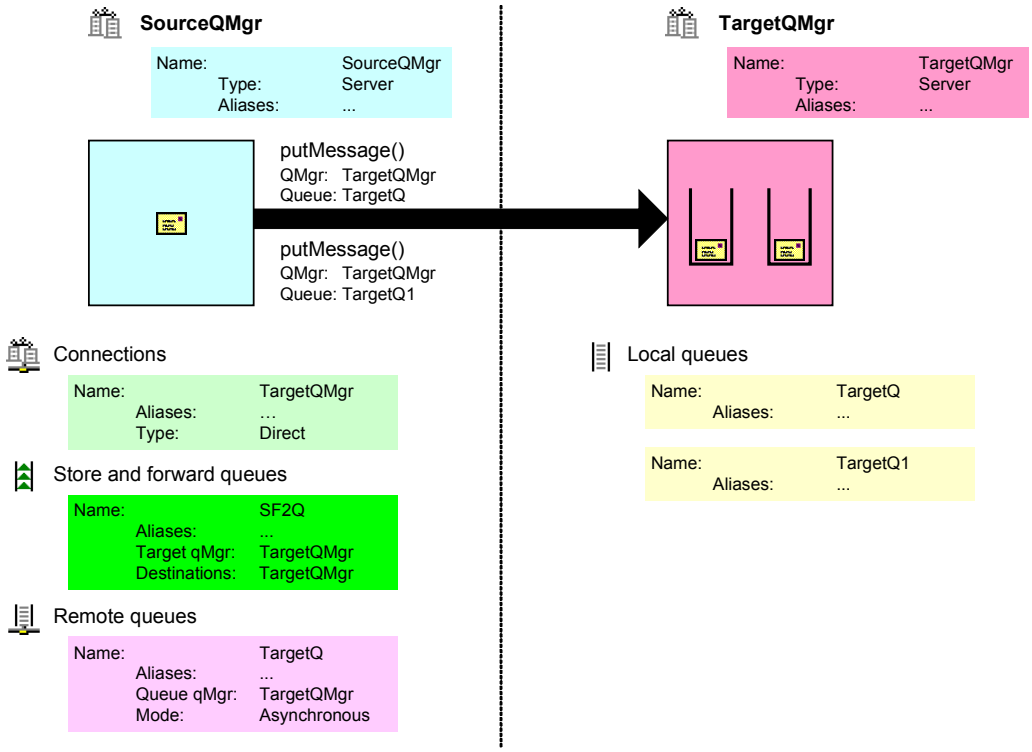

Figure 15-8: Remote and store & forward queues in messaging

In this case the messages put to *TargetQ* will go (asynchronously) from *SourceQMgr*, via the *TargetQ* remote queue on *SourceQMgr*. The messages to TargetQ1 will go (asynchronously) via the store and forward queue *SFQ2*. This configuration illustrates the fact when MQe makes a routing decision between a remote queue definition and a store and forward queue, then the remote queue definition takes priority. If synchronous communication was needed to *TargetQ*, the remote queue definition on *SourceQMgr* must have the mode specified as *synchronous*.

The above configuration is equally applicable to the situations where:

(a) *SourceQMgr* and *TargetQMgr* are both peers.

(b) *SourceQMgr* is a client and *TargetQMgr* is a server.

16 Multi-hop and advanced messaging

In most cases simple point-to-point messaging configurations are not adequate to represent commercial requirements. In this chapter we consider networks of three queue managers, though the principles extend to any number of levels of indirect indirection.

16.1 Synchronous operation

The configuration below is an extension of the example *Figure 15-1: Direct, synchronous messaging* on page 119. The application on *SourceQMgr* is to send messages, as before, to the queue *TargetQ* on the *TargetQMgr*. In this case however, the message is to go via an intermediate queue manager *ViaQMgr*. Both hops are to be synchronous, with the *TCP/IP* protocol used between *SourceQMgr* and *ViaQMgr*, and the UDP protocol between *ViaQMgr* and *TargetQMgr*.

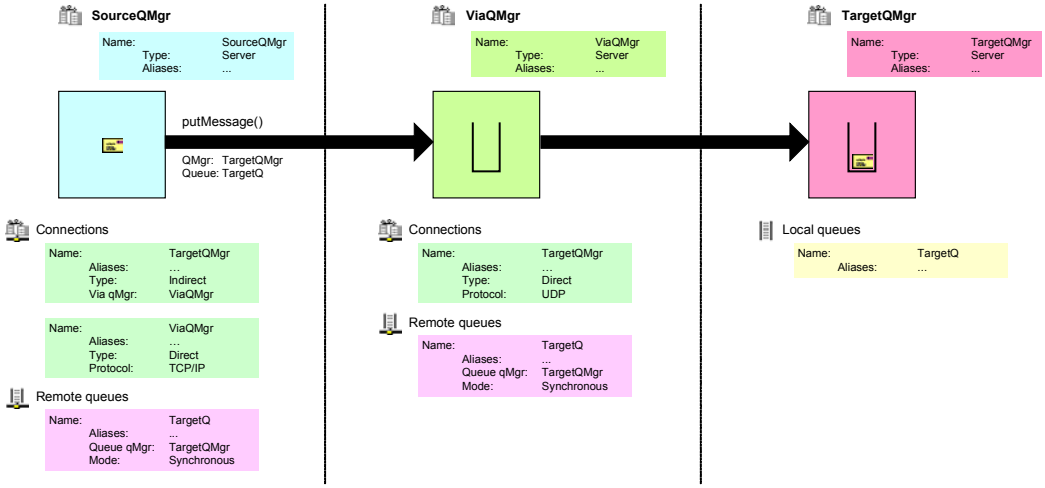

Figure 16-1: Indirect, synchronous messaging

In the diagram above, the key changes on *SourceQMgr* are:

(a) The connection definition for *TargetQMgr* is changed from direct to indirect.

(b) A new connection definition for *ViaQMgr* is added (specifying TCP/IP).

On the *ViaQMgr*:

(a) The connection definition for *TargetQMgr* is direct (specifying UDP).

(b) A synchronous remote queue definition for *TargetQ* on *TargetQMgr*.

No application change is required and exceptions will be received just as described previously. As there are no remote asynchronous queue definitions, or store (and forward) queue definitions involved, the message will never be stored in the network. It will either be delivered to the *TargetQ* or the application will receive an exception.

If the synchronous remote queue definition for *TargetQ* is not defined on *ViaQMgr*, then MQe will automatically add it through queue discovery.

The above configuration is equally applicable to the situations where:

(a) *SourceQMgr*, *ViaQMgr* and *TargetQMgr* are all peers.

(b) *SourceQMgr* is a client; *ViaQmgr* and *TargetQMgr* are servers.

16.2 Asynchronous operation

The only change needed to make transmission asynchronous, with messages awaiting transmission being queued in the remote queue *TargetQ* on *SourceQMgr*, is to change the mode of this queue to *asynchronous*:

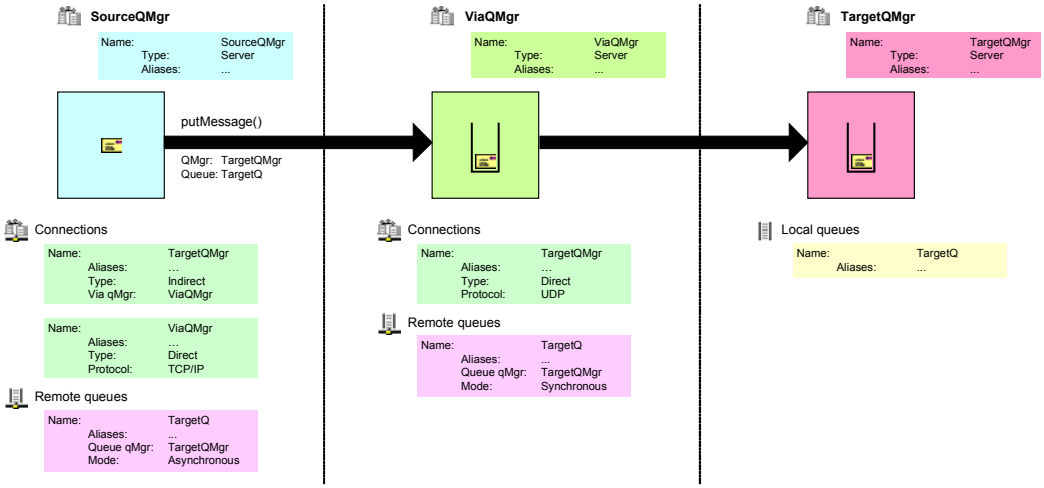

Figure 16-2: Indirect, asynchronous messaging

This exactly parallels the direct case shown in *Figure 15-2: Direct, asynchronous messaging* on page 121.

If the possibility for additional message staging is required on the *ViaQMgr*, then this can be achieved by changing the remote queue definition there to be asynchronous, as shown below:

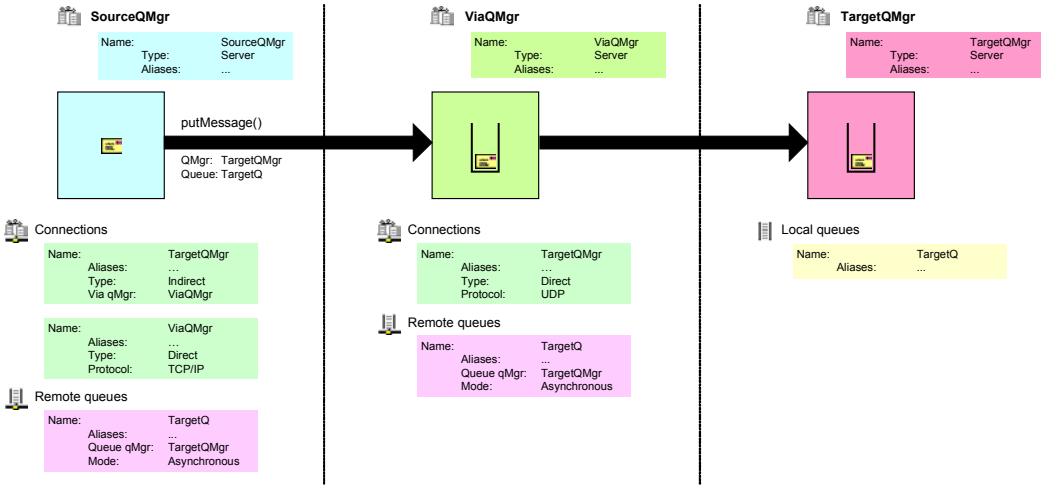

Figure 16-3: Indirect, asynchronous messaging – with staging

If the staging on *ViaQMgr* is to be generic and cover all queues on *TargetQMgr*, then the asynchronous *TargetQ* remote queue definition can be replaced with a store and forward queue that pushes messages to the *TargetQMgr*, as seen in the simpler configuration of *Figure 15-7: Store and forward queues in messaging* on page 126. Similarly, it cannot be configured not to forward them, but keep them instead until collected, as seen previously in *Figure 15-6: Client/server messaging* on page 125.

The above configurations are equally applicable to the situations where:

(c) *SourceQMgr*, *ViaQMgr* and *TargetQMgr* are all peers.

(d) *SourceQMgr* is a client; *ViaQMgr* and *TargetQMgr* are servers.

16.3 Backbone routes

A backbone route is a common network route that is shared by multiple source queue managers sending to multiple targets. Here the configuration is illustrated with six queue managers:

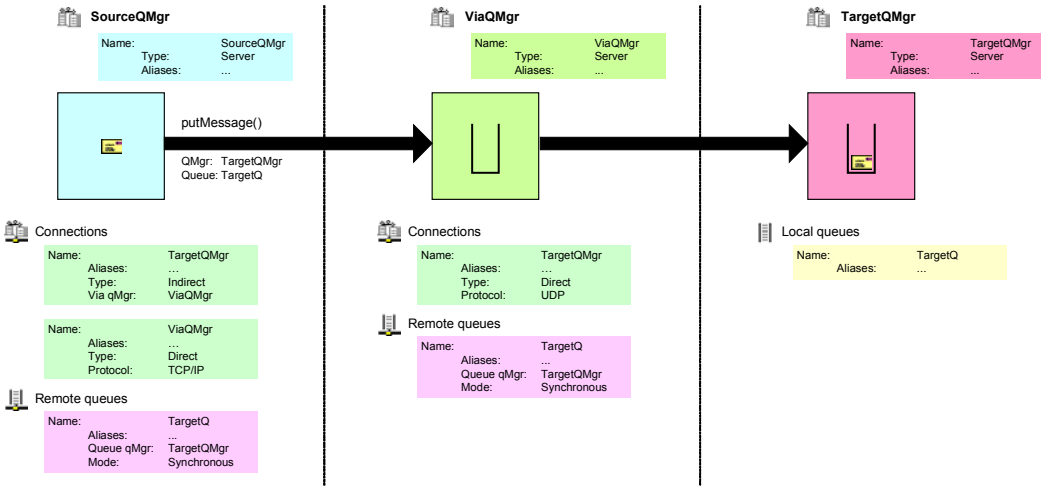

Figure 16-4: Backbone routes

Only a subset of the objects are shown, being just those on the two backbone queue managers *ViaQmgr1* and *ViaQMgr2*, concerned with sending messages from the source queue managers to the targets. In this example a single connection definition on *ViaQMgr1* has been used; this gives a path to *ViaQMgr2* only. All messages arriving on *ViaQMgr1* are collected on a single store and forward queue and sent to *ViaQMgr2*. On *ViaQMgr2*, connection definitions exist to all the targets, and a store and forward queue per target collects the messages.

Many other variations are possible, even assuming messages are to be routed at the queue manager rather than the queue level. For example, on *ViaQMgr1*, there could have been store and forward queues per target queue manager, this breaking up the messages on a per target basis. Alternatively on this queue manager, there could have been indirect connection definitions for all the targets, routing them through *ViaQMgr2*; in this case there would have been no need for a list of target queue managers in the store and forward queue *SFQ*, just the single *ViaQMgr2* queue manager name. In situations where a choice of queue manager-level routing occurs, because both a connection definition and a store and forward queue definition exist for the same destination, the store and forward queue will always be used.

16.4 Alternate routes

Queue manager aliases allow alternates routes to be defined. For example, in the configuration below, there are two different routes defined between *SourceQMgr* and *TargetQMgr*:

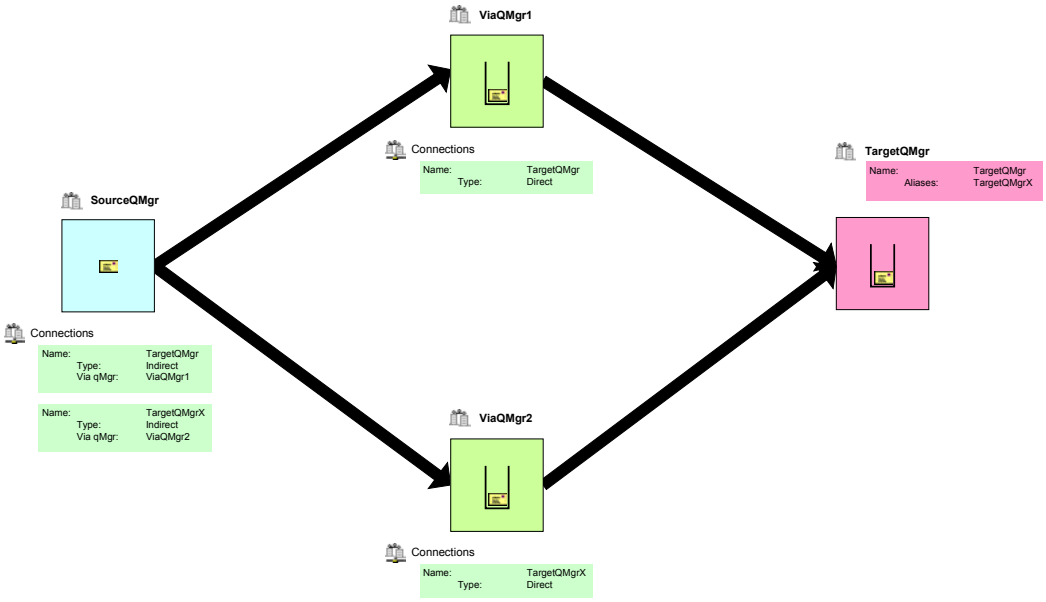

Figure 16-5: Alternate routes

A *putMessage()* on *SourceQMgr* to *TargetQMgr* sends the message via *ViaQMgr1*; a *putMessage()* to *TargetQMgrX* sends the message to the same target but via *ViaQMgr2*.

In more complicated networks, both connection aliases and queue manager aliases can be used in this way.

16.5 Security considerations in routing

In the section *Channel security* on page 65 it was seen that remote queue definitions define the security requirements that must be satisfied by channels moving messages to target queues. The queue manager attribute rule defines the rules for upgrading channels; consequently with a sufficiently flexible rule, multiple security requirements can be met by a single channel.

When a message must be stored on a queue, either en route or at the destination, then the queue attribute rule determines if the channel security meets the requirements of the queue. Note however that there are message transfers that do not involve a channel, e.g. when a home server places a message it has received from a store queue on to its destination queue. In these cases there are no security requirements to be satisfied in the transfer, but the message will be stored in its destination queue in a manner controlled by that queue's security characteristics. Thus to continue this example in more depth, when the home server queue gets the message from the store queue a channel is involved (with characteristics determined by the home server queue and which must be acceptable to the store queue); however, when the home server queue passes the message to the destination queue, there are no channel characteristics to be compared with the destination queue's security characteristics.

In a single hop, message transfer, the security checking is between the source and target queue managers. In multiple hop, asynchronous message transfers, security checking occurs stepwise over each hop.

16.6 Routing rules

In the examples so far we have seen how MQe resolves conflicts between definitions:

1. A remote queue definition takes priority over a store and forward queue definition.
2. A store and forward queue definition takes priority over a connection definition.

17 Certificate management

In the chapter *Registry* on page 19 the concept of WLTS mini-certificates was introduced. These certificates are used to establish identity and are used in conjunction with the *com.ibm.mqe.attributes.MQeWLTSCertAuthenticator*. Mini-certificates can be associated with the queue manager itself, and optionally and additionally, also with individual queues. Mini-certificates are acquired through auto-registration, whereby MQe contacts the mini-certificate server and requests the required certificates. At the queue manager level, auto-registration can take place when the queue manager itself is being configured or started (for more detail see *Auto-registration with the mini-certificate server* on page 27); at the queue level, auto registration can take place when the queue is being created or modified (for more details see *Setting up a private registry for a queue* on page 112).

Certificates are normally issued with a lifetime of 12 months; after this time they must be renewed. Consequently certificates must be periodically examined to determine their expiry dates and renew if necessary.

17.1 Examining mini-certificates

MQe_Explorer allows mini-certificates to be examined; they are shown on the property pages for the relevant authenticatable object. For example, for a queue:

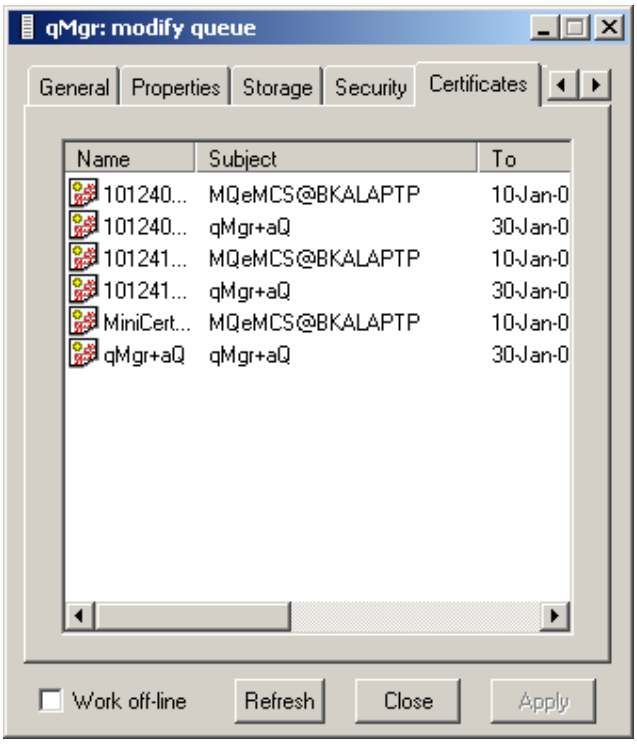

Figure 17-1: Queue mini-certificates

Not all the columns are visible in the figure; the full set of columns is: *Name*, *Subject*, *To*, *From* and *Issuer*. Multiple certificates are shown, a current ones for the queue and the certificate server, plus two sets of replaced (and renamed) certificates.

Queue manager credential examination

The code necessary to access the queue manager certificates is shown below:

```
//get queue manager registry
MQeRegistry registry = qMgr.getRegistry();                          //qMgr registry

if (registry != null)
{
        //get certificates
        MQeFields certs = registry.search(MQeRegistry.MiniCert);
        Enumeration enCerts = certs.fields();
        String certName = null                                      //certificate name
        long beforeTime = 0;                                        //before time (ms)
        long afterTime = 0;                                         //after time (ms)
        String issuer = null;                                       //issuer name
        String subject = null;                                      //subject name
        MQeWTLSCertificate certificate = null;                      //WTLS certificate
        MQeFields cert = null;
        while (enCerts.hasMoreElements())
        {
                //for each certificate
                certName = (String) enCerts.nextElement();
                cert = certs.getFields(certName);
                certificate = new MQeWTLSCertificate(cert.getArrayOfByte("WTLS"));
                beforeTime = certificate.getNotBefore()*1000;
                afterTime = certificate.getNotAfter()*1000;
                issuer = certificate.getIssuerString();
                subject = certificate.getSubjectString();
        }
}
```

Example 17-1: Examining queue manager credentials

The queue manager *registry* object allows access to the registry. The certificates are returned in an *MQeFields* object from the *search()* method on the registry object; enumeration provides access to each certificate in turn, with the name of embedded *MQeFields* object being the name of that certificate. These certificate *MQeFields* objects are each then transformed into a *MQeWTLSCertificate* object, followed by queries to return the properties.

Queue credential examination

The code necessary to access queue certificates is very similar, only the registry access is changed:

```
//get queue registry
String entityName = qMgr.getName() + "+" + queueName;
MQeRegistry registry = qMgr.getRegistry(entityName);                    //queue registry

if (registry != null)
        //get certificates
        MQeFields certs = registry.search(MQeRegistry.MiniCert);
        Enumeration enCerts = certs.fields();
        String certName = null                                          //certificate name
        long beforeTime = 0;                                            //before time (ms)
        long afterTime = 0;                                             //after time (ms)
        String issuer = null;                                           //issuer name
        String subject = null;                                          //subject name
        MQeWTLSCertificate certificate = null;                          //WTLS certificate
        MQeFields cert = null;
        while (enCerts.hasMoreElements())
        {
                //for each certificate
                certName = (String) enCerts.nextElement();
                cert = certs.getFields(certName);
                certificate = new MQeWTLSCertificate(cert.getArrayOfByte("WTLS"));
                beforeTime = certificate.getNotBefore()*1000;
                afterTime = certificate.getNotAfter()*1000;
                issuer = certificate.getIssuerString();
                subject = certificate.getSubjectString();
        }
}
```

Example 17-2: Examining queue credentials

The variable *queueName* holds the name of the queue whose credentials are required.

17.2 Renewing mini-certificates

Mini-certificates can be renewed at any time; it is not necessary to wait until a certificate is close to expiry or has even expired. The standard practice is not to discard the old certificates but to rename them and keep them. If necessary, such retained certificates can be renamed back and brought into use. Another key aspect of the renewal process is that the public and private keys of the new certificate remain the same as the one bring expired, the only significant change therefore being the new expiry date.

Queue manager credential renewal

Renewal of a queue manager credentials is a similar process to that of the original issue; first the mini-certificate server must be authorized to issue a certificate, then the request must be made. Renewal also includes the receipt of an updated version of the mini-certificate server's own certificate.

To authorize renewal, put the certificate server in *admin mode*, enter the name of the authenticatable entity, together with a new certificate request PIN. Then click the *Update* button. Close the server and restart it in *server mode*.

Using MQe_Explorer, start the queue manager and select its icon in either the tree or list view panes. Then click on the *Action→Renew Credentials* menu item; MQe_Explorer will prompt for the new certificate request PIN. The display of queue manager properties will now show four certificates, the two previously present have been renamed, and two new ones have been added.

The equivalent code is shown below:

```
//initialize strings
String thisQMgrName = qMgr.getName();                          //local qMgr name
String regPin = "xxxx";                                        //private registry PIN
String certReqPIN = "yyyy";                                    //new request PIN
String registryAdapter = "com.ibm.mqe.MQeTcpipHttpAdapter";    //cert. server adapter
String registryAddress = "127.0.0.1";                          //cert. server IP
                                                               //      address
String registryPort = "8085";                                  //cert. server port
String prefix = Long.toString(new Date().getTime()) + "_";     //arbitrary prefix

//get [Registry] section
MQeFields regSect = environment.getFields("Registry");         //registry section

//instantiate a private registry configure object
MQePrivateRegistryConfigure regConf = new MQePrivateRegistryConfigure(
        thisQMgrName , regSect, regPIN);

//renew
regConf.renewCertificates(regPIN,
        registryAdapter + ":" + registryAddress + ":" + registryPort,
        certReqPIN, prefix);

//close the configurator
regConf.close();
```

Example 17-3: Renewing queue manager credentials

Care must be taken when using the *environment* variable – if it has ever been passed to MQe, then the passwords will have been removed on return. If needed for credential renewal, a cloned copy should be saved with the contents in tact. The *prefix* variable holds an arbitrary string that is prefixed to the name of the old certificate.

Queue credential renewal

Mini-certificates associated with queues are identical to those associated with queue managers; consequently they expire 12 months after issue and must be renewed. Renewal is a similar process to that of the original issue; first the mini-certificate server must be authorized to issue an updated certificate, then the request must be made.

To authorize renewal, put the certificate server in *admin mode*, enter the name of the authenticatable entity, together with a new certificate request PIN. Then click the *Update* button. Close the server and restart it in *server mode*.

Using MQe_Explorer, start the queue manager and select the relevant queue icon in either the tree or list view panes. Then click on the *Action→Renew Credentials* menu item; MQe_Explorer will prompt for the new certificate request PIN. The subsequent display of queue properties will show that the previous pair of certificates in force have been renamed, and two new ones have been added.

The equivalent code is shown below, this differs significantly to that shown for queue manager credential renewal in Example 17-4:

```
//initialize strings
String thisQMgrName = qMgr.getName();                          //local qMgr name
String thisQName = "mySecureQ";                                //queue name
String regPin = "xxxx";                                        //private registry PIN
String certReqPIN = "yyyy";                                    //new request PIN
String registryAdapter = "com.ibm.mqe.MQeTcpipHttpAdapter";    //cert. server adapter
String registryAddress = "127.0.0.1";                          //cert. server IP
                                                               //      address
String registryPort = "8085";                                  //cert. server port
String prefix = Long.toString(new Date().getTime()) + "_";     //arbitrary prefix

//calculate queue registry PIN
byte[] qRegPINByte = new byte[20];
MQeKey.sha(MQe.asciiToByte(regPIN), 0, regPIN.length(), qRegPINByte, 0);
String qRegPIN = MQe.byteToAscii(qRegPINByte );

//get [Registry] section and update
MQeFields regSect = environment.getFields("Registry");         //registry section
regSect.putAscii("PIN", qRegPIN);

//instantiate a private registry configure object
MQePrivateRegistryConfigure regConf = new MQePrivateRegistryConfigure(
        thisQMgrName + "+" + thisQName , regSect, regPIN);

//renew
regConf.renewCertificates(regPIN,
        registryAdapter + ":" + registryAddress + ":" + registryPort,
        certReqPIN, prefix);

//close the configurator
regConf.close();
```

Example 17-4: Renewing queue credentials

Care must be taken when using the *environment* variable – if it has ever been passed to MQe, then the passwords will have been removed on return. If needed for credential renewal, a cloned copy should be saved with the contents in tact. The *prefix* variable holds an arbitrary string that is prefixed to the name of the old certificate.

The key features of the above code are the calculation of the queue registry PIN from a digest the queue manager registry PIN, and its subsequent storage in the registry section fields object. The entity name to be renewed is now a composite string constructed from the queue manager and queue name.

18 Class requirements

MQe ships as a class library, but not all classes are required in any particular deployment. Where footprint is a consideration, only essential classes should be present. The table below indicates the classes groups associated with a particular function or configuration:

- Mandatory classes
- Queue manager type
- Registry type
- Communications
- Queue types
- Message storage
- Storage adapters
- Message types
- Administration
- Queue manager creation and deletion
- Authenticators
- Compressors
- Cryptors
- Application security services
- Miscellaneous
- Bindings
- Queue manager services
- User-defined MQe extensions

Category	Detail	Classes required
Mandatory classes	For all queue managers	*com.ibm.mqe.MQe* *com.ibm.mqe.MQeAbstractQueueManagerProxy* *com.ibm.mqe.MQeAdapter* *com.ibm.mqe.MQeAttribute* *com.ibm.mqe.MQeAuthenticator* *com.ibm.mqe.MQeCompressor* *com.ibm.mqe.MQeCryptor* *com.ibm.mqe.MQeEnumeration* *com.ibm.mqe.MQeEventLogInterface* *com.ibm.mqe.MQeException* *com.ibm.mqe.MQeField* *com.ibm.mqe.MQeFields* *com.ibm.mqe.MQeKey* *com.ibm.mqe.MQeLoader* *com.ibm.mqe.MQeLocalQueueManagerProxy* *com.ibm.mqe.MQeMessageEvent* *com.ibm.mqe.MQeMessageListenerInterface* *com.ibm.mqe.MQeMsgObject* *com.ibm.mqe.MQeQueueManager* *com.ibm.mqe.MQeQueueManagerRule (or replacement)* *com.ibm.mqe.MQeSecurityInterface* *com.ibm.mqe.MQeTraceInterface* *com.ibm.mqe.registry.MQeRegistry* *com.ibm.mqe.MQeRule* *examples.AttributeRule (or replacement)* *examples.Trace.* (desirable)*

Category	Detail	Classes required
Registry type	One option in this category must be selected	
File registry	Add required: Storage adapter	*com.ibm.mqe.registry.MQeFileSession* *com.ibm.mqe.registry.MQeRegistrySession*
Private registry w/o credentials	Add: File registry	*com.ibm.mqe.registry.MQePrivateRegistry* *com.ibm.mqe.registry.MQePrivateSession*
Private registry with mini certificates credentials	Add: Private registry w/o credentials	*com.ibm.mqe.attributes.MQeMiniCertRequest* *com.ibm.mqe.attributes.MQeSharedKey* *com.ibm.mqe.attributes.MQeWTLSCertificate*

Category	Detail	Classes required
Queue manager type	For all types add required: Administration Storage adapters Message store Authenticators Cryptors Compressors Rules Security	
Standalone qMgr.		
Client qMgr.	Add required: Communications	*com.ibm.mqe.MQeChannel* *com.ibm.mqe.MQeChannelCommandInterface* *com.ibm.mqe.MQeConnectionDefinition* *com.ibm.mqe.MQeRemoteQueueManagerProxy* *com.ibm.mqe.MQeTransporter*
Peer qMgr.	Add: Client qMgr. Add required: Communications	*com.ibm.mqe.PeerChannel* *com.ibm.mqe.PeerChannelReceiver* *com.ibm.mqe.PeerChannelThread*
Server qMgr.	Add: Client qMgr. Add required: Communications	*com.ibm.mqe.ChannelListener* *com.ibm.mqe.ChannelListenerSlave* *com.ibm.mqe.ChannelListenerTimer* *com.ibm.mqe.ChannelManager*
Gateway qMgr.	Add: Server qMgr. Add required Communications Transformers	*com.ibm.mqe.mqbridge.** *examples.trace.** *examples.mqbridge.trace.**

Category	Detail	Classes required
Communications		
TCP/IP w/o history & persistence		*com.ibm.mqe.adapters.MQeTcpipAdapter* *com.ibm.mqe.adapters.MQeTcpipLengthAdapter*
TCP/IP with history & persistence	Add: TCP/IP w/o history and persistence	*com.ibm.mqe.adapters.MQeTcpipHistoryAdapter* *com.ibm.mqe.adapters.MQeTcpipHistoryAdapterElement*
HTTP 1.0 Not to WES Proxy Authentication server		*com.ibm.mqe.adapters.MQeTcpipAdapter* *com.ibm.mqe.adapters.MQeTcpipHttpAdapter*
HTTP To WES Proxy Authentication server		*com.ibm.mqe.adapters.MQeTcpipAdapter* *com.ibm.mqe.adapters.MQeWESAuthenticationAdapter*
UDP		*com.ibm.mqe.adapters.MQeUdpipAdapter* *com.ibm.mqe.adapters.MQeUdpipAdapter$Address* *com.ibm.mqe.adapters.MQeUdpipAdapter$Queue* *com.ibm.mqe.adapters.MQeUdpipAdapter$Queue$Item* *com.ibm.mqe.adapters.MQeUdpipAdapter$State* *com.ibm.mqe.adapters.MQeUdpipDatagramPacket* *com.ibm.mqe.adapters.MQeUdpipDatagramPacket$Id* *com.ibm.mqe.adapters.MQeUdpipDatagramPacket$Queue* *com.ibm.mqe.adapters.MQeUdpipDatagramPacket$Queue$Item* *com.ibm.mqe.adapters.MQeUdpipPacketReader*

Category	Detail	Classes required
Queue types	For all queue types add required: Authenticators Cryptors Compressors Rules	
Local	Add: Storage adapter Message storage	*com.ibm.mqe.MQeAbstractQueueComponent* *com.ibm.mqe.MQeEventTrigger* *com.ibm.mqe.MQeMessageAcceptor* *com.ibm.mqe.MQeQueue* *com.ibm.mqe.MQeQueueRule (or replacement)*
Remote	Add: Local queue (storage adapter & msg. storage only if needed)	*com.ibm.mqe.MQeRemoteQueue* *com.ibm.mqe.MQeMessageDispatcher* *com.ibm.mqe.MQeWriteOnlyFunctionFilter*
Home server	Add: Remote queue (no storage adapter or msg. storage)	*com.ibm.mqe.MQeHomeServerQueue*
Store and forward	Add: Remote queue	*com.ibm.mqe.MQeStoreAndForwardQueue*
Bridge queue	Add: Remote queue	*com.ibm.mqe.mqbridge.MQeMQBridgeAdminMsg* *com.ibm.mqe.mqbridge.MQeBridgeServices* *com.ibm.mqe.mqbridge.MQeMQBridgeQueue* *com.ibm.mqe.mqbridge.MQeMQQMgrName* *com.ibm.mqe.mqbridge.MQeMQQName*

Category	Detail	Classes required
Message storage		
Standard	Default requirements	*com.ibm.mqe.messagestore.MQeMessageStore* *com.ibm.mqe.MQeAbstractMessageStore* *com.ibm.mqe.messagestoreMQeIndexEntryConstants* *com.ibm.mqe.messagestoreMQeIndexEntry* *com.ibm.mqe.messagestore.MQeMessageStore1Enumerator*
Short filename	Only if instructed Add: Standard message store	*com.ibm.mqe.messagestoreMQeShortFilenameMessageStore*
4690-only	4690-specific Add: Short filename message store	*com.ibm.mqe.messagestoreMQe4690ShortFilenameMessageStore*
Message type		
Basic		Support for *com.ibm.mqe.MQeMsgObject* is in Mandatory classes
MQSeries		*com.ibm.mqe.mqemqmessage.**
Storage adapters		
Assured disk	Independence from OS lazy writes	*com.ibm.mqe.adapters.MQeDiskFieldsAdapter*
Non-assured disk	Dependence on OS lazy writes Add: Assured disk	*com.ibm.mqe.adapters.MQeReducedDiskFieldsAdapter*
Memory	Volatile storage	*com.ibm.mqe.adapters.MQeMemoryFieldsAdapter*

Category	Detail	Classes required
Administration		
All targets	Add: Local queue	*com.ibm.mqe.administration.MQeAdminQueueAdminMsg* *com.ibm.mqe.administration.MQeQueueAdminMsg* *com.ibm.mqe.administration.MQeQueueManagerAdminMsg* *com.ibm.mqe.MQeAdministrator* *com.ibm.mqe.MQeAdminMsg* *com.ibm.mqe.MQeAdminQueue* *com.ibm.mqe.MQeAdminQueueTimer*
Target with connection definitions	Add: All targets	*com.ibm.mqe.administration.MQeConnectionAdminMsg*
Target with remote queues	Add: All targets	*com.ibm.mqe.administration.MQeRemoteQueueAdminMsg*
Target with home server queues	Add: Remote queues	*com.ibm.mqe.administration.MQeHomeServerQueueAdminMsg*
Target with store and forward queues	Add: Remote queues	*com.ibm.mqe.administration.MQeStoreAndForwardQueueAdminMsg*
Target with bridge queues	Add: Remote queues	*com.ibm.mqe.mqbridge.MQeMQBridgeQueueAdminMsg* *com.ibm.mqe.mqbridge.MQeCharacteristicLabels*
Target with a bridge to MQSeries	Add: Remote queues	*com.ibm.mqe.mqbridge.**
Queue manager creation and deletion		*com.ibm.mqe.MQeQueueManagerConfigure*

Category	Detail	Classes required
Authenticators		
mini-certificate		*com.ibm.mqe.attributes.DHk (source may be generated)* *com.ibm.mqe.attributes.MQeSharedKey* *com.ibm.mqe.attributes.MQeRandom* *com.ibm.mqe.attributes.MQeWTLSCertificate* *com.ibm.mqe.attributes.MQeWTLSCertAuthenticator*
Compressors		
GZIP		*com.ibm.mqe.attributes.MQeGZIPCompressor*
LZW		*com.ibm.mqe.attributes.MQeLZWCompressor* *com.ibm.mqe.attributes.MQeLZWDictionaryItem*
RLE		*com.ibm.mqe.attributes.MQeRleCompressor*
Cryptors		
triple DES		*com.ibm.mqe.attributes.MQe3DESCryptor*
DES		*com.ibm.mqe.attributes.MQeDESCryptor*
MARS		*com.ibm.mqe.attributes.MQeMARSCryptor*
RC4		*com.ibm.mqe.attributes.MQeRC4Cryptor*
RC6		*com.ibm.mqe.attributes.MQeRC6Cryptor*
XOR		*com.ibm.mqe.attributes.MQeXorCryptor*

Category	Detail	Classes required
Application security services		
Local security	Add required: Cryptors	*com.ibm.mqe.attributes.MQeLocalSecure*
Message-level security	Add required: Cryptors	*com.ibm.mqe.attributes.MQeMAttribute*
Message-level security with digital signature & validation	Add: Public registry. Add required: Cryptors	*com.ibm.mqe.attributes.MQeMTrustAttribute*
Miscellaneous		
Cryptographic support	Application or installation use only	*com.ibm.mqe.attributes.MQeCL*[32] *com.ibm.mqe.attributes.MQeGenDH* (generates a version of *com.ibm.mqe.attributes.MQeDHk.java*)
Mini-certificate server SupportPac ES03		*MQe_MiniCertServer (or command line tool)* *See ES03 installation instructions*
MQe_Explorer SupportPac ES02		*MQe_Explorer* *See ES02 installation instructions*
Public registry	Applicable to types of message-level security Add: Private registry with credentials	*com.ibm.mqe.registry.MQePublicRegistry*
Private registry	Mini-certificate management functions	*com.ibm.mqe.attributes.MQeListCertificates* *com.ibm.mqe.registry.MQePrivateRegistryConfigure*

[32] Requires com.ibm.mqe.attributes.MQeRandom

Category	Detail	Classes required
Queue manager services		
Application run list		*com.ibm.mqe.RunList* *com.ibm.mqe.RunListInterface*
Bindings	Access to Java classes from other languages	
C language		*com.ibm.mqe.bindings.**
User-defined MQe extensions		
		Authenticators Communications adapters Compressors Cryptors Logging classes Message classes Rule classes Security control Storage adapters Trace handler

Figure 18-1: Class requirements

Excepting a number of important classes that are only available as examples (such as *examples.rules.MQeAttribute Rule*) the classes in the *examples.** directories are not listed in the above table.

The table indicates the classes are actually used by MQe, as opposed to the classes that are referenced in the code. There are two considerations that apply:

1. Certain JVMs insist on loading all classes that are referenced, at the time that the first class is loaded. In these cases, additional classes may be required.

2. Certain smart linkers will optionally remove classes that are deemed not to be required; this will happen to all classes that are dynamically loaded by MQe (for example all classes referenced through an MQe class alias are dynamically loaded). Care should be taken to either avoid this option or to explicitly name classes that must be present, even though not apparently referenced in the code.

19 Appendix A: Notices

The following paragraph does not apply in any 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 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.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates.

Any reference to an IBM licensed program or other IBM product in this publication is not intended to state or imply that only IBM's program or other product may be used. Any functionally equivalent program that does not infringe any of the intellectual property rights may be used instead of the IBM product.

Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, New York 10594, USA.

The information contained in this document has not be submitted to any formal IBM test and is distributed AS-IS. The use of the information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While IBM has reviewed each item for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM’s application programming interfaces.

Trademarks

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

AIX

IBM

MQSeries

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

UNIX is a registered trademark of X/Open in the United States and other countries.

Windows and Windows NT are registered trademarks of Microsoft Corporation in the United States and other countries.

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

End of Document