Using MQSeries Everyplace with WebSphere Everyplace Server

Version 1.0

December 2001

Edward McCarthy
Muhammed Omarjee
Juan Rodriquez

IBM Corporation
Research Triangle Park
North Carolina
USA

Take Note!

Before using this report be sure to read the general information under "Notices".

Editor's note:

This material was first published as a chapter in the IBM Redbook: *Enterprise Wireless Applications using IBM WebSphere Everyplace Server Service Provider and Enable Offerings*, SG24-6519. It has been updated by the Editor to reflect the MQSeries Everyplace for Multiplatforms V1.26 product and the associated management tool MQe_Explorer V1.26. Miscellaneous corrections and minor changes are included.

Editor: Barry Aldred
IBM UK Laboratories
Hursley Park
Winchester
Hampshire
UK SO21 2JN

First Edition, December 2001

This edition applies to Version 1.0 of *Using MQSeries Everyplace with Websphere Everyplace Server* 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 report 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
19 December 2001	Version 1.0 (initial release)

SupportPac contents

This SupportPac comprises:

- *Using MQSeries Everyplace with Websphere Everyplace Server* manual (this book)
- Sample code

Preface

This book describes transaction messaging, as implemented by the MQSeries Everyplace component of the WebSphere Everyplace Server. The topics covered are:

- *Overview* on page 1:
  - Provides a description of MQSeries Everyplace and its features, together with a comparison with standard MQSeries
- *Installation and samples* on page 10:
  - Describes installation of MQSeries Everyplace, and how to run one of the supplied examples
- *Chat room – An MQSeries Everyplace application* on page 17, to *Setting up the chat room queue managers* on page 39:
  - Describe a sample application called the *Chat room*, the aim of which is to demonstrate various features of MQSeries Everyplace such as
  - Running the queue manager in client and server mode
  - Running the queue manager as a servlet in WebSphere Application Server
  - Different queue types, such as local and home server
  - Synchronous and asynchronous messaging
  - Encryption of messages
- *Extending the YourCo Application* on page 75:
  - Describes how MQSeries Everyplace can be used to extend access to existing applications
  - Demonstrates a simple customized authentication adapter
- *Integration with WebSphere Everyplace Suite* on page 82:
  - Describes how the Wireless Gateway can be used to support MQSeries Everyplace applications on wireless-type devices

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
- *MQSeries SupportPac ES02: MQSeries Everyplace for Multiplatforms – MQe_Explorer, User Guide*
- IBM Redbook, *Enterprise Wireless Applications using IBM WebSphere Everyplace Server Service Provider and Enable Offerings*, IBM Corporation, SG24-6519
- IBM Redbook, *Programming with Visual Age for Java V3.5*, IBM Corporation, SG24-5264

URLs

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

Download sites

IBM WebSphere (MQSeries SupportPacs):

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

IBM Boulder (MQSeries Everyplace 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. (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 Overview

The purpose of MQSeries Everyplace (MQe) is to provide a once-only assured delivery of messages for applications running on devices with one or more of the following characteristics:

- Typically could not support a fully configured MQSeries queue manager
- Connect via a wireless protocol

The types of devices that would use MQSeries Everyplace are:

- Personal Digital assistants
- Phones
- Sensors
- Laptops

As these sorts of devices are typically being used outside of an organizations intranet, security is an important factor. MQSeries Everyplace provides comprehensive security capabilities to address this aspect.

A detailed introduction to MQSeries Everyplace can be found in the manual, *MQSeries Everyplace Introduction*, GC34-5843. The following sections provide a brief overview of the MQSeries Everyplace functionality. The diagram below gives a high level overview of the MQSeries Everyplace components.

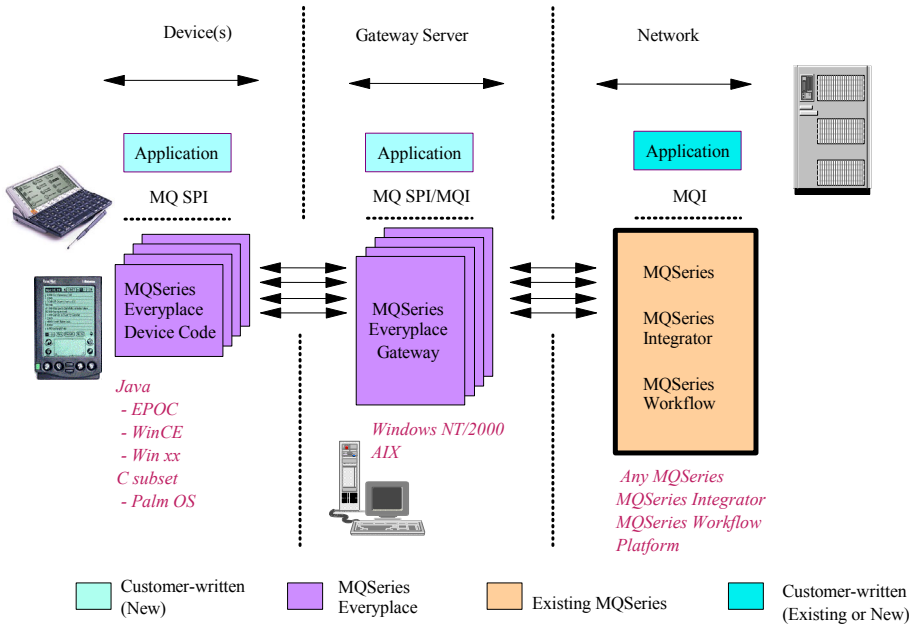

Figure 1-1 MQSeries Everyplace components

1.1 Queue manager comparison

The concept of queue managers is quite different in MQSeries Everyplace compared to the standard MQSeries product.

With standard MQSeries, the typical process is for an MQSeries system administrator to install the MQSeries product, and then create a queue manager, channels and queues. The queue manager is generally started and remains active for an extended period of time, e.g.

weeks or longer. Application developers then use the MQSeries APIs to send and receive messages from the queue manager.

In MQSeries Everyplace however, a set of Java classes and C bindings are provided, which are used by programs to create and control the operation of queue managers. For example, in standard MQSeries, the CRTMQM command is used to create a queue manager, however there is no equivalent command with MQSeries Everyplace. In fact, there are no commands at all.

MQSeries Everyplace queue managers are object-oriented. Essentially, an MQSeries Everyplace queue manager functions as part of the application code. They are active only as long as the application program that activates them is running. Of course, one application can take responsibility for initiating and terminating the queue manager, and then other applications can be concurrently or successively run against this persistent queue manager.

For example, in the case where the Java classes are used to create a queue manager, the queue manager exists as an object in the Java virtual machine (JVM).

1.2 Creating an MQSeries Everyplace queue manager

To simplify the discussion, we will describe the process of creating MQSeries Everyplace queue managers using the supplied Java classes.

Creating an MQSeries Everyplace queue manager typically[1] involves writing a program. The program needs to do the following:

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

Typically an *.ini* file is used to store start-up parameters associated with the queue manager. It contains a number of parameters that describe the queue manager, with probably the two most important being:

- The name of the queue manager
- Details about the registry location used to store definitional information that describes the queue manager

The registry location is a directory on the disk subsystem where MQSeries Everyplace will store queue manager information, such as queue and connection definitions.

Based on the *.ini* file contents, a program uses the appropriate Java classes to firstly create the queue manager and then to start it running when required.

More details about this process can be found in the *MQSeries Everyplace Programming Guide*, SC34-5845.

---

[1] The MQe_Explorer, available in the ES02 SupportPac, will create queue managers without programming; likewise, sample code shipped with MQSeries Everyplace.

1.3 Types of queue manager

All MQSeries Everyplace queue managers are essentially the same, but the functionality used determines their role. The four roles (or types) of MQSeries Everyplace queue managers are:

- Client
- Peer
- Server
- Gateway

Although a particular queue manager normally plays just one role at a time, it is possible for a queue manager to be simultaneously a client, peer, server and gateway, for example.

The process for creating and starting the queue manager is still the same regardless of what type of queue manager is being used. The following diagram shows an overview of the configuration for an MQSeries Everyplace queue manager:

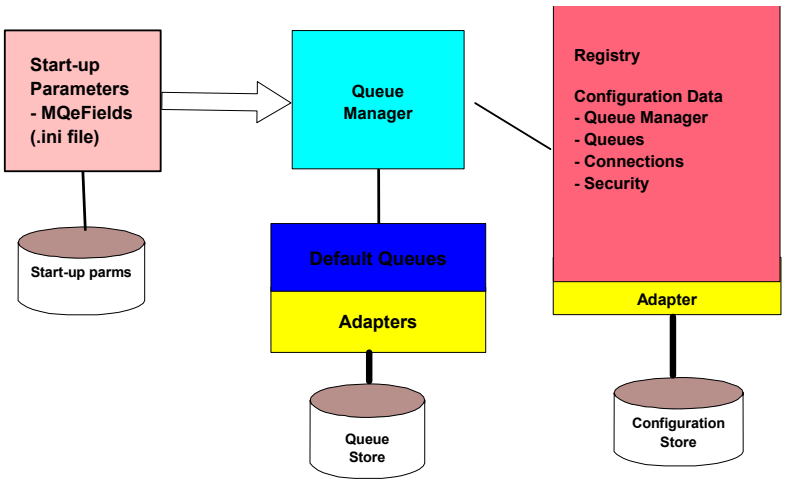

Figure 1-2 Queue manager configuration

Client queue manager

A client queue manager can connect to any number of other MQSeries Everyplace queue managers via a client/server type channel. A client queue manager would typically be active for a short period of time. A characteristic of a client queue manager is that it initiates all data transfers to other queue managers. Clients can connect to multiple servers simultaneously if required.

For example, a salesperson each time they make a sale may start up an application on their laptop. The application starts the client queue manager, and writes information to a queue, and then ends, stopping the queue manager as well. At the end of the day, the salesman dials up the office from home, starts the application again, which starts the client queue manager, which can now connect to a server queue manager and transfer the messages.

Peer queue manager

A peer queue manager is one that can connect to other peer queue managers. Irrespective of who initiated the connection, the peers can subsequently both exploit it to transfer data. Peers use peer-to-peer channels as opposed to the client/server channels used by clients and servers. Client/server channels are able to easily pass through firewalls; peer channels are not.

Server queue manager

A server queue manager is one that typically runs for a long period of time and supports multiple simultaneously attached clients. Additionally, it can connect to any number of server queue managers. Typically such a queue manager would be located at some central location in the organization, and the client queue managers would connect to it.

Gateway queue manager

A gateway queue manager is a server queue manager that has been configured with the ability to use the MQSeries-bridge function. This functionality allows messages to flow between MQSeries Everyplace queue managers and standard MQSeries queue managers.

1.4 Channel types

In MQSeries Everyplace, your program would define a connection to one or more queue managers. When the program tries to send a message, then MQSeries Everyplace will dynamically create a channel to the other queue manager.

MQSeries Everyplace has two types of channels:

- Peer-to-peer
- Client/server

Peer to peer channel

A peer-to-peer channel has the following characteristics:

- Can be established by the queue manager at either end of the channel
- Queue manager at each end can send or receive messages
- A queue manager can have any number of active peer to peer channels to other queue managers
- A queue manager can only have one active peer to peer channel connected to it

Client server channel

A client/server channel has the following characteristics:

- Can be established from the client end of the connection
- Only the queue manager at the client end can send or retrieve messages
- A client queue manager can connect via client server channels to any number of server queue managers

1.5 Adapters

In MQSeries Everyplace, adapters are used to map MQSeries Everyplace components into device interfaces. Certain adapters are also used to control storage of queues into appropriate storage media.

MQeDiskFieldsAdapter

Provides support for reading and writing MQeFields[2] object data and message information to a local 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 do lazy writes to disk.

MQeMemoryFields Adapter

Provides a non-persistent, temporary store for messages, i.e. in memory. Typically 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, this adapter cannot be used for the registry.

MQeReducedDiskFieldsAdapter

Provides support for a high-speed alternative to the MQeDiskFieldsAdapter for writing MQeFields object data and message information to disk. However, it does introduce a dependency on the operating system staying up long enough to empty its buffers on the physical disk subsystem.

MQeTcpipAdapter

Provides support for reading data over TCP/IP streams. This adapter is used as the ancestor object for other adapters and cannot be used directly.

MQeTcpipHttpAdapter

Extends the MQeTcpip adapter to provide basic support for the for the HTTP 1.0 protocol.

MQeTcpipLengthAdapter

Extends the MQeTcpipAdapter to provide a simple, byte efficient protocol.

MQeTcpipHistoryAdapter

Extends the MQeTcpipAdapter to provide a more efficient protocol that caches recently used data. This adapter takes options, such as <PERSIST><HISTORY>.

MQeUdpipAdapter

Provides support for assured data transfer over UDP/IP datagrams.

MQeWesAuthenticationAdapter

Provides support for tunneling HTTP requests through WebSphere Everyplace Authentication and transparent proxies.

1.6 Types of messaging

In standard MQSeries all messaging is asynchronous, in that a message must be committed to a queue, before another process can remove the message.

In MQSeries Everyplace, there are two types of message delivery, asynchronous and synchronous.

---

[2] MQeFields is the base class from which MQSeries message objects are constructed.

Asynchronous Messaging

Asynchronous messaging is similar to standard MQSeries operation. An application on one MQSeries Everyplace queue manager wants to put message to a queue located on some other MQSeries Everyplace queue manager. The local queue manager requires a remote queue definition of the target queue.

The application puts the message to the remote queue, but it is actually stored locally in the local definition of the remote queue. At some undetermined time later, MQSeries Everyplace will deliver that message to the remote queue at the remote queue manager. Actual transmission of the message would occur when the connection between the queue managers became available.

Where these messages for remote queues are stored is controlled by the local definition of that queue on the local queue manager. Different adaptors are available to control were these messages are stored. For example there is an adaptor to have the messages saved to disk, but there is also one to save the message to memory.

Synchronous Messaging

Synchronous messaging is when an application attempts to put a message to a remote queue at a remote queue manager. MQSeries Everyplace will transmit the message only if both the local and target queue managers are online and a connection can be established.

The advantage of synchronous messaging offers is one of performance in that the message is not saved locally, but rather transmitted immediately, and knowing that a message has reached its destination.

1.7 Messages

In standard MQSeries, there are a number of different types of fixed format messages. The main one is the standard message that consists of an MQSeries Message Descriptor and the application message. The message descriptor contains a number of fields used by MQSeries.

In MQSeries Everyplace, messages are message objects.

There is no concept of a message header or body. In MQSeries Everyplace, a message consists of a unique identifier that is generated automatically, plus one or more named field objects.

Each field object consists of:

- A name
- A type indicator, e.g. numeric, character
- A value

The following diagram depicts this concept:

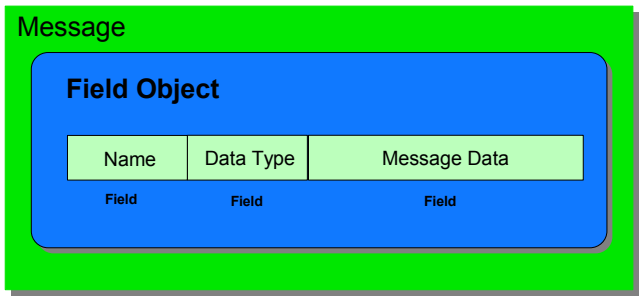

Figure 1-3 MQSeries Everyplace message structure

Each field object is responsible for defining how it is stored and retrieved from a queue. When an application builds a message, it will eventually put the message to a queue, when it does MQSeries Everyplace uses the dump specification on each field object to determine how the value of the field is stored on the queue for transmission. At the end, when an application retrieves the message, MQSeries Everyplace will use the restore specification on each field object to determine how to restore the transmitted data back to its original format.

1.8 Message persistence

There is no concept of persistent and non-persistent messages in MQSeries Everyplace, as essentially every message is treated as persistent. However the strength of this persistence depends on what method is being used to achieve this persistence.

For example you could define that messages are to be persisted to memory, so clearly if the device was turned off the messages are lost. Alternatively you could specify that the messages be written to disk. This means that if the device is powered off, the messages are still there on the disk drive for subsequent retrieval and transmission

MQSeries Everyplace supplies a number of adaptors to handle this persistence, but application programmers can develop their own, for example to persist message data to a database.

1.9 MQSeries Everyplace bridge

The mechanism that allows standard MQSeries and MQSeries Everyplace to exchange messages is referred to as the MQSeries Everyplace Bridge.

This bridging mechanism is in reality a standard MQSeries Java client connection into a standard MQSeries queue manager but used in such a way as to maintain state history of message traffic. This technique results in once-only assured delivery of messages, which is not a built-in feature of MQSeries client connections. Hence messages transmitted between MQSeries Everyplace and MQSeries cannot be lost.

A default transformer is supplied with MQSeries Everyplace, which handles conversion between the standard MQSeries format and the MQSeries Everyplace message structure. However it is possible to develop your own customized transformer.

1.10 Administration

The standard MQSeries product provides administration tools, for example the runmqsc command, which allows you to define queues, channels etc.

MQSeries Everyplace administration is done via specialized administration messages sent to the queue manager. To perform this type of administration requires that the queue manager be defined with the following two queues:

- AdminQ
- AdminReplyQ

To perform this administration requires the writing of an application program. The program needs to build the administration request and then send it to the AdminQ on the target queue manager. When the message is received at the target queue manager the message is processed. The resource that the message is targeted at uses the administration information in the message to action the request.

1.11 SupportPac ES02

As mentioned in the previous sections, creation of the queue manager and associated objects is achieved by writing programs to do this. While not a complex task, it does require some familiarity with MQSeries Everyplace to achieve this.

To get going faster with MQSeries Everyplace, it is strongly recommended to use SupportPac ES02, available at no charge from IBM. This contains the MQe_Explorer program, with full IBM service support for customers with an MQSeries Everyplace deployment license. MQe_Explorer is a Java-based GUI administration tool. It is a set of classes that perform the tasks that have been described above. The GUI interface lets you define queue managers, queues and connections, and perform many other functions related to MQSeries Everyplace.

It can be downloaded from:

http://www.ibm.com/software/mqseries/txppacs/es02.html

1.12 Security

MQSeries Everyplace provides a comprehensive set of security features to protect message data when held locally and when transmitted between queue managers. These features provide authentication, encryption and compression.

Full details about this security are comprehensively covered in the MQSeries Everyplace Introduction, GC34-5843 manual. Briefly however, what is provided is three categories of security:

- Local security, which provides local protection of messages
- Queue based security, which provides protection of messages between queue managers, and
- Message level security, which provides message level protection between initiator and recipient.

The above security features are invoked when a message is stored or retrieved by MQSeries Everyplace.

Local security

Local security can be used by an application to store a message locally in a queue manager, for example to encrypt a message stored on a local queue.

Queue based security

Using queue based security means that the application can leave the issue of security to MQSeries Everyplace. The queues can be defined with attributes that control the type of authentication and encryption used between queue managers.

The only exception is that authentication cannot be done for asynchronous messaging. If authentication is required then message level security must be used.

Message level Security

To use message level security requires the application to set up the message level attribute when putting the message to the queue. There are two supplied attributes that can be used by applications:

- *MQeMAttribute*
- *MQeMTrustAttribute*

MQeMAttribute could be used between queue managers where there is high degree of trust, as it provides a high level of encryption without the use of Public Key Infrastructure technology.

MQeMTrustAttribute provides a more advanced solution involving the use of Public Key Infrastructure. This approach involves the use of digital certificates to authenticate the parties at both ends. As with all certificate based security mechanisms, it is a non-trivial exercise to setup and manage. The documentation in the MQSeries Everyplace does cover this area in depth.

2 Installation and samples

This section provides a brief overview on installation of MQSeries Everyplace and running some of the samples supplied with that product. A detailed description is available in the MQSeries Everyplace Read Me First manual, GC34-5862.

2.1 Install overview

On AIX platforms, the WebSphere Everyplace Server installer can be used to install MQSeries Everyplace.

On Windows platforms, MQSeries Everyplace installation is done by executing a supplied Java *.jar* file. When executed a standard installation process is driven, asking where you want to install the product to etc. For example, accepting all the defaults during installation on a Windows 2000 system would result in the product being installed into the C:\Program Files\MQe directory.

After installation is complete, the directory where MQSeries Everyplace was installed will contain the Java classes and C bindings that can be used by applications.

Version 1.2.1 of MQSeries Everyplace is shipped with WebSphere Everyplace Server. However version 1.2.6 of MQSeries Everyplace can be downloaded from

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

This later version, V1.26 was used during the development of this chapter.

2.2 Supplied samples

A number of sample programs are supplied with the product, which can be used to both verify the installed classes are working, and to provide sample code showing how to use the classes.

Chapter 2 of the MQSeries Everyplace Programming Guide, SC34-5845, provides details of the supplied examples and the different functionality they show.

The simplest example to try as a first step would be to create and then use an MQSeries Everyplace queue manager.

Creating a sample queue manager

The first step is to create an example queue manager. The Windows platform is used in this example, but the process is the similar on a Unix platform.

Firstly open a Command prompt window, and change to the directory where MQSeries Everyplace examples for Windows are installed, in this case to *C:\Program Files\MQe\Java\Demo\Windows*.

To create a sample queue manager type in:

```
CreateExampleQm.bat
```

This batch file uses as input an *.ini* file called *ExamplesMQeServer.ini*. The output produced from running this command looks like this (the output below has had contiguous comments significantly edited and re-formatted to aid readability):

```
C:\Program Files\MQe\Java\demo\Windows>createexampleqm

Create the example queue manager – ExampleQM

This batch file invokes the java class that creates and populates a registry for the example queue
manager. The registry must be populated before a Queue Manager can run. The queue manager
created is determined by entries in queue manager startup parameters. The examples shipped
with MQSeries Everyplace use ini files to hold the parameters. By default
.\ExamplesMQeServer.ini startup parameters file is used.

Parameters
        java environment name (see JavaEnv.bat file for details )

C:\Program Files\MQe\Java\demo\Windows>call JavaEnv

C:\Program Files\MQe\Java\demo\Windows>Set JDK=c:\IBM\jdk1.1.8

C:\Program Files\MQe\Java\demo\Windows>set JavaCmd=java

C:\Program Files\MQe\Java\demo\Windows>Set PATH=c:\IBM\jdk1.1.8\bin;C:\Program
Files\ibm\gsk5\lib;C:\IBM
Connectors\Encina\bin;C:\IBMCON~1\CICS\BIN;C:\WINNT\system32;C:\WINNT;
C:\WINNT\System32\Wbem;C:\IMNnq_NT;C:\Program Files\SQLLIB\BIN;C:\Program
Files\SQLLIB\FUNCTION;C:\Program Files\SQLLIB\SAMPLES\REPL;
C:\Program Files\SQLLIB\HELP;C:\WebSphere\AppServer\bin

C:\Program Files\MQe\Java\demo\Windows>set MQE_BASE_DIR=C:\Program Files\MQe

C:\Program Files\MQe\Java\demo\Windows>set CLASSPATH=C:\Program Files\MQe\java;..\..

C:\Program Files\MQe\Java\demo\Windows>set CLASSPATH=C:\Program
Files\MQe\java;..\..;c:\IBM\jdk1.1.8\lib\classes.zip

C:\Program Files\MQe\Java\demo\Windows>set MQDIR=C:\Program Files\IBM\MQSeries

C:\Program Files\MQe\Java\demo\Windows>if Exist "C:\Program Files\IBM\MQSeries\java\lib"
set CLASSPATH=C:\Program Files\MQe\java;..\..;c:\IBM\jdk1.1.8\lib\classes.zip;C:\Program
Files\IBM\MQSeries\java\lib;C:\Program Files\IBM\MQSeries\java\lib\com.ibm.mq.jar;C:\Program
Files\IBM\MQSeries\java\lib\com.ibm.mqbind.jar;C:\Program
Files\IBM\MQSeries\java\lib\com.ibm.mq.iiop.jar

C:\Program Files\MQe\Java\demo\Windows>if Exist "C:\Program Files\IBM\MQSeries\java\lib"
set PATH=c:\IBM\jdk1.1.8\bin;C:\Program Files\ibm\gsk5\lib;C:\IBM
Connectors\Encina\bin;C:\IBMCON~1\CICS\BIN;C:\WINNT\system32;C:\WINNT;C:\WINNT\Syst
em32\Wbem;C:\IMNnq_NT;C:\Program Files\SQLLIB\BIN;C:\Program
Files\SQLLIB\FUNCTION;C:\Program Files\SQLLIB\SAMPLES\REPL;C:\Program
Files\SQLLIB\HELP;C:\WebSphere\AppServer\bin;C:\Program Files\IBM\MQSeries\java\lib
```

```
C:\Program Files\MQe\Java\demo\Windows>if Exist "C:\Program Files\IBM\MQSeries\bin" set
PATH=c:\IBM\jdk1.1.8\bin;C:\Program Files\ibm\gsk5\lib;C:\IBM
Connectors\Encina\bin;C:\IBMCON~1\CICS\BIN;C:\WINNT\system32;C:\WINNT;C:\WINNT\Syst
em32\Wbem;C:\IMNnq_NT;C:\Program Files\SQLLIB\BIN;C:\Program
Files\SQLLIB\FUNCTION;C:\Program Files\SQLLIB\SAMPLES\REPL;C:\Program
Files\SQLLIB\HELP;C:\WebSphere\AppServer\bin;C:\Program Files\IBM\MQSeries\bin;

C:\Program Files\MQe\Java\demo\Windows>java
examples.install.SimpleCreateQM.\ExamplesMQeServer.ini    .\ExampleQM\Queues\

C:\Program Files\MQe\Java\demo\Windows>
```

Example 2-1 Output from running CreateExampleQm.bat

After this sample completes, a new directory called *ExampleQM* will now be present. This is the location specified to store the registry information and queues used for the sample queue manager. An expanded view of this directory is shown below:

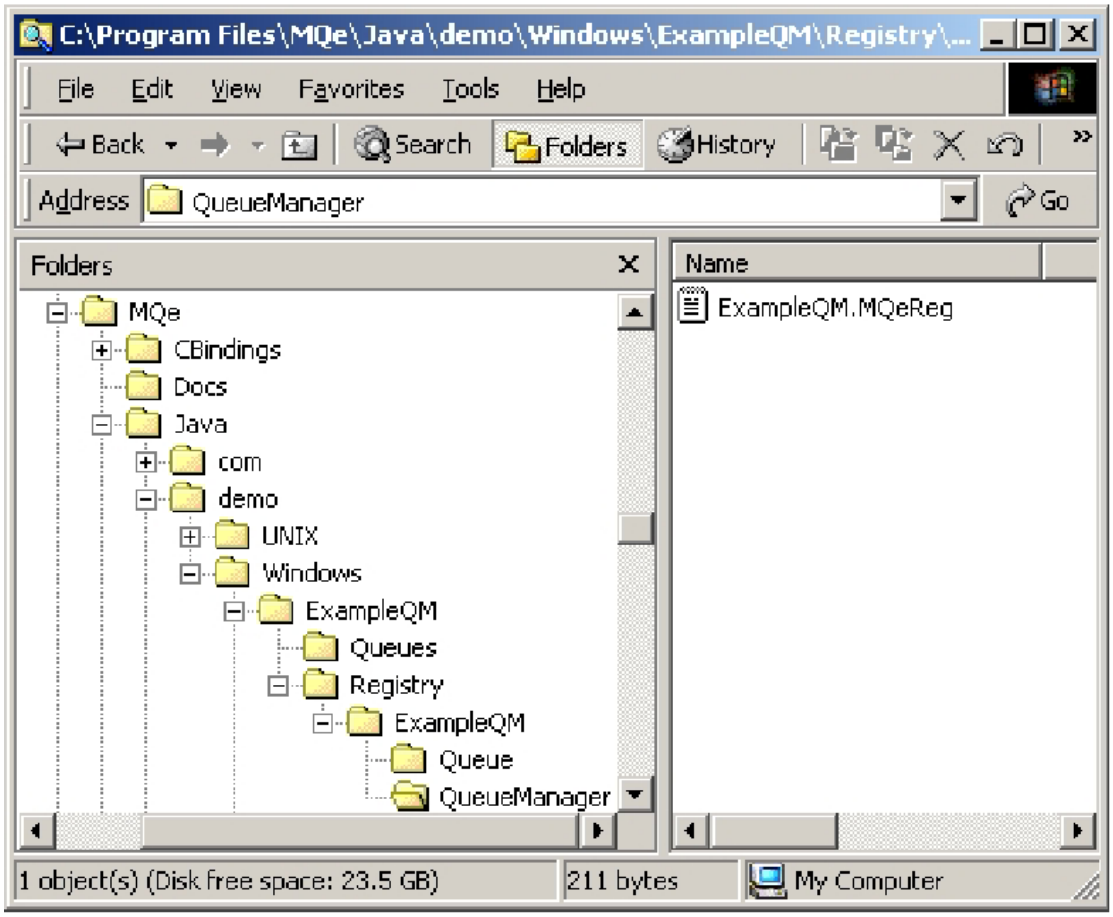

Figure 2-1 Expanded view of a queue manager

Exercising the example queue manager

The next step is to use this example queue manager. From the same Command prompt window type in:

```
ExampleMQeClientTest
```

What the sample does is to write a simple message to the queue *SYSTEM.DEFAULT.LOCAL.QUEUE* and then retrieve it.

The output produced is shown below:

```
C:\Program Files\MQe\Java\demo\Windows>examplesmqeclienttest

C:\Program Files\MQe\Java\demo\Windows>call JavaEnv

C:\Program Files\MQe\Java\demo\Windows>Set JDK=c:\IBM\jdk1.1.8

C:\Program Files\MQe\Java\demo\Windows>set JavaCmd=java

C:\Program Files\MQe\Java\demo\Windows>Set PATH=c:\IBM\jdk1.1.8\bin;C:\Program
Files\ibm\gsk5\lib;C:\IBM
Connectors\Encina\bin;C:\IBMCON~1\CICS\BIN;C:\WINNT\system32;C:\WINNT;C:\WINNT\Syst
em32\Wbem;C:\IMNnq_NT;C:\Program Files\SQLLIB\BIN;C:\Program
Files\SQLLIB\FUNCTION;C:\Program Files\SQLLIB\SAMPLES\REPL;C:\Program
Files\SQLLIB\HELP;C:\WebSphere\AppServer\bin

C:\Program Files\MQe\Java\demo\Windows>set MQE_BASE_DIR=C:\Program Files\MQe

C:\Program Files\MQe\Java\demo\Windows>set CLASSPATH=C:\Program Files\MQe\java;..\..

C:\Program Files\MQe\Java\demo\Windows>set CLASSPATH=C:\Program
Files\MQe\java;..\..;c:\IBM\jdk1.1.8\lib\classes.zip

C:\Program Files\MQe\Java\demo\Windows>set MQDIR=C:\Program Files\IBM\MQSeries

C:\Program Files\MQe\Java\demo\Windows>if Exist "C:\Program Files\IBM\MQSeries\java\lib"
set CLASSPATH=C:\Program Files\MQe\java;..\..;c:\IBM\jdk1.1.8\lib\classes.zip;C:\Program
Files\IBM\MQSeries\java\lib;C:\Program Files\IBM\MQSeries\java\lib\com.ibm.mq.jar;C:\Program
Files\IBM\MQSeries\java\lib\com.ibm.mqbind.jar;C:\Program
Files\IBM\MQSeries\java\lib\com.ibm.mq.iiop.jar

C:\Program Files\MQe\Java\demo\Windows>if Exist "C:\Program Files\IBM\MQSeries\java\lib"
set PATH=c:\IBM\jdk1.1.8\bin;C:\Program Files\ibm\gsk5\lib;C:\IBM
Connectors\Encina\bin;C:\IBMCON~1\CICS\BIN;C:\WINNT\system32;C:\WINNT;C:\WINNT\Syst
em32\Wbem;C:\IMNnq_NT;C:\Program Files\SQLLIB\BIN;C:\Program
Files\SQLLIB\FUNCTION;C:\Program Files\SQLLIB\SAMPLES\REPL;C:\Program
Files\SQLLIB\HELP;C:\WebSphere\AppServer\bin;C:\Program Files\IBM\MQSeries\java\lib

C:\Program Files\MQe\Java\demo\Windows>if Exist "C:\Program Files\IBM\MQSeries\bin" set
PATH=c:\IBM\jdk1.1.8\bin;C:\Program Files\ibm\gsk5\lib;C:\IBM
Connectors\Encina\bin;C:\IBMCON~1\CICS\BIN;C:\WINNT\system32;C:\WINNT;C:\WINNT\Syst
em32\Wbem;C:\IMNnq_NT;C:\Program Files\SQLLIB\BIN;C:\Program
Files\SQLLIB\FUNCTION;C:\Program Files\SQLLIB\SAMPLES\REPL;C:\Program
Files\SQLLIB\HELP;C:\WebSphere\AppServer\bin;C:\Program Files\IBM\MQSeries\bin;

C:\Program Files\MQe\Java\demo\Windows>java examples.application.Example1 ExampleQM
.\ExamplesMQeClient.ini
Example1 Started
..Start a queue manager using ini file: .\ExamplesMQeClient.ini
... nested fields [Registry]
LocalRegType = FileRegistry
DirName = .\ExampleQM\Registry\
Adapter = RegistryAdapter
... nested fields [QueueManager]
Name = ExampleQM
... nested fields [Alias]
QueueManager = com.ibm.mqe.MQeQueueManager
DefaultTransporter = com.ibm.mqe.MQeTransporter
RegistryAdapter = com.ibm.mqe.adapters.MQeDiskFieldsAdapter
Trace = examples.trace.MQeTrace
MsgLog = com.ibm.mqe.adapters.MQeDiskFieldsAdapter
```

```
EventLog = examples.log.LogToDiskFile
PrivateRegistry = com.ibm.mqe.registry.MQePrivateSession
FastNetwork = com.ibm.mqe.adapters.MQeTcpipHistoryAdapter
FileRegistry = com.ibm.mqe.registry.MQeFileSession
ChannelAttrRules = examples.rules.AttributeRule
AttributeKey_2 = com.ibm.mqe.attributes.MQeSharedKey
AttributeKey_1 = com.ibm.mqe.MQeKey
DefaultChannel = com.ibm.mqe.MQeChannel
Network = com.ibm.mqe.adapters.MQeTcpipHttpAdapter

..Started queue manager: ExampleQM
..Create a message and add data:Example1:Humpty dumpty sat on a wall ...
..Put the message to QM/queue: ExampleQM/SYSTEM.DEFAULT.LOCAL.QUEUE
..Get a message from QM/queue: ExampleQM/SYSTEM.DEFAULT.LOCAL.QUEUE
..Message retrieved contains data Example1:Humpty dumpty sat on a wall ...
Example1 Finished
C:\Program Files\MQe\Java\demo\Windows>
```

Example 2-2 Output from running ExampleMQeClientTest

As mentioned in the overview section, the queue manager though defined, only becomes active when an application program activates it.

2.3 Integration with Visual Age for Java

After installing MQSeries Everyplace, in the *<install directory>/java/jars* directory are the following *.jar* files containing the Java classes associated with the product:

- *MQeExamples.jar*
- *MQEHighSecurity.jar*
- *MQeMQBridge.jar*
- *MQeMiniCertificateServer.jar*
- *MQeGateway.jar*
- *MQeDevice.jar*
- *MQeDiagnostics.jar*

To develop programs using Visual Age for Java that use MQSeries Everyplace, the above packages need to be imported. The following outlines the steps to do this:

1. Start VisualAge for Java
2. Create a project to contain the application you plan to develop, for example ITSO WES MQe Example
3. Import the MQSeries Everyplace Java *.jar* files into the project. From the "workbench" window, select file -> import. Select the radio button to indicate that the source to be imported is a *.jar* file. Press *Enter*, and you will then view the display shown below:

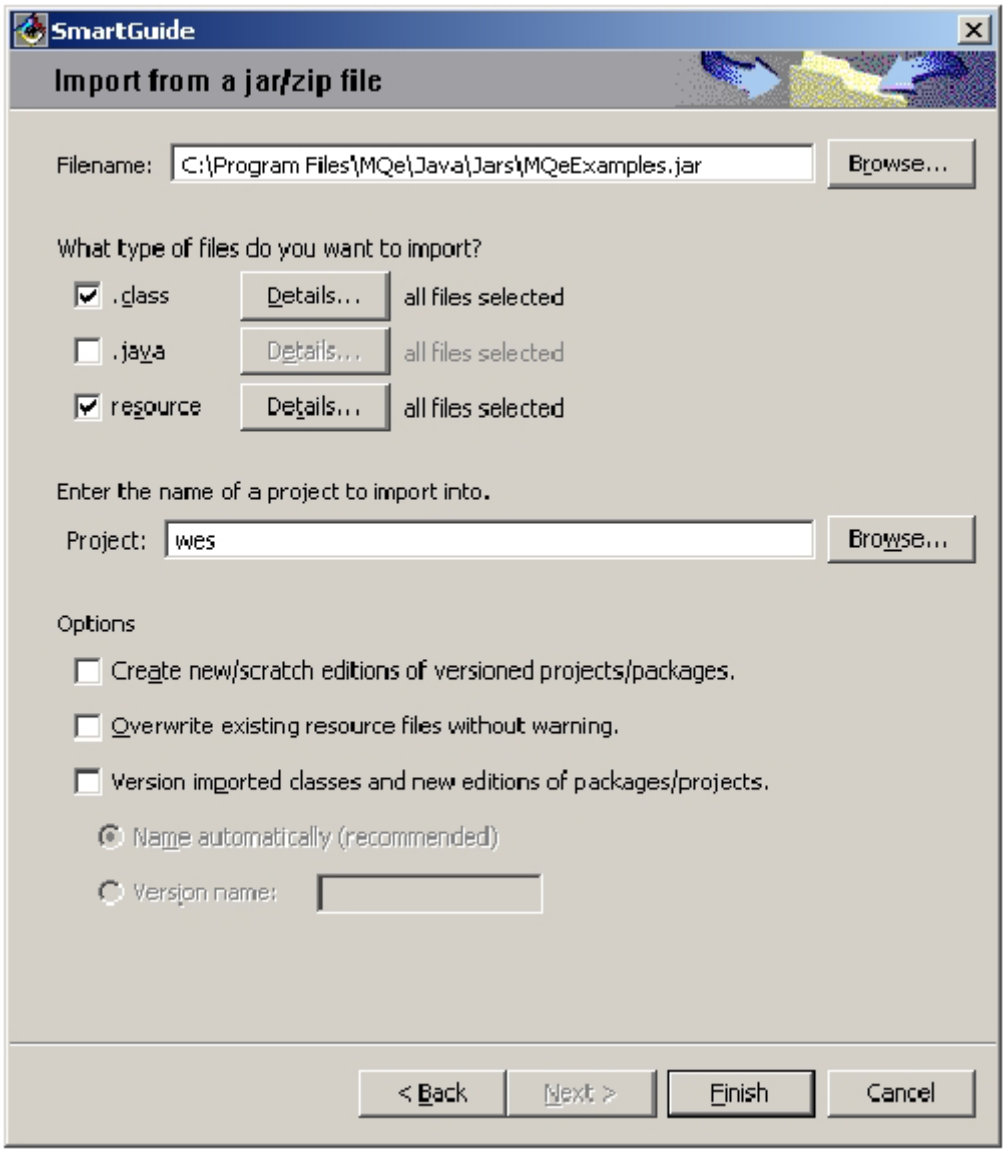

Figure 2-2 Importing .jar files

4. Use the Browse button to locate the *.jar* file to import and then click Finish.

Additional information on using Visual Age for Java to develop applications can be found in the IBM Redbook, *Programming with Visual Age for Java V3.5*, SG24-5264.

2.4 Installation of the transaction messaging samples

A number of transaction messaging samples that complement the text in this SupportPac are supplied in the accompanying zip file. This contains the following:

1. Packages for the chat room application:

   itso.mqe.chatwindow
   itso.mqe.chatclient
   itso.mqe.chatserver

2. Package for running the MQSeries Everyplace queue manager as an application in Websphere

   itso.mqe.was

3. Package for implementing a sample authentication adapter

   itso.mqe.security

4. Package for extending the WebSphere YourCo examples

   WebSphereSamples.YourCo.Timeout

5. Properties and miscellaneous text files

The default unzipping of the file will create an appropriate relative directory structure. In the subsequent text it is assumed that the root directory is C:\ED02. Further detailed installation instructions are given as required, in the sections below:

Chat room application setup on page 61.

Configure WebSphere on page 62.

Set up property files on page 64.

Additional beans on page 80.

3 Chat room – An MQSeries Everyplace application

This section describes a sample application that uses MQSeries Everyplace. The aim of this application is to show how an application can use MQSeries Everyplace. The application shows the use of:

- Server type queue manager
- Client type queue manager
- Queue manager running as a servlet in WebSphere Application Server
- Local queue
- Remote queue
- Store and forward queue
- Synchronous messaging
- Asynchronous messaging
- Use of queue manager and queue aliasing
- Controlling access to a queue using an authority adapter
- Encryption of messages

The Windows platform is used throughout this chapter to describe the application, however the AIX platform can be used if desired.

The application consists of three queue managers. They can all be run on one Windows system, or spread out over three if desired.

3.1 Overview

This sample application is an implementation of a chat room, except that MQSeries Everyplace is used to transfer the "chat" as messages between the two participants. The example has been kept simple, and as such this particular chat room only supports two participants, the server and one client.

The diagram below shows an overview of the application and the queue managers used.

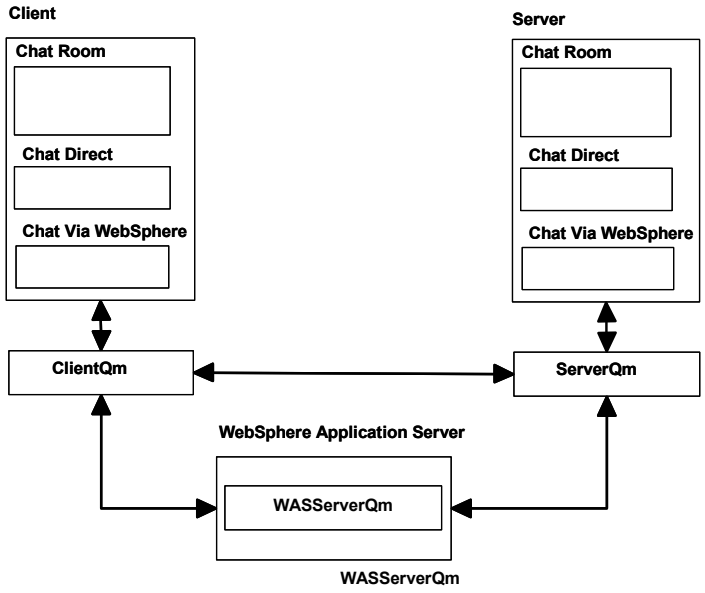

Figure 3-1 Overview of application

Firstly we will describe how the application works, then describe how to setup this application and the programs that support it.

The chat room

This application implements a simple chat room. When run, two Java Swing windows are displayed, the titles of the windows being:

- MQSeries Everyplace Server
- MQSeries Everyplace Client

Each window has an output text box to display chat messages that are sent and received.

Each window has two input text boxes. Text entered into the box labeled *Chat Direct*, is sent direct between the *ClientQm* and *ServerQm* queue managers.

Text entered into the box labeled *Chat Via WebSphere* is also sent between the *ClientQm* and *ServerQm* queue managers, but passes through the *WASServerQm* queue manager running in a servlet in WebSphere Application Server.

Additionally the client window has three extra buttons labeled:

- Trigger Transmission
- Display Admin GUI
- YourCo Secure Query

The Trigger Transmission button is described in *Asynchronous chatting* on page 67.

The Display Admin GUI button is described in *The admin GUI* on page 69.

The YourCoQuery button is described in *YourCo extensions* on page 75.

When first started, the *Chat Direct* input text box in the server window is disabled. Only when the client sends a message does it become enabled.

All that is required to send a message to the other participant is to type in some text in the input text boxes, and then press the *Enter* key.

3.2 The queue managers

Three MQSeries Everyplace queue managers are used for this application.

Client side

The application that operates the client side of the chat room uses a client type queue manager. This is an example of how an application is started, and then starts the queue manager to perform messaging.

In this example the client queue manager is called *ClientQm*.

Server side

The application that operates the server side of the chat room uses a server type MQSeries Everyplace queue manager. This is an example of how the queue manager is initially started, and then an application is loaded after the start-up is complete.

In this example the server queue manager is called *ServerQm*.

WebSphere Application Server

Within WebSphere, a servlet is used to run the queue manager. This queue manager is used to act as an intermediary queue manager between the client and server queue managers, but is also used to allow access to the *YourCo* sample application that comes with the WebSphere Application Server.

In this example the queue manager running in WebSphere is called *WASServerQm*.

Characteristics of the client-side queue manager

The client-side queue manager can establish a connection to any number of queue managers, which in this case will be to *ServerQM* and *WASServerQM*. However no channel listener is configured for this client-side queue manager, thus no other queue manager can initiate a client/server channel connection to it.

This means that applications using this queue manager have two ways of receiving messages:

- Using the *getMessage* API to get a message from a remote queue on some remote queue manager, this requires that a connection to a remote queue manager exists
- Relying on a home-server queue to pull messages from a store and forward queue on a remote queue manager, which the client queue manager will then place in a local queue, from were the application can use the *getMessage* API to retrieve the message

Characteristics of the server-side queue manager

The server-side queue manager has a channel listener configured, so it is able to receive connections from client and server type queue managers. It can also establish connections to other server type queue managers, in this case to *WASServerQm*

Applications using this queue manager cannot directly put a message on to a queue located at a remote client type queue manager.

Characteristics of the queue manager in WebSphere

The queue manager in WebSphere is started during the initialization phase of a servlet that is invoked. This queue manager is a server type queue manager, but has no listener configured. In essence, the HTTP server that receives HTTP requests is the de-facto listener for the queue manager. Connections can be established in both directions between this queue manager and the server-side queue manager.

3.3 Connections

There a number of connections between the various queue managers as follows:

ClientQm to ServerQm

This is a direct channel connection using the default adapter that is the TCP/IP adapter. Messages are sent in IP packets back and forth over this connection between the queue managers.

ClientQM to WASServerQm

This is a direct channel connection using the HTTP adapter. Messages are wrapped in HTTP headers by the adapter code, and then sent to the machine running WebSphere Application Server. In the definition is specified the name of the servlet in WebSphere Application Server to be invoked.

ClientQm to ServerQmViaWas

This is an indirect channel definition. When it is defined, it is configured to first send the messages to the queue manager *WASServerQm*. Configuration information in *WASServerQm* will then be used to determine how the message is then sent on to the queue manager named *ServerQmViaWas*.

ServerQm to WASServerQm

This is a direct channel connection using the HTTP adapter. Messages are sent wrapped in HTTP headers.

WASServerQm to ServerQm

This is a direct channel connection using the TCP/IP adapter.

3.4 Queue discovery

One of the features of MQSeries Everyplace is an ability to perform queue discovery. For example, say that there is an existing definition for a queue called *ABC* on *ServerQm*. If an application running on another queue manager called *ClientQm*, tried to access that queue, the *ClientQm* would detect that is has no local definition for this queue. MQSeries Everyplace requires that the queue managers at each end have a local definition of the queue defined with the same attributes. This comes into play when MQSeries Everyplace is establishing a connection between the queue managers, as many connections can be established, but with different attributes, depending on the queues involved.

When *ClientQm* detects that it does not have a local copy definition of a queue being accessed on a remote queue server, it will query the attributes of the queue defined there, and use those values to define a local definition of the queue.

In this example application however we will define all queue definitions required.

3.5 MQSeries Everyplace queue definitions

The following queue definitions are used in this application. Later sections of this chapter explain how to actually define these queues using the MQ_Explorer tool.

Server-Side queue definitions

Queue: *ChatRoomQ*

Type: Local

Mode: Not applicable

Alias: *ChatRoomQAsync*, *ChatRoomQViaWas*, *ChatRoomQViaAsync*

Purpose: Receives messages from the client side, the application retrieves the messages from this queue and displays them in the output text area on the window. Note the client-side application can send messages to this queue synchronously or asynchronously.

Queue: *ChatSFQ*

Type: Store and forward

Target queue manager: none

Mode: Not applicable

Purpose: Hold messages at the server destined for the client. The messages are subsequently pulled from the server by the client, using the home server queue *ChatSFQ* on the client.

Queue: *ChatClientQViaWas*

Type: Remote

Mode: Synchronous

Alias: None

Target queue manager: *WASServerQm*

Purpose: This queue is used to demonstrate both indirect messaging and the use of WebSphere Application Server to run a queue manager. The aim is that a message typed into the input text area of the window labeled *Chat via WebSphere* will still end up in the *ChatClientQ* on *ClientQm*, but will travel via the queue manager running in WebSphere Application Server. Messages entered into the *Chat via WebSphere* area will be placed onto this queue.

Client-Side queue definitions

Queue: *ChatClientQ*

Type: Local

Mode: Not applicable

Alias: None

Purpose: Receives messages from the server-side, the application retrieves the messages from this queue and displays them in the output text area on the window.

Queue: *ChatRoomQAsync*

Type: Remote

Target queue manager: *ServerQm*

Mode: Asynchronous

Purpose: If the application is unable to synchronously put the message onto the *ChatRoomQ* local queue on *ServerQm*, it will put the message to this queue. It is then the responsibility of MQSeries Everyplace to transfer the message to the target queue manager, when a connection becomes available.

Queue: *ChatRoomQViaWas*

Type: Remote

Target queue manager: *ServerQmViaWas*

Mode: Synchronous

Purpose: Messages entered in the *Chat via WebSphere* box are to be sent to the *ServerQm* via the queue manager in WebSphere Application Server. The application will put the message to this queue. This is used to demonstrate indirect message routing.

Queue: *ChatRoomQAsyncViaWas*

Type: Remote

Target queue manager: *ServerQmViaWas*

Mode: Asynchronous

Purpose: If the application is unable to synchronously put the message onto the *ChatRoomQ* local queue on *ServerQm*, it will put the message to this queue. MQSeries Everyplace will transfer the message to the target queue manager, when a connection becomes available.

Queue: *ChatSFQ*

Type: Home server

Target queue manager: *ServerQm*

Purpose: MQSeries Everyplace polls the corresponding store and forward queue of the same name on the specified target queue manager. When it detects that a message is on that queue on the server, it will pull the message from the server to the client. Once the message is received, MQSeries Everyplace will then place the message into the local queue specified by the application that originally put the message on the queue. Note, that applications cannot access this queue in any way

Queue: *ChatSFQViaWas*

Type: Home server

Target queue manager: *WASServerQm*

Mode: Not applicable

Alias: None

Purpose: As for the *ChatSFQ* queue, MQSeries Everyplace will poll the corresponding store and forward queue of the same name on the *WASServerQm* and pull any messages found there for *ClientQm*.

WebSphere queue definitions

Queue: *ChatClientQViaWas*

Type: Local

Purpose: Temporary store for messages entered into the *Chat via WebSphere* area on the server side.

Queue: *ChatSFQViaWas*

Type: Store and forward

Target queue manager: none

Mode: Not applicable

Destinations: *ClientQm*

Purpose: *ClientQM* will poll this queue, and pull any messages for *ClientQm* to the corresponding home server queue.

All the queues

The following diagram shows all queues used in the Chat Room Application. Note some queues in the diagram are described in later sections of this chapter.

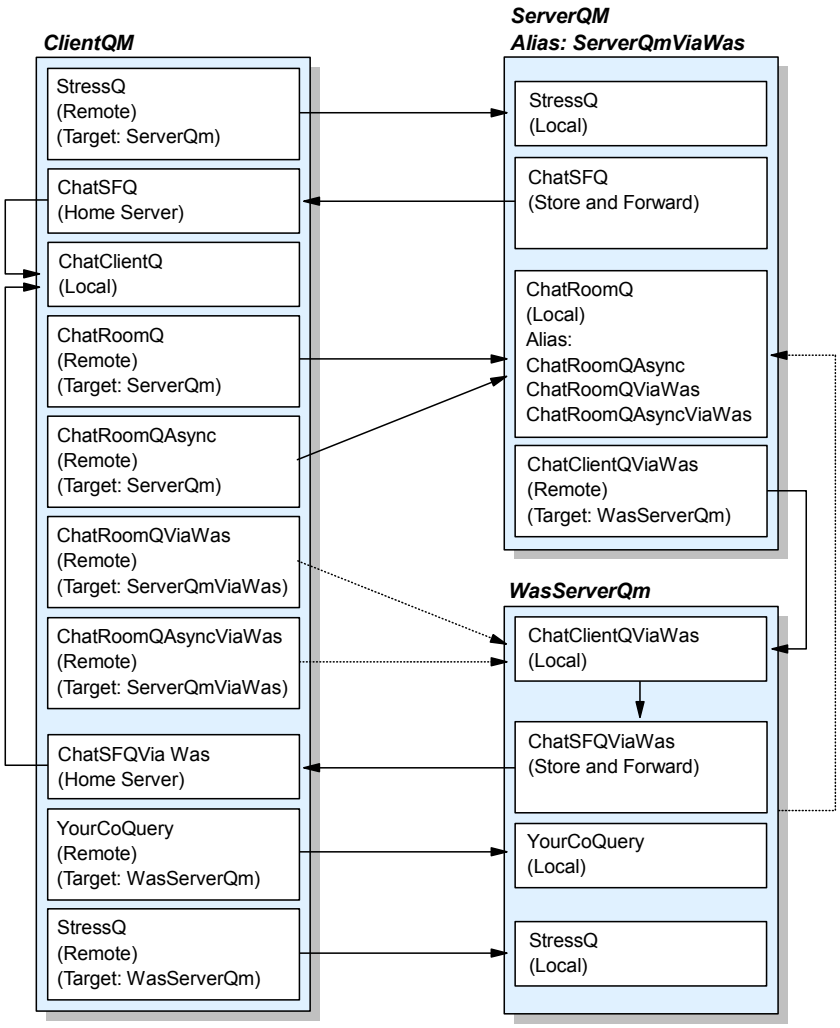

Figure 3-2 Chat room application queues

3.6 The application Java packages

The application is written in Java and consists of these packages.

itso.mqe.chatwindow

This package is used to display the chat room window. It has only one class called RoomWindow. This package is used by the client and server side to display the chat window.

itso.mqe.chatclient

This package is the application used to control the client side of the chat room.

itso.mqe.chatserver

This package is the application used to control the server side of the chat room.

itso.mqe.was

This package contains the code to run the MQSeries Everyplace queue manager as a servlet in WebSphere Application Server.

itso.mqe.security

This package contains the code that implements a sample authentication adapter, explained in *YourCo extensions* on page 75.

3.7 Client side – class interaction

The diagram below shows a high level view of the interaction between the major classes involved on the client side:

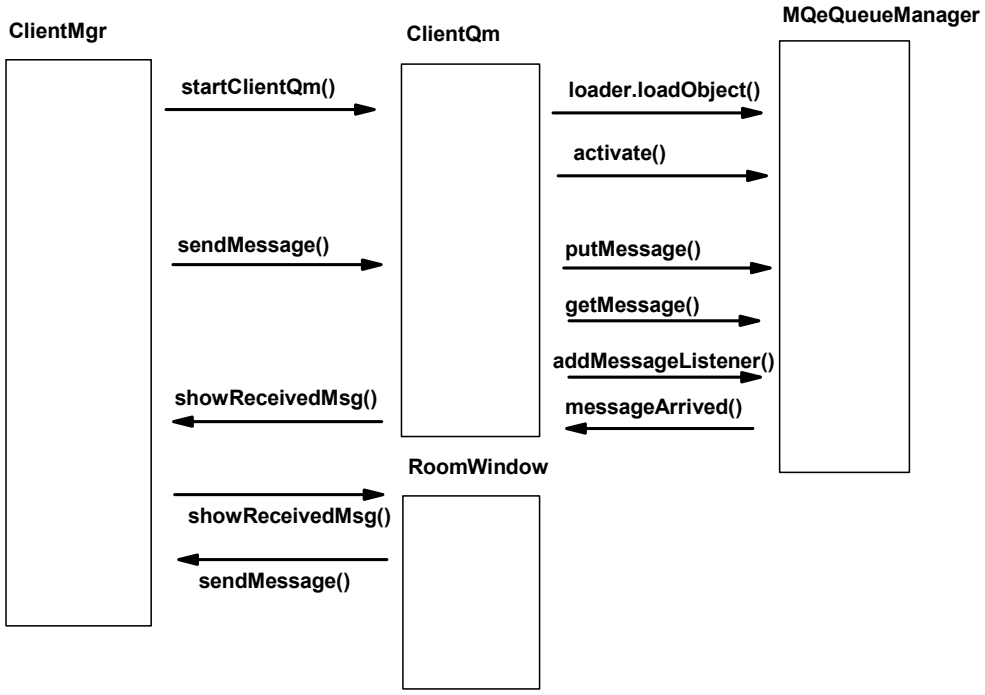

Figure 3-3 Class object interaction - client side

3.8 Server side – class interaction

The diagram below shows a high level view of the interaction between the major classes involved on the server side.

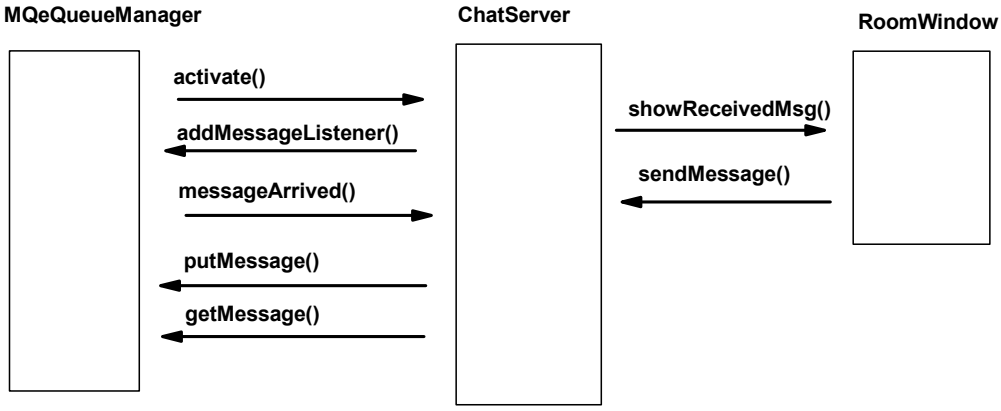

Figure 3-4 Class object interaction - server side

4 Starting a queue manager

In the chat room application, queue managers are used in three different ways; this section explains how this occurs.

4.1 Started by application

The client side of the chat room is an example showing where the application is started and then the application activates the queue manager.

This is the normal approach for client-side type applications, as they do not typically require a queue manager running at all times. Rather an end user will typically want to start the application, have the queue manager started, perform some messaging and then end the application. Such a user does not perhaps require a queue manager running for extended periods of time.

For the chat room application there are three objects involved in the client side as follows:

- *ClientMgr* - the main application
- *ClientQm* - the queue manager
- *RoomWindow* - controls displaying of the GUI chat window

When the application is started, the main method creates a new instance of *RoomMgr*, which results in a *RoomWindow* object being created by the constructor, and then calls the *startChatRoom* method.

The *startChatRoom* method of the *RoomMgr* class starts, creates a *ClientQm* object, and then calls the *startClientQm* method of that class.

The startClientQm method contains the code to start the queue manager. The code below, in *Example 4-1 Reading in the .ini file* to *Example 2-1 Output from running CreateExampleQm.bat*, is all from this method.

Firstly the *.ini* file containing information relating to the queue manager is read in, as shown below:

```
// Access the file
File diskFile = new File(clientIniFile);

// Create a byte array big enough to hold the file's contents.
byte data[] = new byte[(int) diskFile.length()];

// Read the file into the byte array and close the file.
FileInputStream inputFile = new FileInputStream(diskFile);
inputFile.read(data);
inputFile.close();
```

Example 4-1 Reading in the .ini file

Note: The code above illustrates a standard way of performing file input. An alternate approach would be to read an *.ini* file into a fields object, using the *MQeQueueManagerUtils.loadConfigFile*() method.

Once read in, the data is parsed and stored in *MQeField* type objects, as shown in the code below:

```
MQeFields iniSections =
        MQeFields.restoreFromString("\r\n",          // end of record string
        "[#0]",                                      // section pattern
        "(#0)#1=#2",                                 // keyword pattern
        configDataBuff.toString() + "\r\n");
```

Example 4-2 Parsing the .ini file

Then a queue manager object is created, alias definitions from the *.ini* file are processed. Alias definitions are a way of assigning a shorter logical name to class names. These alias definitions can then be used in other sections of the *.ini* file if required. The code to do this is shown below:

```
/* Create queue manager object */

qMgr = new MQeQueueManager();

if (iniSections.contains(Section_Alias)) {
        // Get all the fields inside the alias section
        MQeFields section = iniSections.getFields(Section_Alias);
        Enumeration keys = section.fields();
        while (keys.hasMoreElements()) {

                // For each key, get the value and add the mapping to the MQe
                // internal alias table
                String key = (String) keys.nextElement();
                MQe.alias(key, section.getAscii(key).trim());
                System.out.println("Key: " + key + " Val: " + section.getAscii(key).trim());
        }

        // sectionProcessed(Section_Alias);
  }
```

Example 4-3 Processing the alias entries

Finally now we call the loader method of the *MQe* class to activate the queue manager, as shown in the code below:

```
if (iniSections.contains(Section_QueueManager)) {
        qMgr = (MQeQueueManager) MQe.loader.loadObject(Section_QueueManager);
        if (qMgr != null) {

                // Activate the queue manager.
                qMgr.activate(iniSections);

                // Processing was successful.
        }
  }
```

Example 4-4 Activating the queue manager

The queue manager is now active within the Java virtual machine and can be used by the application.

4.2 Started by the MQe_Explorer

On the server side, MQe_Explorer is used to start the queue manager. A program to start the server-side queue manager, would be similar to the one developed for the client side, except it would need to start some other queue manager functionality such as the channel listener.

Typically the requirement for a server-side queue manager is to be active at all times, so that client type queue managers can connect at any time. Any number of applications may be loaded into the JVM to enable them to perform messaging

MQe_Explorer 1.26 allows you to create and start client, peer, server and gateway queue managers. Properties of the queue manager can be changed, whether they exist in the registry or an *.ini* file. Effectively MQe_Explorer eliminates the need for manual editing of *.ini* files and queue managers can be changed between clients, peers, servers or gateways on an ad hoc basis.

4.3 Started by a servlet

The chat room application demonstrates how a queue manager can be run as a servlet in WebSphere Application Server. The queue manager that runs in WebSphere is a server type queue manager.

Normally a server type queue manager has a listener, which listens on a port to which clients can establish a connection. However a queue manager running in WebSphere cannot start a listener. The HTTP server is effect the listener for the queue manager. Other queue managers access the queue manager in WebSphere by sending the message requests wrapped in HTTP headers.

For the chat room application there are two classes involved in the WebSphere part side as follows:

- *ITSOMQeServlet* - starts the queue manager and handles messages received
- *WasQMgr* - in essence the application, it acts upon on messages

When another queue manager sends a message to the queue manager running in WebSphere, the HTTP headers will specify the name of the servlet to be invoked within WebSphere, in this case ‘ITSOMQeWas’, which is defined to Websphere as a servlet of class *ITSOMQeServlet*. The servlet will start the queue manager the first time it is invoked, this occurs in its init method.

The name of the *.ini* file to use is found by reading a property file called *MQe.properties*. The *MQe.properties* file should be in a directory in the *classpath*. The code to obtain the *.ini* file name is as follows:

```
String mqePropName = "MQe";

try {

        PropertyResourceBundle resourceBundle =
        (PropertyResourceBundle) PropertyResourceBundle.getBundle(mqePropName);
        iniFile = resourceBundle.getString("IniFile");
```

Example 4-5 Obtaining the .ini file when starting in WebSphere

The same code as shown in *Started by application* on page 26 is used to read in the *.ini* file, parse it, load the alias and start the queue manager. The only addition is code to activate a channel manager just prior to activating the queue manager. A channel manager is an object used to handle the communication processes involved between queue managers. The code to activate it is shown below:

```
if ( iniSections.contains( Section_ChannelManager ) ) {

        MQeFields section = iniSections.getFields( Section_ChannelManager );
        channelManager = new MQeChannelManager( );
        channelManager.numberOfChannels( section.getInt( "MaxChannels" ) );

        // sectionProcessed(Section_ChannelManager);
  }
```

Example 4-6 Activating a channel manager

Once the init method completes, the queue manager is active within WebSphere Server.

After the queue manager is started, a *WasQMgr* class object is created, and a reference to the queue manager passed as a parameter. As in the *RoomMgr* class, the *activate* method is called which allows the *WasQMgr* class to save a reference to the queue manager that has been started.

The doPost method

The *doPost* method of the *ITSOMQeServlet* class is worth discussing here. Queue managers sending message to a queue manager in WebSphere Application Server, will wrap the message in a HTTP header, specifying that the HTTP request is a POST to a specified servlet.

In the chat room application this will cause the *doPost* method of the *ITSOMQeServlet* class to be invoked.

The method is effect performing the same sort of role that the listener does for a server type queue manager. All it does is to read the HTTP data received and pass it to the channel manager associated with the queue manager.

The channel manager will then take this data, remove the HTTP headers, and place the message on the queue.

The response from this method call to the channel manager is not a message as such, rather it is just a standard HTTP reply that is to be sent back to the sending queue manager as part of the normal HTTP flow.

The code is show below:

```
ServletInputStream httpIn = request.getInputStream();          // input stream

// Get the request
read( httpIn, httpInData, max_length_of_data);
String mqeInput = new String(httpInData);
System.out.println("MQeInput: " + "mqeInput");
```

```
// Process the request
byte[] httpOutData = channelManager.process(null, httpInData);
String mqeReply = new String(httpOutData);
System.out.println("MQeReply: " + "mqeReply");

// Appears to be an error in that content-length is not being set
// so we will set it here
response.setContentLength(httpOutData.length);
response.setIntHeader("content-length", httpOutData.length);

// Pass back the response
httpOut.write(httpOutData);
```

Example 4-7 Passing HTTP input to the queue manager

5 Starting applications

In the chat room application, applications that use queue managers are used in three different ways; this section explains how this occurs.

5.1 Client side

Starting the application on the client side of the chat room is exactly the same as starting any Java program. The Java command in conjunction with the package and class name is used to start the application.

5.2 Server side – application loading

The server side of the chat room, is an example showing where the application is started after the queue manager has started, in effect the queue manager loads it.

This is the normal approach for server-side applications. Typically the requirement is for the queue manager to be active at all times, so that client type queue managers can connect at any time. Any number of applications may be loaded into the JVM to enable them to perform messaging activities.

For the chat room application there are two objects involved in the server side as follows:

- *RoomMgr* - interacts with the queue manager and the GUI window
- *RoomWindow* - controls displaying of the GUI chat window

Section *Started by the MQe_Explorer* on page 28 described how MQe_Explorer was used to start the queue manager.

In the *.ini* file for the queue manager is specified the application we want to have loaded when the queue manger is started, in this case the chat room application. The lines from the *.ini* file are as follows:

```
[AppRunList]
(ascii)App1=itso.mqe.chatserver.RoomMgr
```

Example 5-1 Application loading

Parameters can also be passed to the application from the *.ini* file as shown below:

```
[App1]
(ascii)ClientQueue=ChatClientQ
(ascii)ChatRoomQ=ChatRoomQ
```

Example 5-2 Passing parameters to applications

A detailed explanation of how applications started this way need to be written starts on page 51 of the MQSeries Everyplace Programming Guide, SC34-5845-04.

Briefly however, the *RoomMgr* class extends the base *MQe* class, and implements these three interfaces:

- *runnable* - to allow it to create a new thread to run on
- *MQeRunListInterface* - to allow queue manager to pass information
- *MQeMessageListenerInterface* - to allow application to notify queue manager of what queues it is interested in

The class that will be started must have a method called *activate*. This will be the first method executed when the application is started. The first thing it does is to save a reference to the queue manager passed as a parameter. This will allow the application to interact with the queue manager. The following line saves the queue manager id:

```
qmgr = (MQeQueueManager) owner;          /* Qmgr is owner of the application */
```

Example 5-3 Saving a queue manager reference

A new thread is then created and started which will cause the run method of the RoomMgr class to be executed. The run method consists of this code:

```
if (itsoChatRoom == null) {

        itsoChatRoom = new RoomWindow(this);
        itsoChatRoom.showChatWindow();
        itsoChatRoom.sendChatMsg();
        itsoChatRoom.setupWasInputListener();
  }
try {
        qmgr.addMessageListener(this, chatRoomQ, null);
```

Example 5-4 The RoomMgr run method

The above code creates a *RoomWindow* object and then calls a method on that object to create the GUI window and display it. Then the code adds a message listener on a specified queue. The purpose of the message listener is explained in *The MqeMessageListenerInterface* on page 34.

The queue manager and application are now both active.

Application data can be passed to the application that is started in this fashion. In the *.ini* file, after the section identifying the applications to be loaded, can be added data to pass to the application. For example, these lines could be added to the *.ini* file:

```
[App1]
(ascii)ClientQueue=ChatClientQ
(ascii)ChatRoomQ=ChatRoomQ
```

Example 5-5 Application-related start-up data

Sample code to access this data in the activate method, is shown below:

```
Enumeration enum = setupData.fields();
try {
        while (enum.hasMoreElements()) {
                String fieldName = (String) enum.nextElement();
                String value = setupData.getAscii(fieldName);
                System.out.println("Field name: " + fieldName +
                " value: " + value);
        }
  }
```

Example 5-6 Accessing application start-up data

5.3 Applications in WebSphere Application Server

Applications that are to run in and access a queue manager in WebSphere Application Server function essentially the same as the applications written for a server.

The chat room application demonstrates this. Text entered into the *Chat Via WebSphere* box on the server side is put to the queue *ChatClientQViaWas* on the *WASServerQm*.

When the init method of the *ITSOMQeServlet* class was executed, it created a *WasQMgr* object, passing it a reference to the queue manager, and also added a message listener for the *ChatClientQViaWas* queue. The *WasQMgr* object is the application.

When a message arrives on this queue, the *messageArrived* method of the *WasQMgr* object is invoked. This method then retrieves the message, and puts the message to the *ChatClientQ* queue on the *ClientQm* queue manager, as shown by the code below:

```
msgObj = wasQMgr.getMessage(null, "ChatClientQViaWas", null, null, 0);
System.out.println("From: " + msgObj.getOriginQMgr() +
        " : " + eventQueueName +
        " msg: " + msgObj.getAscii("Message"));
System.out.println("Relay chat msg to ClientQm : " + msgObj.getAscii("Message"));
replyMsg.putAscii("Message", msgObj.getAscii("Message"));
wasQMgr.putMessage("ClientQm", "ChatClientQ", replyMsg, null, 0);
```

Example 5-7 Processing messages in WebSphere

6 Listening for messages

This section describes how a queue manager notifies an application that a message is available for processing.

6.1 The MqeMessageListenerInterface

In standard MQSeries, is quite common for an application to wait for a message to arrive on a queue. It does this by specifying a WAIT option on the *GET* message API.

In MQSeries Everyplace, the corresponding approach is for an application to implement the *MQeMessageListener* interface.

For example this code from the *run* method of the *RoomMgr* class tells the queue manager that the application wants to be notified whenever a message is put onto the queue specified in the variable *chatRoomQ*:

```
qmgr.addMessageListener(this, chatRoomQ, null);
```

Example 6-1 Adding a message listener

An application can add a listener for as many queues as it requires. The application then needs to have a *messageArrived* method, as this will be the method invoked by the queue manager when a message arrives on any of the queues that a listener has been added for.

The *messageArrived* method is passed a *MessageEvent* object, which contains information about the message that has arrived, such as the queue the message is on and the queue manager it is from.

It is now up to the application to get the message and process it as required. For example, in the case of the chat room application, when a message arrives on the *ChatRoomQ*, the *messageArrived* method in class *RoomMgr* is called, the message is retrieved from the queue and displayed on the GUI window, as shown in the code below:

```
try {
        MQeMsgObject msgObj = qmgr.getMessage(null, chatRoomQ, null, null, 0);
        /* Get the message */
        if (originQMgr == null)  originQMgr = msgObj.getOriginQMgr();

        System.out.println(
                "From: " + eventQMgr + " : " + eventQueueName
                + "Really: " + originQMgr + " msg: "
                + msgObj.getAscii("Message"));

        itsoChatRoom.showReceivedMsg("From: " + originQMgr + " : " +
                msgObj.getAscii("Message"));
```

Example 6-2 Retrieving a message

The same approach is used on the client side and in WebSphere Application Server. In the WebSphere case, the *init* method of the *ITSOMQeServlet* class adds the message listeners, and the *messageArrived* method is implemented in the *WasQMgr* class.

Note there is no comparable notion of triggering an application in MQSeries Everyplace as there is in standard MQSeries; however queue rules can be used to achieve this.

7 Chat room application flows

The section describes what happens when a message in entered into the various text input boxes in the chat windows.

7.1 Chat – client to server – direct

The process that occurs when a message is typed into the *Chat Direct* input box in the window of the client side of the chat room is as follows:

- Message typed into input text area, *Enter* key pressed
- The *sendChatMessage* method in class *itso.mqe.chatwindow.RoomWindow* invoked, echoes message to the output text area, calls *sendMessage* method
- The *sendMessage* method in class *itso.mqe.chatclient.ClientMgr* performs the task of first trying to send the message synchronously, and if that fails, it puts the message to the alternate queue, to have the message sent asynchronously, the code from this method is shown below:

```
try {
        MQeMsgObject msgObj = new MQeMsgObject();
        //String venturing = e.getQueueManagerName();      /* get id of Qmgr msg from */
        //String eventQueueName = e.getQueueName();        /* get queue name */
        try {
                System.out.println("Msg to send: " + message);

                msgObj.putAscii("Message", message);        // set up the message

                System.out.println("Send to: " + targQMgr + " destQ: " +
                        targQ + "doing PUT: " + message);

                /* If the string 'Stress Test' do not appear in the text typed in, then put
                the message to the queue to have it sent synchronously. When the
                'Stress Test' string is found, invoke a method to handle that case */

                if (message.indexOf("Stress Test") < 0)
                        myClientQmgr.putMessage(targQMgr, targQ, msgObj, null, 0);
                else stressTest(viaWasFlag, message);

                System.out.println("Sent to: " + destQMgr + " : " + " msg: " +
                        msgObj.getAscii("Message"));
        } catch (Exception ex) {

                /* If an exception occurs as a result of the put message
                attempt, put the message to the alternate queue to have
                the message sent asynchronously */

                System.out.println("Error sending msg" + ex);
                chatRoomClient.showReceivedMsg(
                        "## Chat room server unavailable"
                        + " will attempt to send message asynchronously ##");
                msgObj.resetMsgUIDFields();

                try {
                        System.out.println(
                                "Async Send to: " + targQMgr + " destQ: "
                                + targQAsync + "doing PUT: " + message);

                        myClientQmgr.putMessage(targQMgr, targQAsync, msgObj, null, 0);

                        chatRoomClient.showReceivedMsg(
```

```
                                        "## Message saved, will be sent Asynchonously ##");

                        } catch (Exception ex2) {
                        chatRoomClient.showReceivedMsg(
                                "## Catastrophic failure, async PUT failed ##");
                        System.out.println("Error doing Async PUT" + ex2);
                }
        }
```

Example 7-1 Putting a message

- In this case, *targQ* is set to a value of 'ChatRoomQ', the method attempts to put the message, synchronously to this queue on the remote queue manager called *ServerQm*
- If the remote queue manager is unavailable, the *put* will fail and an exception is raised, this is caught by the Java code, the application will then put the message to the queue *targQAsync*, which has been set to a value of 'ChatRoomQAsync', this is now an asynchronous message operation, MQSeries Everyplace is now responsible for transferring the message to the remote queue manager when a connection becomes available
- At some time in the future when the connection to the remote queue manager becomes available, the queue manager will send the message, however the application put the message to a queue called *ChatRoomQAsync*, but there is no queue called this on the remote queue manager, but the definition for the *ChatRoomQ* queue specifies that it has an alias of *ChatRoomQAsync*, this means that the message is placed in the *ChatRoomQ* on the remote queue manager

7.2 Chat – client to server – via WebSphere

The process that occurs when a message is typed into the *Chat via WebSphere* input box in the window of the client side of the chat room is as follows:

- Message typed into input text area, *Enter* key pressed
- *sendChatMessage* method in class *itso.mqe.chatwindow.RoomWindow* invoked, as it was for the case where text was entered into the *Chat Direct* box, echoes message to the output text area, calls *sendMessage* method
- The *sendMessage* method in class *itso.mqe.chatclient.ClientMgr* performs the task of first trying to send the message synchronously, and if that fails, it puts the message to the alternate queue, to have the message sent asynchronously
- The code executed, shown in *Example 7-1 Putting a message* on page 36 is the same as it was for the *Chat Direct* case

- The difference is that the target queue manager is set to `ServerQmViaWas' and *targQ* and *targQAsync* are set to different values, this is done by this code in the *sendMessage* method:

```
/* The action listener that is invoked when the Enter key is
   pressed in a text box, sets the value of viaWasFlag when it calls
   this method.

   The value is null if called from the listener for the 'Chat Direct'
   box, and not null if called from the listener for the
   'Chat via WebSphere' box */

if (viaWasFlag == null) {
            targQ = destQueue;
            targQMgr = destQMgr;
            targQAsync = destQueueAsync;
} else {
            targQ = destQueueViaWas;
            targQMgr = destQMgrViaWas;
            targQAsync = destQueueAsyncViaWas;
}
```

Example 7-2 Setting the value of the target queue

As in the direct case, the application will try to put the message synchronously to the queue *ChatRoomQViaWas* on *ServerQmViaWas*. Recall however that, *ChatRoomQViaWas* has been defined as an alias for the *ChatRoomQ*, and that *ServerQmViaWas* is an alias for the *ServerQm* queue manager. In effect this will become a *put* to the *ChatRoomQ* on the *ServerQm*. This means the put message request only succeeds if there is a connection right through to the *ServerQm*.

If the message cannot be put synchronously, it is put asynchronously, being put to the *ChatRoomQAsyncViaWas* queue, for transmission by the queue manager when a connection via *WASServerQm* to *ServerQm* is available.

7.3 Chat – server to client – direct

The process that occurs when a message is typed into the *Chat Direct* input box in the window of the server side of the chat room is as follows:

- Message typed into input text area, *Enter* key pressed
- *sendChatMessage* method in class *itso.mqe.chatwindow.RoomWindow* invoked, echoes message to the output text area, calls *sendMessage* method
- Note the *RoomWindow* class is used at both the client and server ends to display the chat window
- The *RoomMgr* class has its own *sendMessage* method, but it is essentially the same as the one in the *ClientMgr* class, the application will put the message to the *ChatClientQ* on the *ClientQm* queue manager,
- The major difference here is that the server only sends its messages asynchronously
- When the put message is executed, the queue manager will store the message on the *ChatSFQ* queue, this is because the server-side queue manager cannot send a message to the client-side queue manager, the queue manager will store any messages for *ClientQm* on this store and forward queue

- On the client side, the *ClientQm* has a corresponding home server queue called *ChatSFQ*. The queue is configured to poll the corresponding store and forward queue on the *ServerQm* every five seconds; when it detects a message on that queue it will pull the message and place it in the home server queue, in this case *ChatSFQ*
- The queue manager will then move the message to the target queue specified by the application, in this case *ChatClientQ'*
- The *messageArrived* method in the *ClientMgr* object will be invoked by the queue manager, it will retrieve the message and display it in the output area of the GUI window

7.4 Chat – Server to client – via WebSphere

The process that occurs when a message is typed into the *Chat via WebSphere* input box in the window of the server side of the chat room is as follows:

- Message typed into input text area, *Enter* key pressed
- *sendChatMessage* method in class *itso.mqe.chatwindow.RoomWindow* invoked, echoes message to the output text area, calls *sendMessage* method, but passes a flag to indicate that the message is to be sent via WebSphere
- The *sendMessage* method puts the message to a queue called *ChatClientQViaWas* on the *WASServerQm*, a synchronous operation
- Note this message is sent as a HTTP request to WebSphere, where the *ITSOMQeServlet* class will be invoked to handle this POST request
- The *messageArrived* method of the *WasQMgr* object is invoked by the *WASServerQm*, the message is retrieved from the *ChatClientQViaWas* and put to the *ChatClientQ* on the *ClientQm* queue manager, as shown in *Example 5-7 Processing messages in WebSphere* on page 33.
- When the *putMessage* is executed, the queue manager will store the message on the *ChatSFQViaWas* queue, this is because the server-side queue manager cannot send a message to the client-side queue manager, the queue manager will store any messages for *ClientQm* on this store and forward queue
- On the client side, the *ClientQm* has a corresponding home server queue called *ChatSFQViaWas*, the queue is configured to poll the corresponding store and forward queue on the *ServerQm* every five seconds, when it detects a message on that queue it will pull the message and place it in the home server queue, in this case *ChatSFQViaWas*
- The queue manager will then move the message to the target queue specified by the application, in this case *ChatClientQ*
- The *messageArrived* method in the *ClientMgr* object will be invoked by the queue manager, it will retrieve the message and display it in the output area of the GUI window

Note, that messages are traveling over two connections using two different protocols. The messages between *ClientQM* and *WASServerQM* use HTTP, but TCP/IP is used between *ServerQM* and *WASServerQm*.

8 Setting up the chat room queue managers

This section describes how to set up the three queue managers used in the chat room application.

Note that the queue managers can be setup on either one system, on two or on three systems.

8.1 Preparing for setup

Creation and configuration of the queue managers is done using the MQe_Explorer tool (SupportPac ES02).

If the default install process is followed, then the product is installed into the C:*program files**MQe* directory.

To run the MQe_Explorer, either add *C:\Program Files\MQe\Java* to the environment variable *classpath* in the *System* properties of the *Control panel* folder, or in a DOS window type in the following:

```
SET CLASSPATH=C:\Program Files\MQe\Java;%CLASSPATH%
```

Start a DOS window and change directory to where SupportPac ES02 was installed. Let us assume it has been installed to *C:\Program Files\MQe\Java\MQe_Explorer*.

Then to start the MQe_Explorer, type in:

```
MQe_Explorer.exe
```

The following window appears:

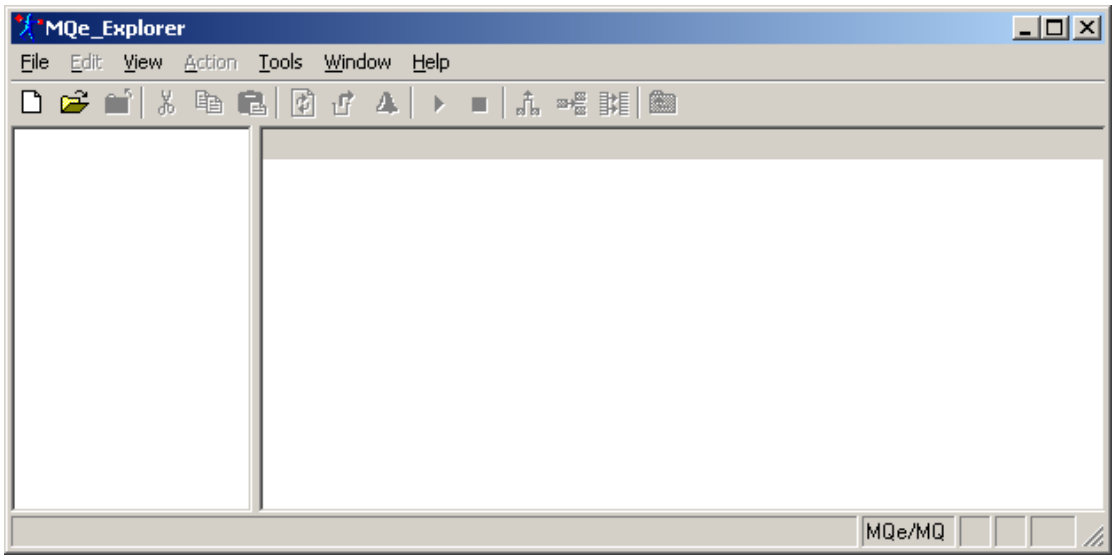

Figure 8-1 Initial MQe_Explorer window

8.2 Creating ServerQm queue manager

From the initial MQe_Explorer window, select File->New->QueueManager. A window will appear in which you will define the attributes of the server queue manager. There are several tabs, the initial one displayed is labeled ‘General’. We will call this queue manager ‘ServerQm’ and define it as being a server-type queue manager. To do this, follow these steps:

Type ‘ServerQm’ into the field *QMgr. name*

Make sure the box next to the label *Server* is ticked.

The window should look similar to this:

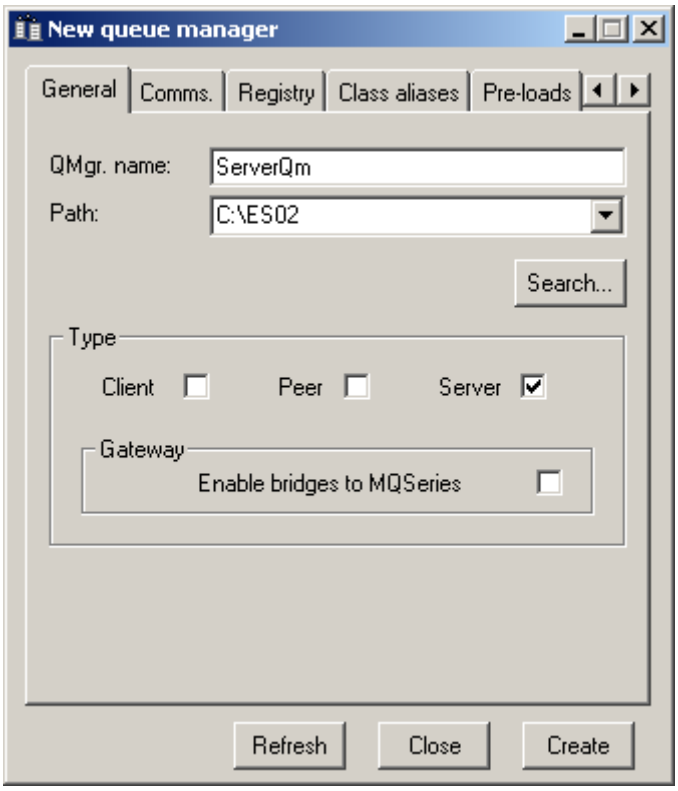

Figure 8-2 Setting the name and type of ServerQm

Select the *Comms* tab which controls the settings related to incoming communications

The IP address of the machine is displayed in the field *IP address*; the *port* that this queue manager will listen on for incoming channel requests is set to the default value used by MQSeries Everyplace, which is 8082.

The *adapter* property is concerned with the protocol that the server queue manager will expect on incoming connections. In this case, use the ‘(default)’ adapter, which is mapped to:

com.ibm.mqe.adapters.MQeTcpipHistoryAdapter

or set it explicitly. This adapter uses a TCP/IP based protocol and keeps track of the history of data already sent in order to reduce network traffic. Later on when you define connections to this server queue manage, you will need to specify the type of adapter used to connect. It must match the value you specify here.

The *channel* property determines the nature of the flows between queue managers, for example whether client/server flows or peer-to-peer flows are used. Set the value to ‘(default)’ and MQe_Explorer will choose the channel type:

com.ibm.mqe.MQeChannel

which is compatible with the chosen type of queue manager (in this case a server).

All other properties should be set to their default values.

The window will look similar to this:

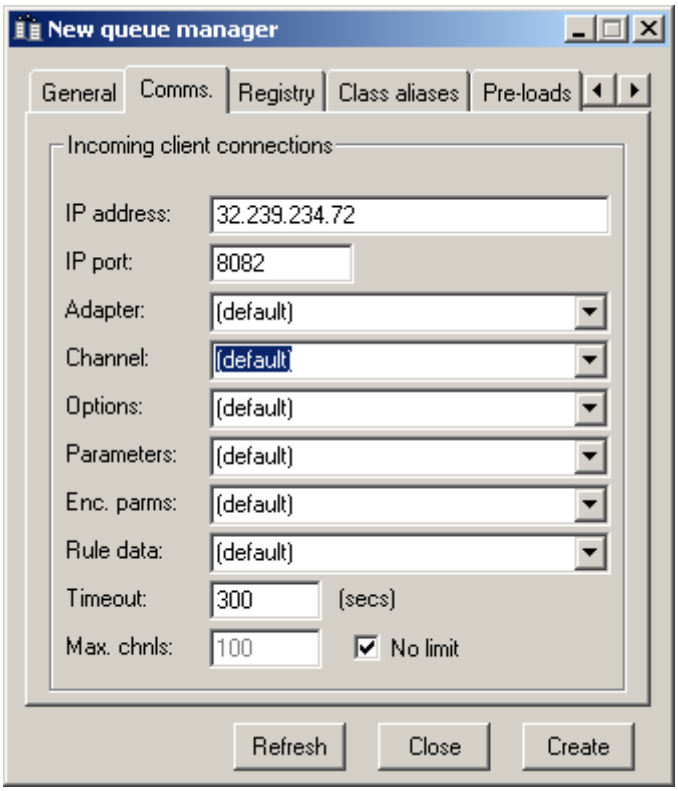

Figure 8-3 Setting the incoming communications parameters of ServerQm

Click the *Create* button to now create the server-side queue manager. Note there are many other tabs that are not explained here, as all the defaults are taken. A window confirming creation of the queue manager appears, similar to that shown below. Make a note of the *.ini* file name – it will be need when the queue manager is re-started:

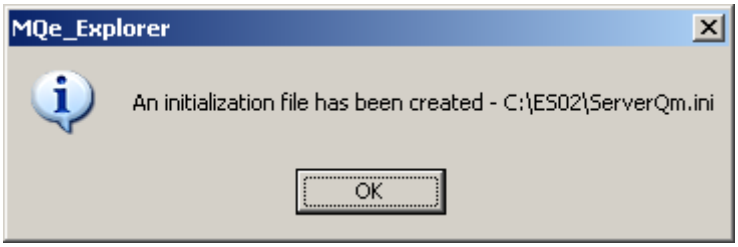

Figure 8-4 ServerQm creation confirmation

The queue manager has now been created, and is in fact running. The MQe_Explorer window will now have an object called *MQe root* with a plus sign beside it. Click on the plus sign to expand the object tree under *MQe root*, and you will see an object for the *ServerQM* you have just created. Continue to expand the objects under *ServerQm* and you will see the default queues that have been set up. The display will be similar to this:

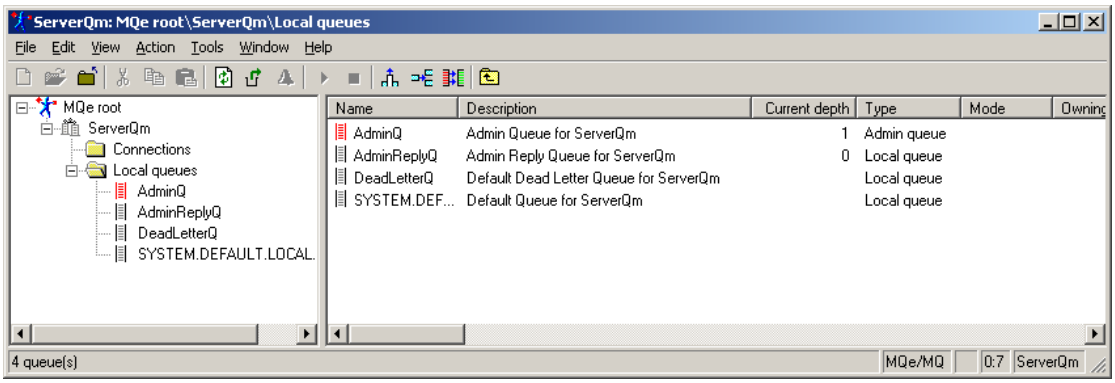

Figure 8-5 Expanded tree view of ServerQm

Note: In future, to start this queue manager if it is not running, open the *.ini* file. Start MQe_Explorer by double clicking its icon (or by running MQe_Explorer.exe from a DOS prompt), then select File->Open, locate the *ServerQm.ini* file and select it. The queue manager will then be started.

Adding an alias to ServerQm

An alias of *ServerQmViaWas* for *ServerQm* is used in this application. An alias can only be added once the queue manager is defined. However, it is possible for aliases to be changed thereafter.

Right click on the *ServerQm* object in the tree, then *Properties*, a window appears, then click on the *Aliases* tab. Type in the alias name *ServerQmViaWas*, and click the *Add* button. The window should now look something like this:

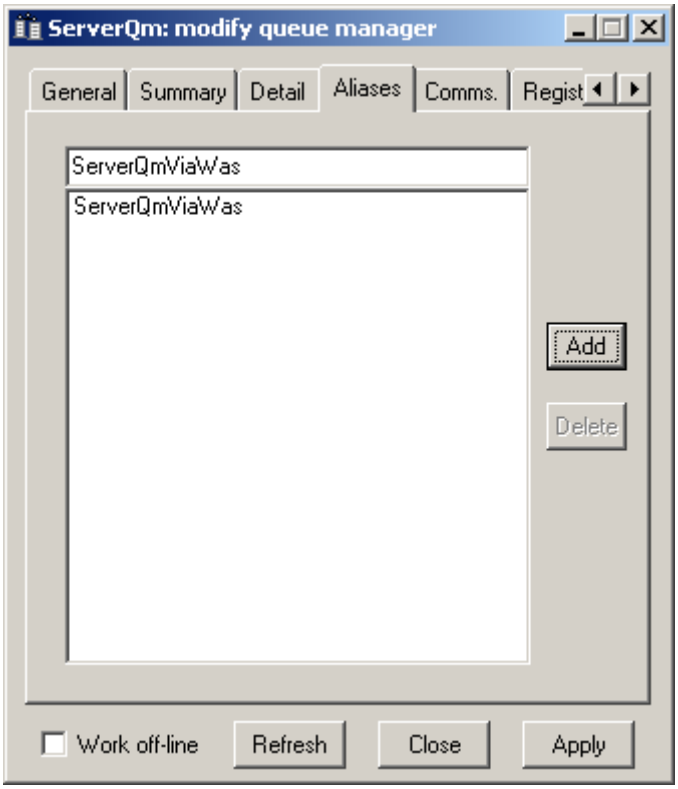

Figure 8-6 Defining a queue manager alias

Click the *Apply* button to make the change. A message indicating that a local connection has been created will be displayed.[3]

8.3 Creating ClientQm queue manager

The process for defining the client-side queue manager is very similar to the server-side queue manager. Start another MQe_Explorer session from the DOS prompt, and from the initial MQe_Explorer window, select File->New->Queue Manager. Enter the name of the client queue manager as 'ClientQm' in the field *QMgr. name*.

Click on the *Client* check box. This will mean that no listener will be setup for this queue manager, as it is to operate in client mode.

[3] This message only appears if a local connection does not already exist (as in this case). Adding additional aliases will not display the message again.

The window should look similar to this:

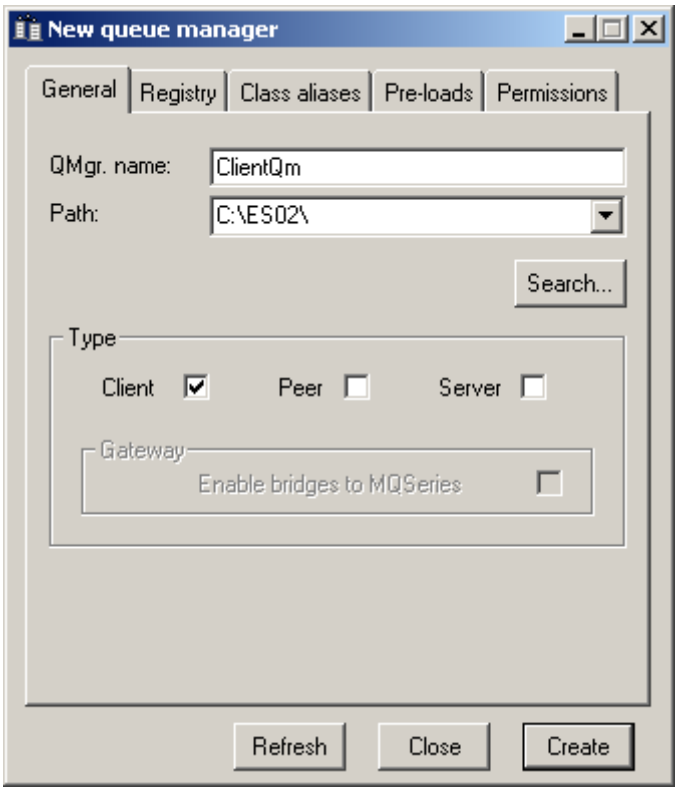

Figure 8-7 Setting name and type of ClientQm

Click the *Create* button to now create the client-side queue manager. A window confirming creation of the queue manager appears, similar to this:

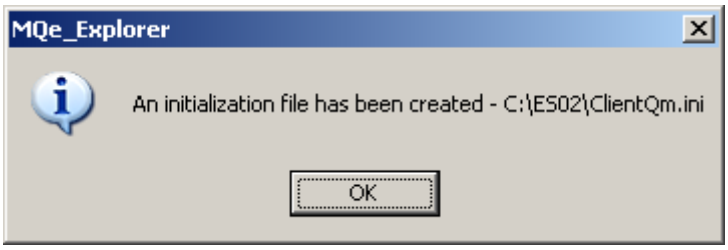

Figure 8-8 ClientQm creation confirmation

The queue manager has now been created, and is in fact running. As for the server queue manager, you can expand the tree in the MQe_Explorer window to view the default objects.

8.4 Configuring WASSeverQm queue manager

Start another MQe_Explorer session, and from the initial MQe_Explorer window, select File->New->QueueManager. Enter the name of the queue manager as ‘WASServerQm’ in the field *QMgr. name*.

Check that the *Server* check box is ticked.

Then select the *Comms* tab. Enter the IP address of the machine. Enter a port number. When the queue manager is running in WebSphere Application Server, the queue manager will not have a listener running, so will not be using the port you specify here in any case. However it is useful to specify a valid port. Doing so means you can run this queue manager outside of

WebSphere Application Server to verify connections between it and other queue managers, and also to enable testing without WebSphere in the mix. If running this queue manager on the same machine as *ServerQm*, be sure to specify a different port, e.g. 8083.

As *WASServerQM* will be running in WebSphere Application Server, it will be expecting HTTP type communications. In the *adapter* field, select:

```
com.ibm.mqe.adapters.MQeTcpipHttpAdapter
```

In the *options* field select either '(default)' or '(none)'. Options are extra information passed to the adapter; in the case of the HTTP adapter above, default is interpreted to mean that no options are to be passed. Do not pass options here; otherwise connections to this queue manager will not work.

Click the *Create* button to now create the queue manager. A window confirming creation of the queue manager will appear.

8.5 Creating connections

Prior to defining the queues create the connections between the queue managers. Creating remote queue definitions cannot be done with the MQe_Explorer tool unless a connection of that name exists.

Connection definitions on ClientQm

Using MQe_Explorer, start the *ClientQM* queue manager.

ClientQm to ServerQm

Right click on the connections object in the tree, and select the *New Connection* menu item.

In the window that is displayed, enter 'ServerQM' into the *Name* field and a descriptive text into the *Description* field.

The window should look similar to this:

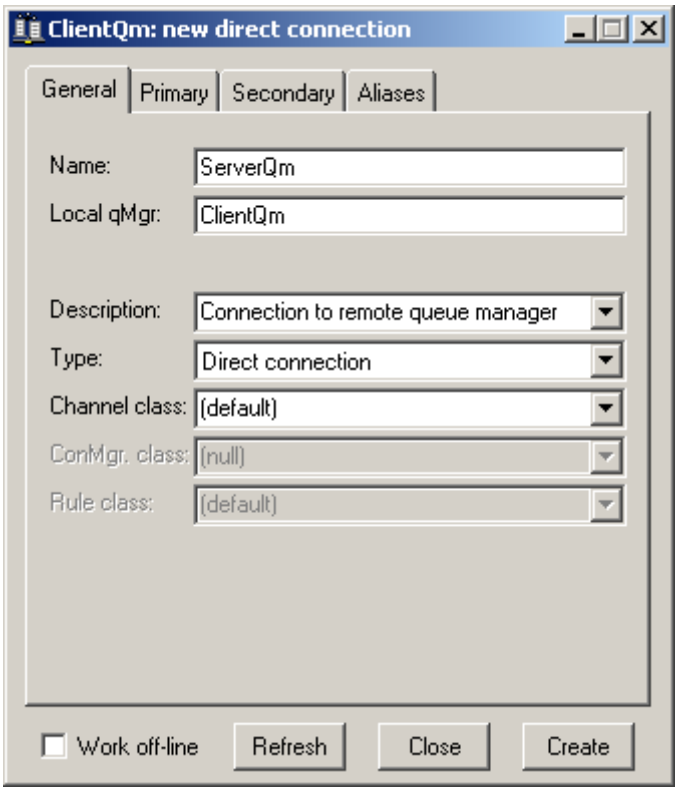

Figure 8-9 Defining a connection to a remote queue manager

Then click on the *Primary*. This tab is used to specify the IP address of the machine hosting the remote queue manager that you want to connect to.

There are three fields to change here.

Type in the IP address of the machine you have defined the server queue manager on. It could be that you have defined the client and server queue manager on the same machine, in which case you could just enter the IP address of '127.0.0.1', which is the traditional loop back address. However it is recommended to specify the IP address.

In the *Port* field type in the port number that *ServerQm* is listening on. In the description above for setting up the server queue manager, we used the default value of 8082. Type '8082' into this field.

In the *Adapter* field, select from the drop down box the same adapter you specified when setting up the server queue manager. In this case select the adapter called:

com.ibm.mqe.adapters.MQeTcpipHistoryAdapter

This is a supplied adapter, which will result in the messages flowing between the two queue managers using standard TCP/IP. The window should look similar to this:

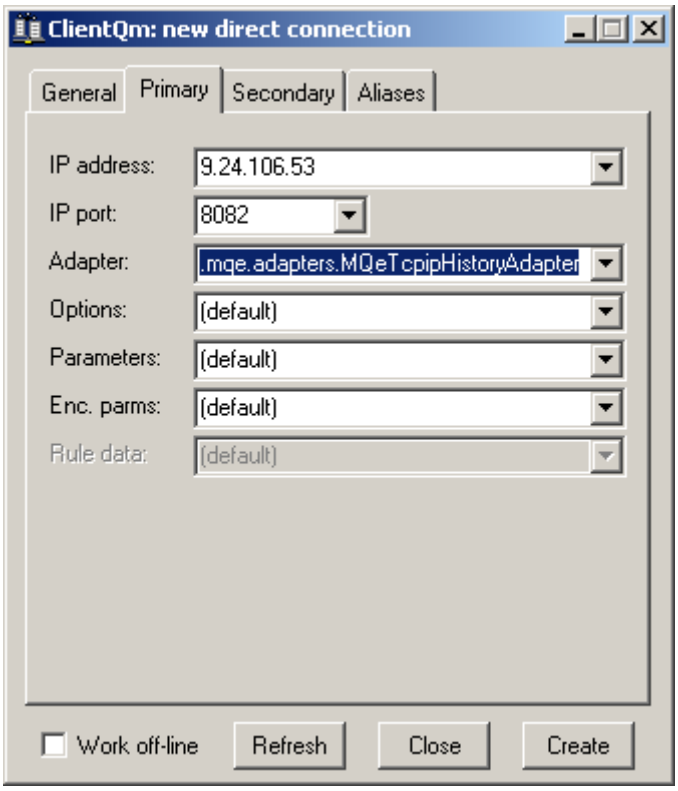

Figure 8-10 Defining the location of a remote queue manager

Then click the *Create* button to have the connection created. No confirmation window is displayed, but you can view the connection definition in the MQe_Explorer window.

ClientQm to WASServerQm

Right click on the connection object in the tree, and select the *New connection* menu item.

In the window that is displayed, enter 'WASServerQm' into the *Name* field.

Then click on the *Primary* tab. This tab is used to specify the IP address where the remote queue manager that you want to connect to is located.

There are five fields to change here.

Type in the IP address of the machine where the WebSphere Application Server is running.

In the *Port* field, type in '80' as the port value, as this is the default port for HTTP traffic, the HTTP server will be listening on[4].

In the *Adapter* field, select from the drop down box the same adapter you specified when setting up the queue manager to run in WebSphere. In this case select the adapter called:

com.ibm.mqe.adapters.MQeHTTPAdapter

[4] Note version 1.23 and earlier of the ES02 SupportPac do not allow a value of 80 to be entered into this field.

This is a supplied adapter, which will result in the messages flowing between the two queue managers using the HTTP protocol.

In the *Options* field select '(none)' from the drop down box.

In the *Parameters* field, type in '/ITSO/ITSOMQeWas', the URL that will invoke the servlet in WebSphere.

The window should similar to this:

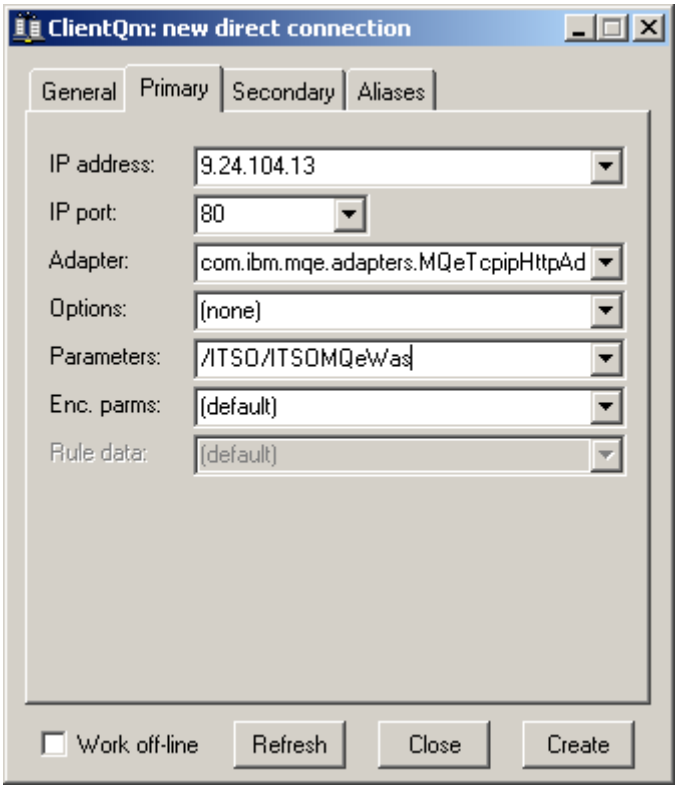

Figure 8-11 Defining a connection using the HTTP adapter

Then click the *Create* button to have the connection created. No confirmation window is displayed.

ClientQm to ServerQmViaWas

Right click on the *Connections* object in the tree, and select the *New connection* menu item.

In the window that is displayed, enter 'ServerQmViaWas' into the *Name* field. In the *Type* field select 'Indirect connection'. The connection to *ServerQmViaWas*, is in reality to the queue manager called *ServerQm*, and messages are to be sent via the queue manager running in WebSphere, an indirect connection.

Then click on the *Primary* tab. Most fields here are greyed out. Enter 'WASServerQm' into the *Via qMgr* field, to indicate that messages destined for *ServerQmViaWas* go via the *WASServerQm* connection.

The window will look similar to this:

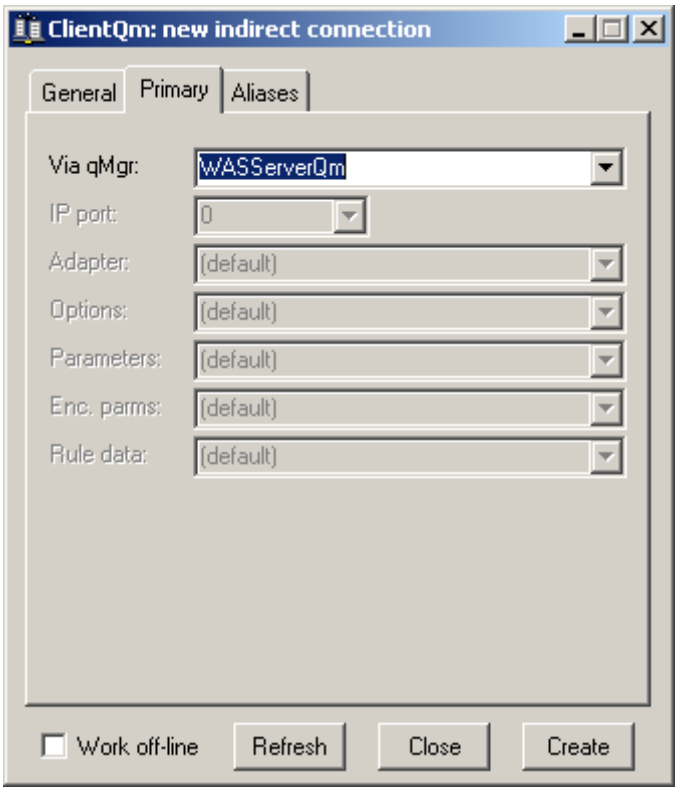

Figure 8-12 Defining an indirect connection

Then click the *Create* button to have the connection created. No confirmation window is displayed.

Connection definitions on ServerQm

Using MQe_Explorer, start the *ServerQm* queue manager.

ServerQm to WASServerQm

Right click on the *Connections* object in the tree, and select the *New connection* menu item.

In the window that is displayed, enter 'WASServerQM' into the *Name* field.

Then click on the *Primary* tab. This tab is used to specify the IP address where the remote queue manager that you want to connect to is located.

There are three fields to change here.

Type in the IP address of the machine where the WebSphere Application Server is running.

In the *Port* field, type in '80' as the port value, as this is the default port for HTTP traffic, the HTTP Server will be listening on.

In the *Adapter* field, select from the drop down box the same adapter you specified when setting up the queue manager to run in WebSphere. In this case select the adapter called:

com.ibm.mqe.adapters.MQeTcpipHttpAdapter

In the *Options* field, select '(none)' from the drop down box.

In the *Parameters* field, type in '/ITSO/ITSOMQeWas', the URL that will invoke the servlet in WebSphere.

Then click the *Create* button to have the connection created. No confirmation window is displayed.

Connection definitions on WASServerQm

Using MQe_Explorer, start the WASServerQm queue manager.

WASServerQm to ServerQm

Right click on the *Connections* object in the tree, and select the *New Connection* menu item.

In the window that is displayed, enter 'ServerQM' into the *Name* field.

Then click on the *Primary* tab. This tab is used to specify the IP address where the remote queue manager that you want to connect to is located.

There are three fields to change here.

Type in the IP address of the machine where the *ServerQM* queue manager is running.

In the *Port* field, type in '8082' as the port value, the port that the *ServerQm* is listening on.

In the *Adapter* field, select from the drop down box the same adapter you specified when setting up the *ServerQm* queue manager. In this case select the adapter called:

```
com.ibm.mqe.adapters.MQeTcpipHistoryAdapter
```

The chat room application will be sending messages to *WASServerQm* that are destined for *ServerQmViaWas*, which is an alias for *ServerQm*. To have these messages forwarded, we use the alias capability of MQSeries Everyplace. Click on the *Aliases* tab. Type in 'ServerQmViaWas' and click the *Add* button. The window should look similar to this:

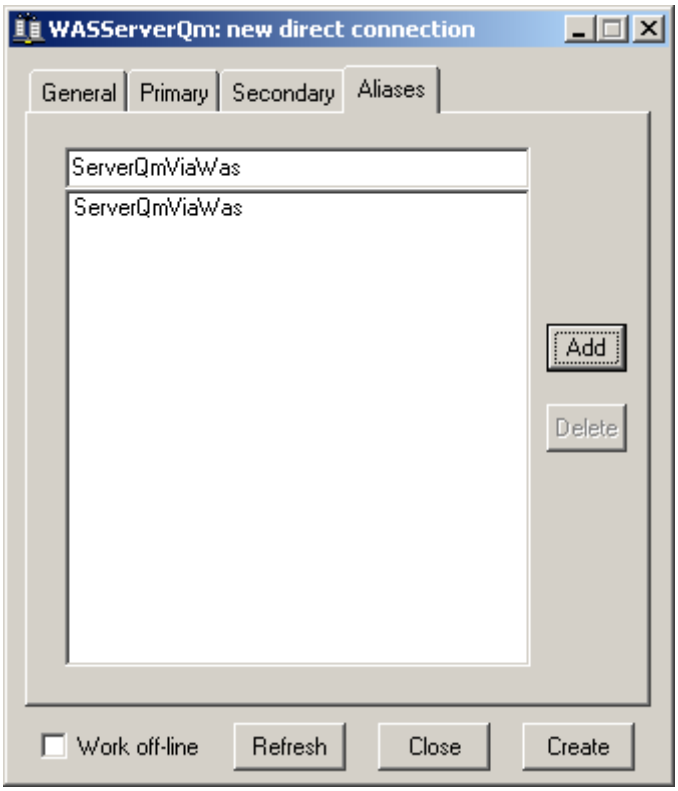

Figure 8-13 Defining an alias on a connection

Then click the *Create* button to have the connection created. No confirmation window is displayed.

8.6 Define ServerQm queues

The following queues need to be defined on *ServerQm*.

Local queue: ChatRoomQ

From the expanded tree view, right click on the *Local queues* object, and then select the *New Queue* menu item. A window will appear, in which to enter the details of the queue you wish to define.

On the General tab, type ‘ChatRoomQ’ into the *Name* field. The window should look similar to this:

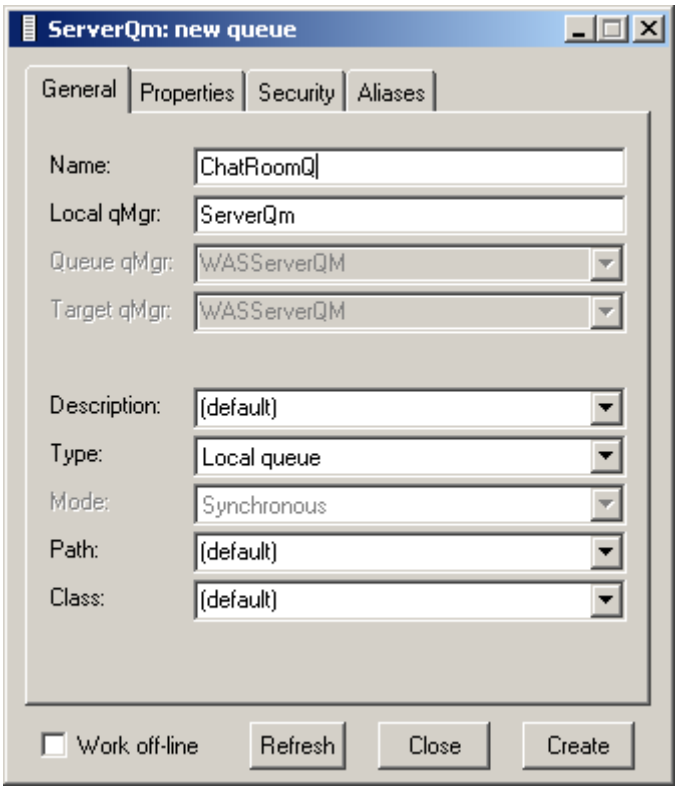

Figure 8-14 Naming the queue to be created

Then click on the *Aliases* tab. In the input text box on this tab type in ‘ChatRoomQAsync’, then click the *Add* button. Add two further entries ‘ChatRoomQViaWas’ and ‘ChatRoomQAsyncViaWas’. These alias entries tell the *ServerQm*, that any messages that it receives for these queues are to be placed on the *ChatRoomQ* queue.

The window should look similar to this:

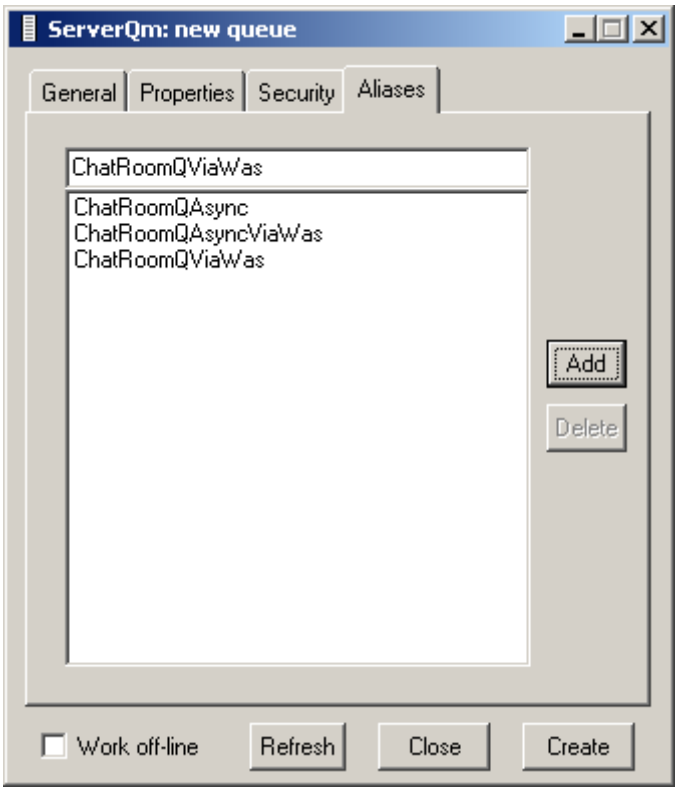

Figure 8-15 Setting up alias names for this queue

Then click the *Create* button. No confirmation window is displayed, but the *Create queue* window is re-displayed allowing you to define another queue if required.

Remote queue: ChatClientQViaWas

You can use the *Create queue* window still displayed from the previous step, of if you closed that window then, as before, from the expanded tree view, right click on the *Local queues* object, then select the *New Queue* menu item, and a window will appear.

On the *General* tab, type ‘ChatClientQViaWas’ into the Name field. There are two more fields to change here.

Change the queue type to one of ‘Remote queue’, by selecting that value from the drop down box in the *Type* field.

In the *Queue qMgr* field, type in the value ‘WASServerQm’.

Check the *Mode* field is set to ‘Synchronous’.

The window should look similar to this:

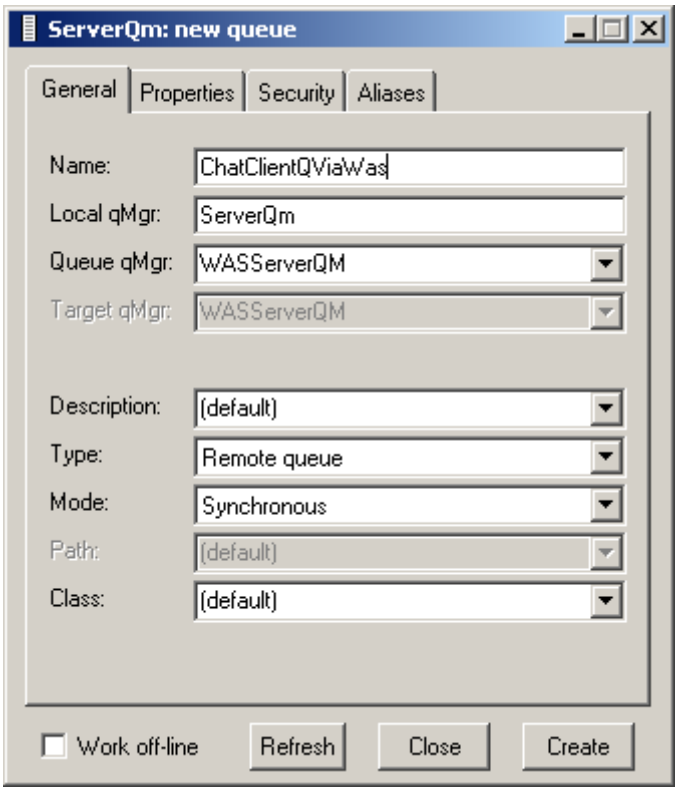

Figure 8-16 Defining a remote queue

Then click the *Create* button. No confirmation window is displayed, but the *Create queue* window is re-displayed allowing you to define another queue if required.

Store and forward queue: ChatSFQ

You can use the *Create queue* window still displayed from the previous step, of if you closed that window, then as before from the expanded tree view, right click on the *Local queues* object, then select the *New Queue* menu item. A window will appear, in which to enter the details of the queue you wish to define.

On the *General* tab, type ‘ChatSFQ’ into the *Name* field.

Most importantly, change the queue type to one of ‘Store and forward queue’, by selecting that value from the drop down box in the *Type* field. Also ensure that the *Target qMgr* field is set to ‘(none)’, i.e. messages are not be forwarded to another queue manager.

The display will look similar to this:

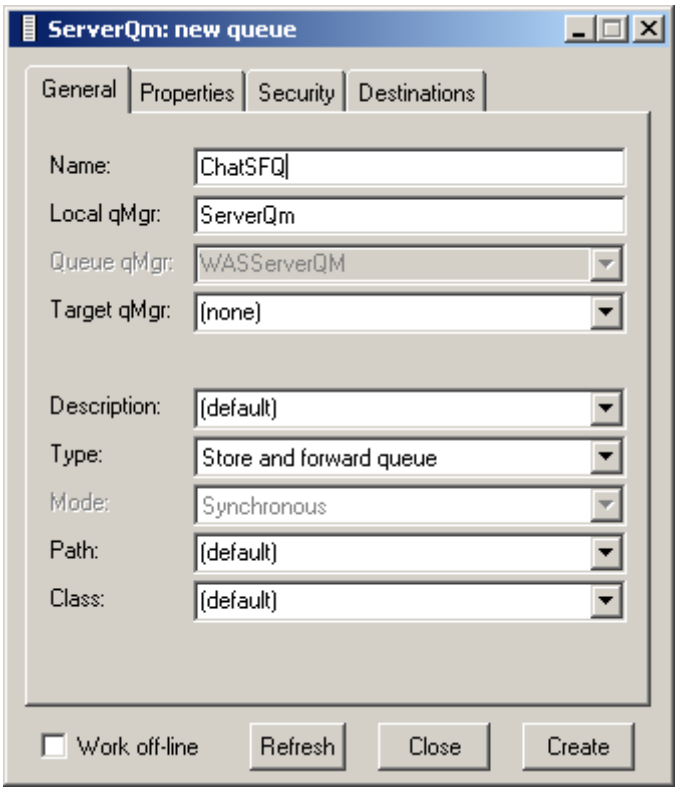

Figure 8-17 Creating the store and forward queue

This queue is used to store messages that will subsequently be pulled by remote, client-type queue managers, in our case the client-side queue manager. The server-side queue manager needs to know that it is to use this queue to store messages destined for the client-side queue manager. To do this, select the *Destinations* tab. In the window type in the value 'ClientQm' and click the *Add* button. The display will look similar to this:

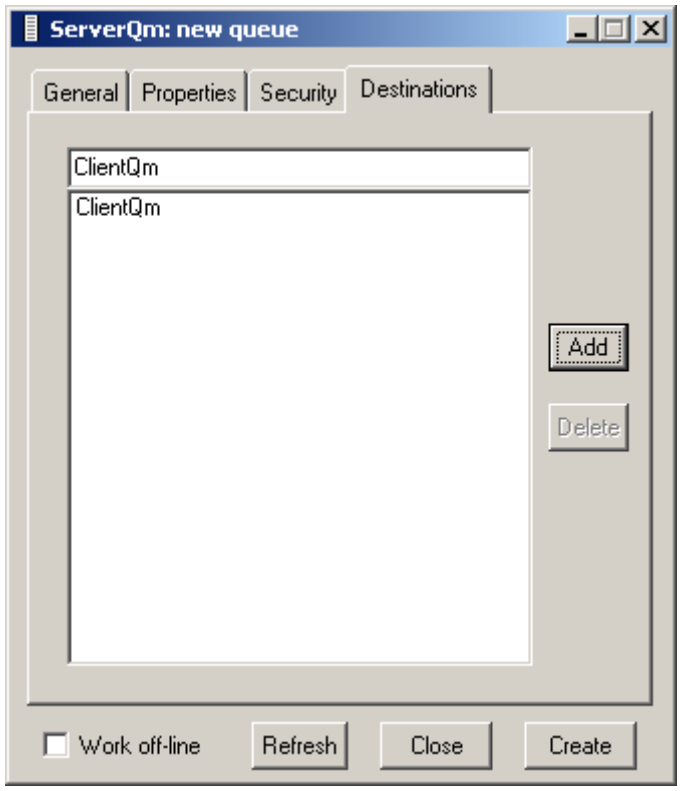

Figure 8-18 Adding the client queue manager as a target to the store and forward queue

Then click the *Create* button and the queue will be defined. No confirmation window is displayed.

8.7 Define ClientQm queues

Local queue: ChatClientQ

This is the same process as for setting up the local queue called *ChatRoomQ* on the server queue manager.

From the expanded tree view, right click on the *Local queues* object, then select the *New Queue* menu item. A window will appear, in which to enter the details of the queue you wish to define.

On the *General* tab, type 'ChatClientQ' into the *Name* field.

Then click the *Create* button. No confirmation window is displayed, but the *Create queue* window is re-displayed, allowing you to define another queue if required.

Remote Queue: ChatRoomQ

You can use the *Create queue* window still displayed from the previous step, of if you closed that window, then as before from the expanded tree view, right click on the *Local queues*

object, then select the *New Queue* menu item. A window will appear, in which to enter the details of the queue you wish to define.

On the General tab, type ‘ChatRoomQ’ into the *Name* field.

There are three fields to change here.

Change the queue type to one of ‘Remote queue’, by selecting that value from the drop down box in the *Type* field.

In the *Queue qMgr* box, select from the drop down box the name of the remote queue manager, in this case ‘ServerQm’.

Change the mode to ‘Synchronous’, by selecting that value from the drop down box in the *Mode* field.

Then click the *Create* button. No confirmation window is displayed, but the *Create queue* window is re-displayed allowing you to define another queue if required.

Remote Queue: ChatRoomQAsync

You can use the *Create queue* window still displayed from the previous step, of if you closed that window, then as before from the expanded tree view, right click on the *Local Queues*, then select *New Queue* menu item. A window will appear, in which to enter the details of the queue you wish to define.

On the *General* tab, type ‘ChatRoomQAsync’ into the *Name* field.

There are three fields to change here.

Change the queue type to one of ‘Remote queue’, by selecting that value from the drop down box in the *Type* field.

In the *Queue qMgr* box, select from the drop down box the name of the remote queue manager, in this case ‘ServerQm’.

Change the mode to ‘Asynchronous’, by selecting that value from the drop down box in the *Mode* field. The display will be similar to this:

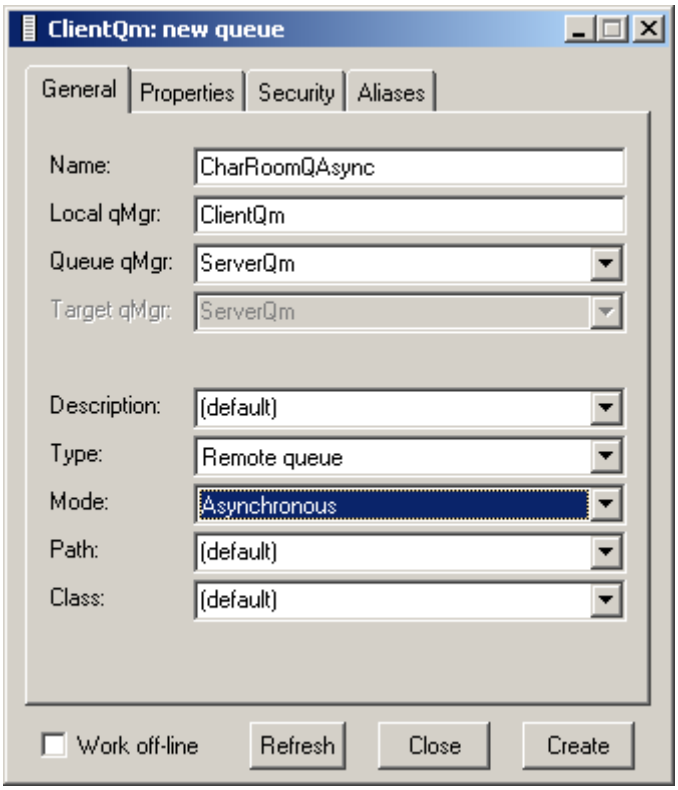

Figure 8-19 Defining a remote queue on the client

Then click the *Create* button and the queue will be defined. No confirmation window is displayed.

When the chat room application tries to send a message, it first tries to put the message to the local queue called *ChatRoomQ* on the server queue manager.

If there is no connection, the application will put the message to this queue. However, because the mode is asynchronous, the queue manager will store the message locally, and then send it to the server queue manager when the connection becomes available.

If the mode was synchronous and the connection was down, then the put message would fail.

Remote Queue: ChatRoomQViaWas

In the window for defining a new queue, on the *General* tab, type ‘ChatRoomQViaWas’ into the *Name* field.

There are three fields to change here.

Change the queue type to one of ‘Remote queue’, by selecting that value from the drop down box in the *Type* field.

In the *Queue qMgr* box, select from the drop down box the name of the remote queue manager, in this case ‘WASServerQm’.

Change the mode to ‘Synchronous’, by selecting that value from the drop down box in the *Mode* field.

Then click the *Create* button. No confirmation window is displayed, but the *Create queue* window is re-displayed allowing you to define another queue if required.

Remote Queue: ChatRoomQAsyncViaWas

In the window for defining a new queue, on the *General* tab, type 'ChatRoomQAsyncViaWas' into the *Name* field.

There are three fields to change here.

Change the queue type to one of 'Remote queue', by selecting that value from the drop down box in the *Type* field.

In the *Queue qMgr* box, select from the drop down box the name of the remote queue manager, in this case 'WASServerQm'.

Change the mode to 'Asynchronous', by selecting that value from the drop down box in the *Mode* field.

Then click the *Create* button. No confirmation window is displayed, but the *Create queue* window is re-displayed allowing you to define another queue if required.

Home Server Queue: ChatSFQ

In the window for defining a new queue, on the *General* tab, type 'ChatSFQ' into the *Name* field.

There are two fields to change here.

Change the queue type to one of 'Home server queue', by selecting that value from the drop down box in the *Type* field.

In the *Queue qMgr* box, select from the drop down box the name of the remote queue manager, in this case 'ServerQm'.

This home server queue will poll the remote corresponding store and forward queue on the remote queue manager, and pull any messages it find there to this queue.

The window will look something like this:

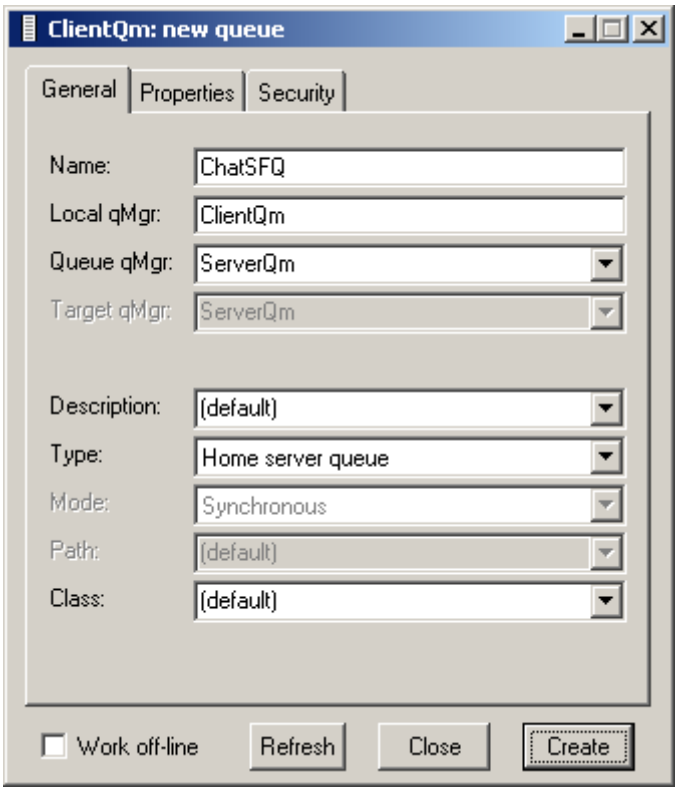

Figure 8-20 Defining a home server queue on the client

Then click on the *Properties* tab. There is one field here to be changed, *Time interval*.

Specify a value here in milliseconds. Specifying a value here greater then zero, tells the client queue manager how often to automatically poll the server-side queue. Set this to some reasonable value, such as 5000, which will mean a check occurs every 5 seconds.

Then click the *Create* button and the queue will be defined. No confirmation window is displayed.

Home Server Queue: ChatSFQViaWas

In the window for defining a new queue, on the *General* tab, type 'ChatSFQViaWas' into the *Name* field.

There are two fields to change here.

Change the queue type to one of *Home server queue*, by selecting that value from the drop down box in the *Type* field.

In the *Queue qMgr* box, select from the drop down box the name of the remote queue manager, in this case, *WASServerQm*.

Then click on the *Properties* tab. There is one field here to be changed, *Time interval*.

Specify a value here in milliseconds. Specifying a value here greater then zero, tells the client queue manager how often to automatically poll the server-side queue. Set this to some reasonable value, such as 5000, which will mean a check occurs every 5 seconds.

Then click the *Create* button and the queue will be defined. No confirmation window is displayed.

8.8 Define WASServerQm queues

Local: ChatClientQViaWas

In the window for defining a new queue, on the *General* tab, type 'ChatClientQViaWas' into the *Name* field.

The *Mode* should default to 'Synchronous', and the *Type* to 'Local queue'.

Then click the *Create* button. No confirmation window is displayed, but the *Create queue* window is re-displayed allowing you to define another queue if required.

Store and forward queue: ChatSFQViaWas

In the window for defining a new queue, on the *General* tab, type 'ChatSFQViaWas' into the *Name* field.

Most importantly, change the queue type to 'Store and forward queue', by selecting that value from the drop down box in the *Type* field. Also ensure that the *Target qMgr* field is set to '(none)', i.e. messages are not be forwarded to another queue manager.

Select the *Destinations* tab. In the window type in the value 'ClientQm' and click the *Add* button.

Then click the *Create* button and the queue will be defined. No confirmation window is displayed.

8.9 Java Swing setup

The chat room application uses the Java swing classes to display windows. These classes are located in a *.jar* file called *SwingAll.jar*. Search your system for one of these files and then add it to the *classpath*. For example:

```
set CLASSPATH=%CLASSPATH%;c:\Program Files\sqllib\java\swingall.jar
```

Note, be sure that the MQSeries Everyplace java classes are also accessible via the *classpath*, as mentioned in *Preparing for setup* on page 39.

8.10 Chat room application setup

As mentioned, the chat room application consists of a number of packages. Extracting the supplied files in the *.zip* file will create the appropriate relative directory structure and put the classes in the correct place.

Modify the *classpath* so that these classes are found when the application is run, by typing in:

```
set CLASSPATH=C:\ED02\;.;%CLASSPATH%
```

The `.' tells the system to look in the current directory.

8.11 Set up start up list

The chat room application demonstrates two ways of running applications with MQSeries Everyplace.

On the server side, the MQSeries Everyplace queue manager is started first, and then the chat room application is loaded.

On the client side, the chat room application is started first, and it starts the MQSeries queue manager.

Start-up List

The start up list approach demonstrates how to use MQe_Explorer to have an application loaded when the queue manager starts.

This is done by editing the *.ini* file associated with the queue manager (in this case *ServerQm.ini*).

Add the following to the bottom of the *.ini* file used for *ServerQm*:

```
[AppRunList]
(ascii)App1=itso.mqe.chatserver.RoomMgr
```

Note that more then one application can be specified, and initialization data can also be passed if required.

8.12 Configure WebSphere

WebSphere Application server will require some configuration to allow the servlet to be run. The servlet was tested in both V3.5 and V4 of WebSphere Application Server, but will only describe here deployment of the servlet in V3.5.

Defining the servlet to WebSphere can be done in many ways, however we chose to define a separate web application under the default application server.

Use the WebSphere Administrative console, to create a new web application, and then create a servlet definition. The window that defines the servlet should look similar to this:

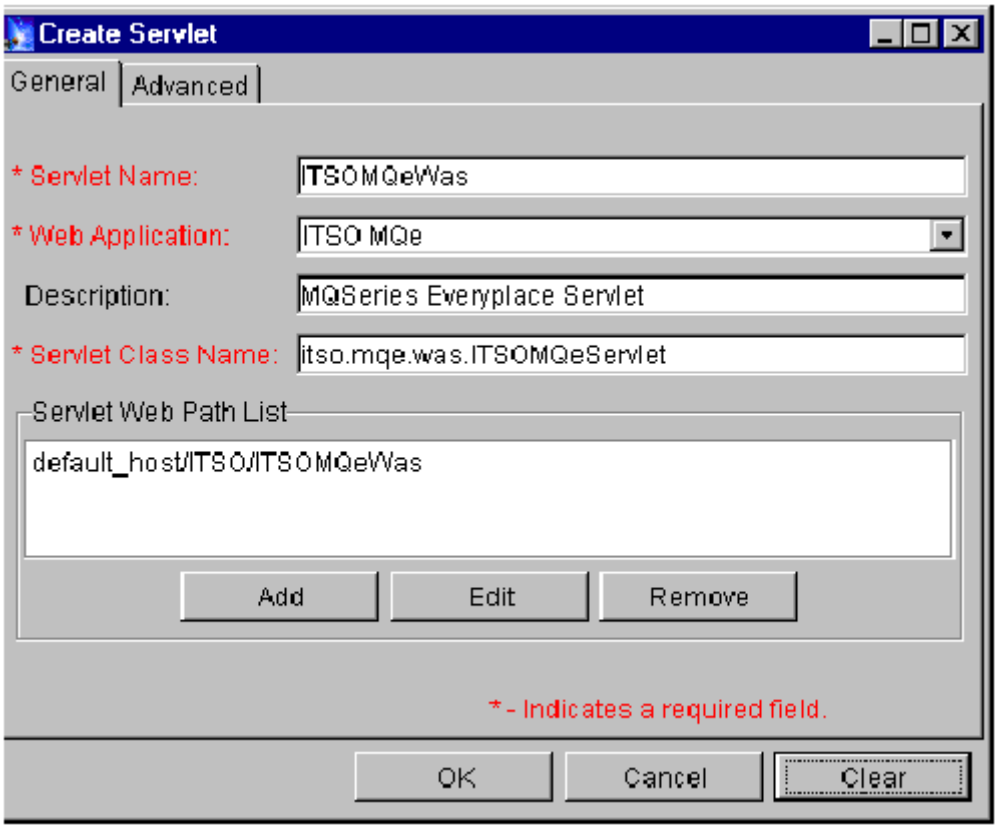

Figure 8-21 Defining the ChatRoom servlet

Add the following directories to the *classpath* for the web application that will run the servlet:

- *C:\Program Files\MQe\Java*
- *C:\ED02*
- *C:\WebSphere\AppServer\hosts\default_host\WSsamples_app\servlets*

What is being added here is the location of the MQSeries Everyplace class files, and the classes used by the chat room application. The last *classpath* is required as the servlet that runs the queue manager, imports this package as part of the YourCo example (see *Extending the YourCo Application* on page 75). Note you will need to copy the *TotalLeaveBean* class from the *.zip* file containing the chat room application, to the above directory, as this class was developed for this SupportPac, and is not part of the supplied WebSphere Application Server YourCo sample.

Note that messages written out by the chat room application are written to standard output, which will appear in the default standard output file for the application server. These messages advise if the queue manager is started successfully for example.

8.13 Set up property files

The following property files need to be defined.

clientChat.properties

This property file is used only on the client side, and can be used to pass the names of queues and queue managers to the application to be used instead of the defaults coded in the Java programs. A sample is shown below:

```
iniFile=C:\\ED02\\ClientQm.ini
clientQueue=ChatClientQ
destQueue=ChatRoomQ
destQueueAsync=ChatRoomQAsync
destQMgr=ServerQm
clientQm=ClientQm
WASIpAddr=9.24.106.53
```

Example 8-1 Sample clientChat property file

Note the values shown above match the default values coded in the java code, so a property file does not need to be used if the defaults are used.

It should be placed in the *ED02* directory.

MQe.properties

This property file is used only by the servlet running in WebSpere Application Server. This property file is used to pass in the location of the *.ini* file for the queue manager to be started by the servlet, and a flag to do with the YourCo example described in *Extending the YourCo Application* on page 75.

The file contains just two lines like this:

```
IniFile=C:\\ED02\\WasServerQm.ini
YourCo=No
```

If you do plan to use the YourCo example, then set the value of the *YourCo* property to anything other then ‘Yes’. If the value is set to ‘Yes’, then you will need to define the *YourCoQuery* queue in the *WASServerQM*, otherwise you will get an exception in the servlet. The property file should also be placed in the *ED02* directory.

ejbLocation.properties

A number of parameters are required so that the bean used in WebSphere can locate the EJB of the YourCo application. These parameters are specified in this property file. The contents of which look like this:

```
userID=WSDEMO
password=wsdemo1
URL=jdbc:db2:SAMPLE
driver=COM.ibm.db2.jdbc.app.DB2Driver
dataSourceName=jdbc/sample
factory=com.ibm.ejs.ns.jndi.CNInitialContextFactory
accessName=Access
providerURL=iiop://9.24.104.13:900
```

Example 8-2 Properties to locate EJB

The IP address in the example above needs to be the address of the system that is executing the EJBs.

This file should be placed into the *ED02* directory. Note that if you do not plan to try out the YourCo example, then you do not need to set up this property file.

8.14 Starting the chat room application

This section describes how to start and use the chat room application.

Starting the server side

To start the server side of the chat room application, follow these steps.

- Open a DOS window
- Set up the *classpath* as described in *Preparing for setup* on page 39
- Change to the *C:\Program Files\MQe\Java\MQe_Explorer* directory
- Type in 'MQe_Explorer.exe'
  - This displays MQe Explorer window, select File->Open
  - A file dialog box appears, locate the *.ini* file for the server queue manager called *ServerQm.ini* and select it
  - The queue manager will start, and its objects can be viewed in the tree
  - It will start the chat room application due to it being specified in the *AppRunList* stanza of the *.ini* file
  - A window will appear titled 'MQSeries Everyplace Server'
  - Note: you cannot send any messages until the client connects

Server-side chat window

The server-side chat window will initially look like this:

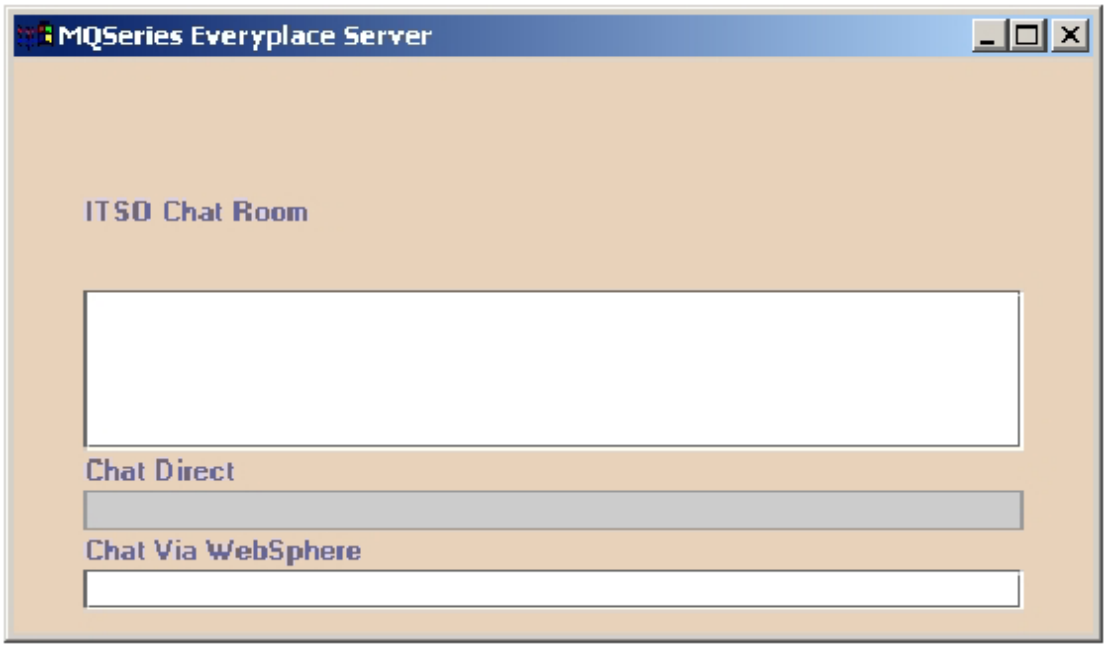

Figure 8-22 Initial server-side chat window

Starting the client side

To start the client side of the chat room application, follow these steps.

- Open a DOS window where *ClientQM* is set up and setup the *classpath*
- Type in 'java itso.mqe.chatclient.ClientMgr'
  - Note that this will first start the chat room application, then application will start the queue manager
  - If you have set up a property file for the client side, add '-p' to the line that invokes the application, this causes the application to read the property file
- A window will appear titled 'MQSeries Everyplace Client'

Client-side chat window

The client-side chat window will initially look like this:

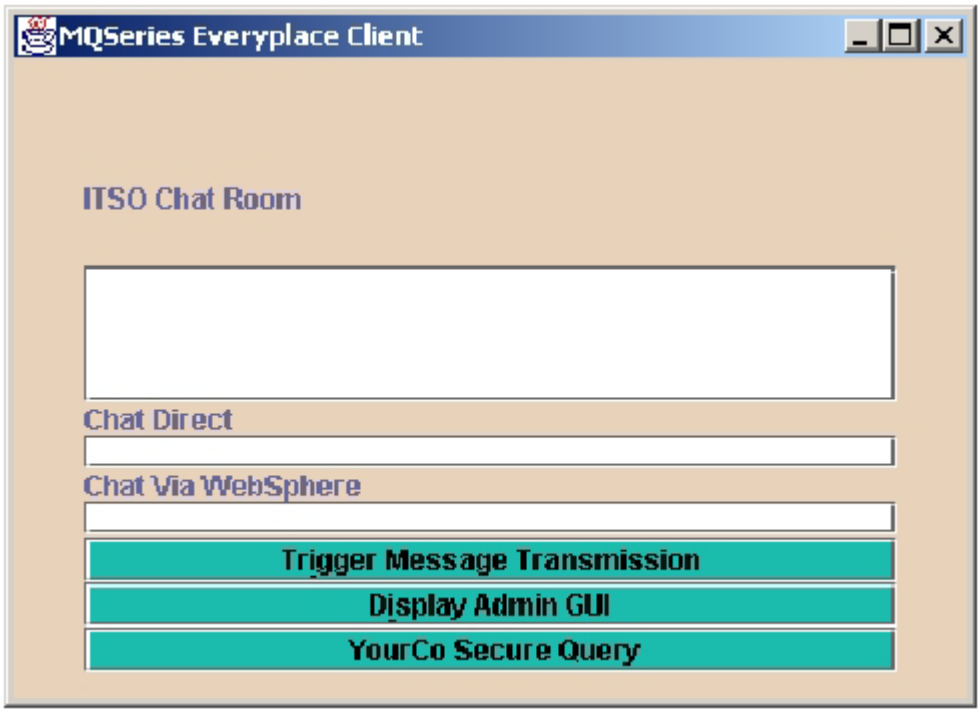

Figure 8-23 Initial client-side chat window

8.15 Operating the chat window

Once the two windows are displayed, you are ready to begin chatting.

Note that the client side must initiate the chat.

In the client window, in the input box below the text ‘Chat Direct’, type in some text and press the *Enter* key.

The text you have typed in should then appear in the server-side window, in output text area below the text ‘ITSO Chat Room’.

The *Chat Direct* input text area on the server-side chat room will now be enabled for input. Type in a message and press the *Enter* key. There will be a slight delay before the message appears in the output text area on the client window, as recall that the client is polling the server store and forward queue at a defined interval.

Then to test chatting via WebSphere, type messages into the *Chat via WebSphere* input boxes.

Continue to chat between the two windows as required.

8.16 Asynchronous chatting

To demonstrate asynchronous message transfer, the client side of the application can be run without the *ServerQm* or *WASServerQm* queue managers being active.

Start just the chat room application on the client side. In the *Chat Direct* input text box type a message. The application first tries to put to the local queue on the remote server, but will not

be able to as the server-side queue manager is down. The application detects this, and then puts the message to the remote queue called *ChatRoomQAsync*. MQSeries Everyplace stores the message locally.

You can verify this by looking in the directory on the file subsystem that is being used to store messages. For example say you typed in three messages, then you can see that there are three files in the corresponding directory, as shown below:

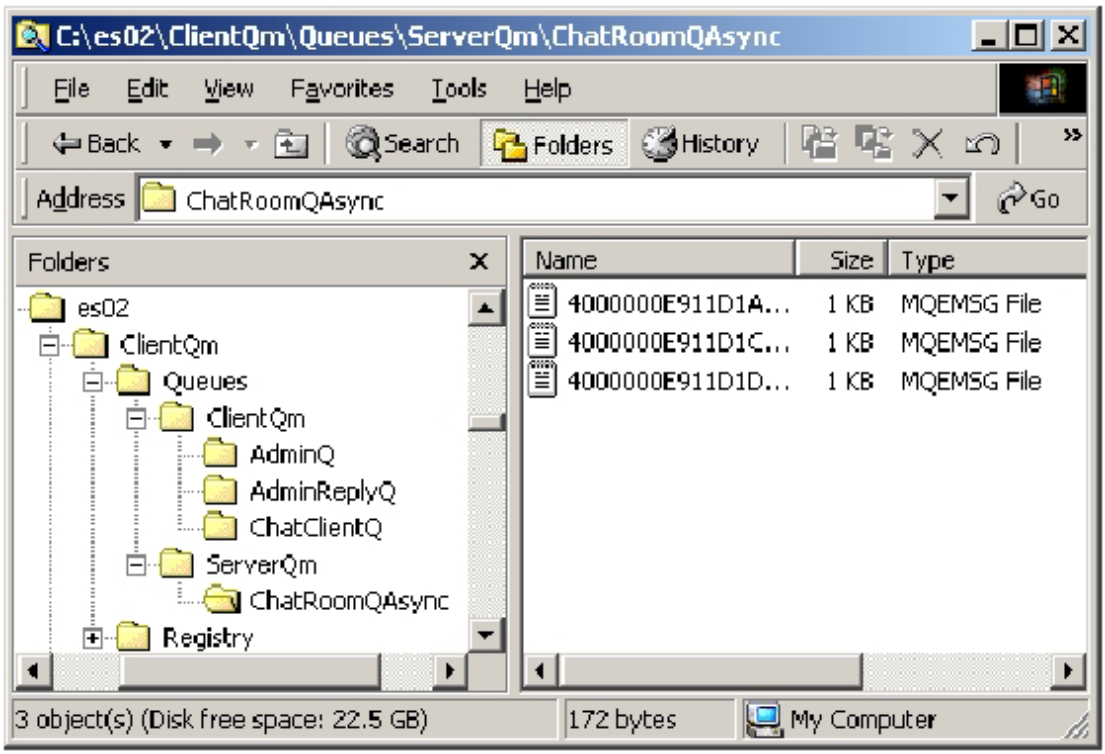

Figure 8-24 Messages waiting to be sent

The messages are in the *ChatRoomQAsync* directory.

MQSeries Everyplace will send these messages when there is a connection to the server-side queue manager.

You can also try stopping the application server where the servlet is defined. Then try to send a message using the *Chat via WebSphere* boxes. As no connection can be established, the messages will be written to the *ChatRoomQAsyncViaWas* queue on *ClientQm*.

Triggering transmission

However for asynchronous message transfer to occur, the queue manager must be triggered. When triggered, the queue manager will attempt to send the queued messages. If a connection is available the messages are sent, if the connection is down, no messages are sent.

The queue manager will not try again until triggered again.

Thus if you now start the server-side queue manager, the messages are not automatically sent.

With the chat room application, there are two ways to trigger this transfer once the server queue manager is restarted.

One way is to stop the client queue manager and restart it. When restarted the trigger transmission method is invoked which causes it to try to send any queued messages.

The second way is to leave the client chat room application running, and click the *Trigger Transmission* button labeled in the client chat window. When clicked this causes the trigger transmission method to be invoked on the queue manager, and the messages to be sent.

It is the responsibility of the application to provide some method to have a trigger transmission method issued on the queue manager, if asynchronous messaging is being used.

8.17 The admin GUI

On the client chat room window is a Display Admin GUI button. When clicked the following window appears:

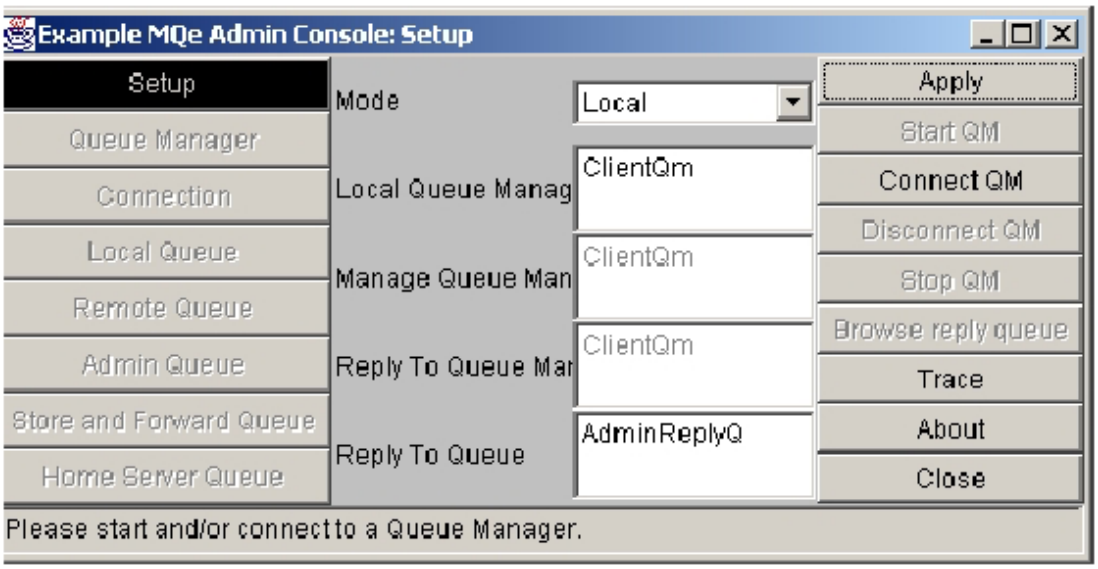

Figure 8-25 The administration GUI

This GUI is an example supplied with MQSeries Everyplace. Its use is described in the MQSeries Everyplace Programming Guide.

It can be used to inquire, update and create definitions in the queue manager.

Trace facility

One of the most useful features is the trace option. If you click on the trace button, the following window appears:

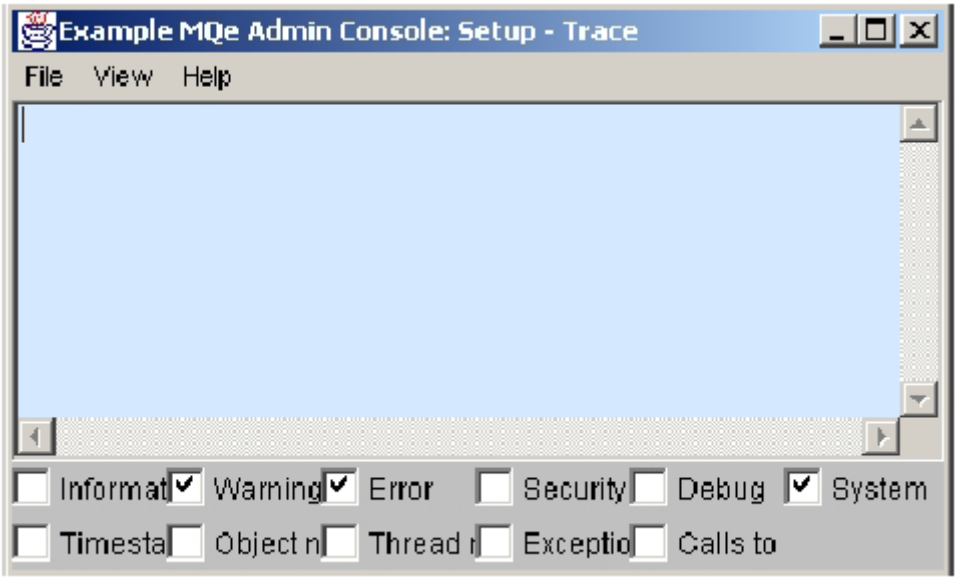

Figure 8-26 MQSeries Everyplace trace window

This can be used to trace activity in MQSeries Everyplace. If your Java program writes messages to standard output, they will appear in this window.

8.18 Encryption and the stress test

As a simple example of using the encryption services supplied with MQSeries Everyplace, and as a measure to compare the performance of using TCP/IP versus HTTP over TCP/IP for sending messages a very simple stress test is incorporated into the chat room application.

The following section describes how to set it up the queues required, how to run the test, and also how to set up and verify encryption.

ClientQm queues

Shut down the client side of the chat room application if it is running, and start up the *ClientQM* queue manager using MQe_Explorer.

Open a create queue window.

On the *General* tab, type ‘StressQ’ into the *Name* field.

There are three fields to change here.

Change the queue type to one of ‘Remote queue’, by selecting that value from the drop down box in the *Type* field.

In the *Queue qMgr* box, select from the drop down box the name of the remote queue manager, in this case ‘ServerQm’.

Check the mode is ‘Synchronous’, by selecting that value from the drop down box in the *Mode* field.

Then click on the *Security* tab. In the *Cryptor* field select 'com.ibm.mqe.attributes.MqeXorCryptor'. The window should look something like this:

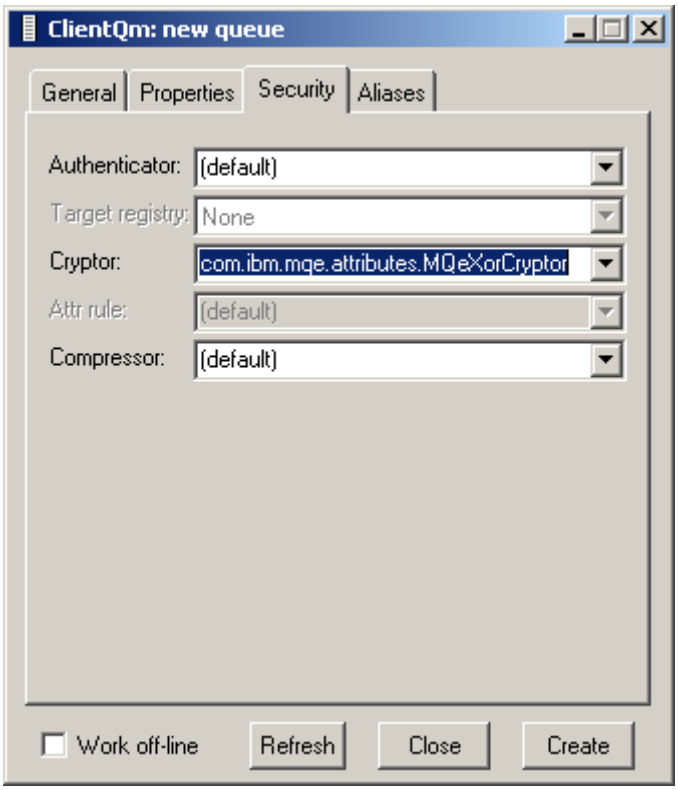

Figure 8-27 Selecting a cryptor adapter

Then click the *Create* button and the queue will be defined. No confirmation window is displayed.

Repeat this process to create another remote queue, but with a value of 'WASServerQm' in the *Queue qMgr* field.

ServerQm queue

Using MQe_Explorer, start the *ServerQm* queue manager, and open the *Create queue* window.

On the *General* tab, type 'StressQ' into the *Name* field.

The queue type should be 'Local queue', the mode 'Synchronous'.

Then click on the security tab. In the *Cryptor* field, select 'com.ibm.mqe.attributes.MqeXorCryptor'.

Then click the *Create* button and the queue will be defined. No confirmation window is displayed.

WASServerQm queue

Using the WebSphere Application Server console, stop the application server that is running the ITSOMQeWas servlet. The using MQe_Explorer, start the *WASServerQm* queue manager, and open the *Create queue* window.

In the tab labeled `General', type StressQ into the field labeled `Name'.

The queue type should be ‘Local queue’, the mode ‘Synchronous’.

Then click on the *Security* tab. In the *Cryptor* field, select ‘com.ibm.mqe.attributes.MqeXorCryptor’.

Then click the *Create* button and the queue will be defined. No confirmation window is displayed.

Then stop MQe_Explorer and restart the application server in WebSphere Application Server.

Run a stress test

Start the client side of the chat room application again. Then in the *Chat Direct* window type in the string ‘Stress Test’ and press the *Enter* key. The code in the *sendMessage* method of *ClientMgr* checks for the above string in the message to be sent. When it detects it, it will invoke the *stressTest* method. All this method does is to send ten messages of 150 bytes each to the *ServerQm*. It records the time it takes for this to occur, and calculates the throughput rate. It then displays a message with the results in the chat room window.

Typing the same string in the *Chat via WebSphere box*, sends the same set of messages to the *StressQ* on the *WASServerQm*.

If you set up the client queue manager on one workstation, and the *WASServerQm* and *ServerQm* on some other workstation, then you can use this simple stress test to compare sending messages via TCP/IP versus via HTTP.

Encryption

Since the queues were defined using an encryption adapter, the messages have been encrypted. The messages are encrypted as they are put onto the queue, and not decrypted till they are retrieved from the queue and passed to the application.

Note even though this is a synchronous put message to a local queue on a remote queue manager, the encryption would still occur on the client-side queue manager before the messages were sent.

Using MQe_Explorer, right click on the *StressQ* object, and you will see that it contains a number of messages. Double click on one of these messages, and you will have a display similar to the one shown below:

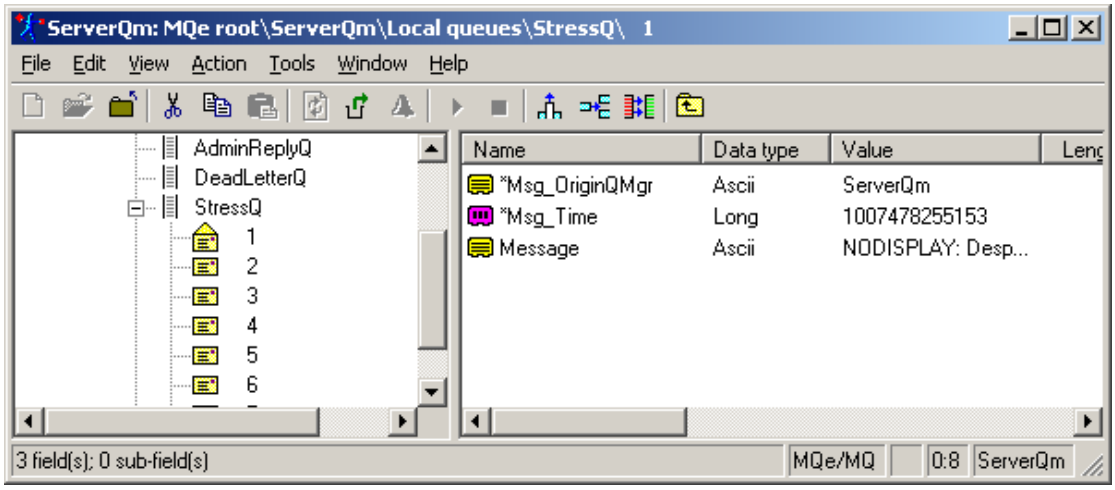

Figure 8-28 Displaying stress test messages

The text of the message is shown under the *Value* column heading, on the line with a value of 'Message' under the *Name*. The message contents can be viewed, because the application, in this case MQe_Explorer, has read the message, and the queue manager has decrypted it.

To check that the message is indeed encrypted, use the Windows Explorer program, and drill down to the *StressQ* folder in the directory where the queues associated with the *ServerQm* are stored. In this directory you will see a number of files; each file represents a message. Double click on one of the messages, and it will display in a default editor (*Notepad.exe* is fine). The contents will look something like this:

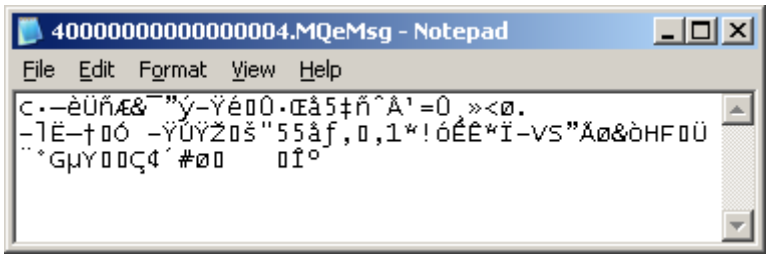

Figure 8-29 Encrypted message contents

As the message is encrypted, the contents displayed are unintelligible.

MQSeries Everyplace comes with adapters that provide much stronger levels of encryption then the one used in this example, though there are some additional steps involved in using them, which are explained in Chapter 8 in the MQSeries Everyplace Programming Guide, SC34-5845.

TCP/IP vs HTTP - comparison

Using this simple stress test, we compared throughput rate of TCP/IP between queue managers, to that of HTTP between queue managers.

We set up *ClientQm* on one machine, and *WASServerQm* and *ServerQm* on a second machine, connected over a LAN. Then ran the above stress test to each with these results:

- Took 250ms to send 1500 bytes to *ServerQm* using TCP/IP
  - Throughput rate 6000 bytes/sec
- Took 1844ms to send 10 messages of total of 1500 bytes to *WASServerQm* using HTTP, with *WASServerQm* running in WebSphere
  - Throughput rate 813 bytes/sec

8.19 Coding administration messages

Because in our example we have used MQe_Explorer to configure our queue managers, it has not been necessary to develop any code to do this task. Without MQe_Explorer, it would have been necessary to develop programs to perform the administrative task of creating and configuring queue mangers.

Configuration is done using MQSeries Everyplace administrative type messages. When a queue manager is defined, two queues to handle administration are defined. They are called:

- *AdminQ*
- *AdminReplyQ*

The messages are written to the *AdminQ*, where the queue manager actions the message, and replies are typically received on the *AdminReplyQ*.

Building these administration messages requires writing a program to just that. MQe_Explorer does this for you, by turning the actions you generate by clicking on various objects, into administration messages.

In the *ClientMgr* class there are two methods, which provide sample code to show how to build administration type messages. The *addConnection* defines a connection, while the *addQueue* method defines a queue.

9 Extending the YourCo Application

This section describes how MQSeries Everyplace can be used to extend existing applications.

9.1 Overview

The YourCo application is a sample application supplied with WebSphere Application Server. As supplied it demonstrates various features of WebSphere, EJBs, servlets etc, with access being browser based.

As part of the chat room application, some extra code was developed to demonstrate how MQSeries Everyplace could be used to access an existing application running in WebSphere. Also this example demonstrates how to set up a simple adapter to implement authenticated access to a queue.

The YourCo sample application consists in part of a database containing in one database table a list of employees, and in another table a list of different types of leave that the employees have owing them.

The aim of the example is to show how a manager on a remote device could access information securely from an application that to that stage only had a browser interface.

This example is implemented on typical Windows type desktops, with the client application running on a Windows desktop. However the client application could be implemented on a device like a palm type device, which perhaps does not support a normal browser interface. Using MQSeries everyplace, and the Wireless Gateway support of the WebSphere Everyplace Suite, a person could gain remote secure access an application that previously had required a browser to access.

9.2 YourCo extensions

The example set up here involves sending a predefined message to the *YourCoQuery* queue in the *WASServerQm* queue manager running in WebSphere. When the message arrives, a bean is invoked to determine the total amount of different types of leave due to all staff of YourCo. A message with these totals is returned to the chat room window.

The following describes the extra Java packages set up to handle the interaction with the YourCo application.

Part of the supplied YourCo application is the *WebSphereamples.YourCo.Timeout* package. This is used to display leave information about individual staff members of YourCo on a browser. The code was copied into Visual Age for Java, and then the following new classes developed:

- *totalLeaveBean* - used to store leave values
- *totalLeaveServlet* - contains method to retrieve leave info

When the init method of the *ITSOMQeWas* servlet is run, a message listener is defined for the *YourCoQuery* queue. When a message arrives on that queue, the *messageArrived* method of the *WasQMgr* object is called. The following shows the code executed from this method:

```
if (eventQueueName.indexOf("YourCoQuery") >= 0) {
        System.out.println("call ejb to get info");
        msgObj = wasQMgr.getMessage(null, "YourCoQuery", null, null, 0);
        System.out.println("From: " + msgObj.getOriginQMgr() +
                " : " + eventQueueName +
                " msg: " + msgObj.getAscii("Message"));
        findTotalLeave();
        String yourCoMsg = "YourCo leave Totals: Vactional: " + sumVactional +
                " Personal: " + sumPersonal +
                " Sick: " + sumSick;
        replyMsg.putAscii("Message", yourCoMsg);
        wasQMgr.putMessage("ClientQm", "ChatClientQ", replyMsg, null, 0);
```

Example 9-1 Handling messages on the YourCoQuery queue

The result of the above code is that the *findTotalLeave* method of the *WasQMgr* object is called. This main part of this method is shown below:

```
TotalLeaveBean totalLeaveInfo = new TotalLeaveBean();

totalLeaveInfo = totalLeaveServlet.calcTotalLeave(null);

sumVactional = totalLeaveInfo.getTotalVactional();
sumPersonal = totalLeaveInfo.getTotalPersonal();
sumSick = totalLeaveInfo.getTotalSick();
```

Example 9-2 Calling bean to access YourCo information

The above code shows that a Java bean object of type *TotalLeaveBean* is created. This bean was written for this example. The *totalLeaveServlet* class was written initially to run the *TotalLeaveBean* via a browser to test it's functionality, prior to using it in this example.

The above code shows that the *calcTotalLeave* method of the *totalLeaveServlet* class is called, which will return a bean of type *totalLeaveInfo*.

The *calcTotalLeave* method of the *totalLeaveServlet* class uses another bean developed for this SupportPac, *InviteesDBBean*. This bean returns a list of all employees from the YourCo database. Then for each employee in the list, the amount of different types of leave they have is obtained, using an existing EJB, and a running total kept for each type. The code is shown below:

```
try {
        while (true) {
                ii = ii + 1;
                employeeId = Integer.valueOf(InviteesDBBean.getEMPNO(ii)).intValue();
                System.out.println(
                        "Employee id: " + employeeId + " String: " +
                        InviteesDBBean.getEMPNO(ii));
                try {
                        totalVactional = totalVactional + access.getBalance(employeeId, 1);
```

```
                        totalPersonal = totalPersonal + access.getBalance(employeeId, 2);
                        totalSick = totalSick + access.getBalance(employeeId, 3);
                        System.out.println("tv:" + access.getBalance(employeeId, 1));
                    } catch (Exception e) {
                        System.out.println("TL - Exception: " + e.getMessage());
                        e.printStackTrace();
                    }
                } // End while
            } // End try
```

Example 9-3 Calculating the total of the different leave types

What we have demonstrated here is that a remote application using the messaging technology of MQSeries Everyplace, can easily be used to access an existing web based application.

9.3 Customized authenticator adapter

MQSeries Everyplace comes with some sample authentication type adapters. For this example however, a simple authentication adapter was developed to illustrate the principles involved.

A detailed description of the MQSeries Everyplace authentication adapter can be found in Chapter 2 of the *MQSeries Everyplace Programming Reference* manual, SC34-5846.

The following diagram outlines the processing that occurs when an authentication adapter is used.

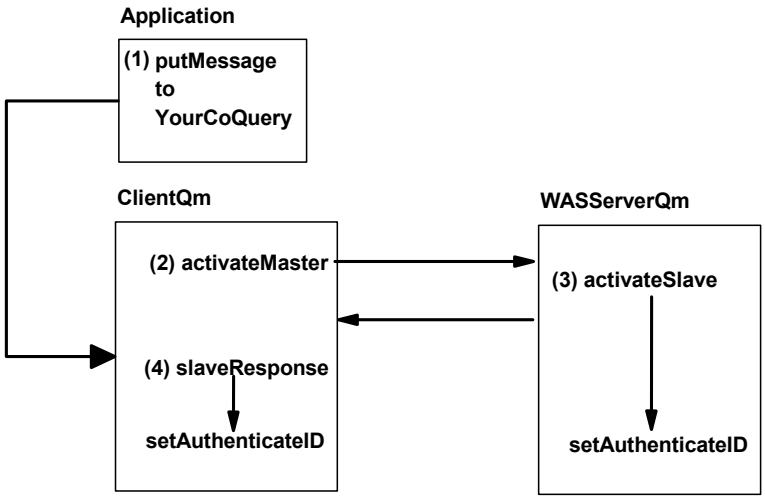

Figure 9-1 Authentication adapter flow

The processing that occurs, with reference to the above diagram, is as follows:

1. The application issues a *putMessage* to the *YourCoQuery* queue

2. The *ClientQm* queue manager detects that an authentication adapter is specified in the queue definition, and invokes the *activateMaster* in the class specified, in this case the customized adapter is in the package *itso.mqe.security.QueueAuthenticator*

    i. The *activateMaster* method will then display a small window to ask the end user for a password

    ii. The password entered by the user is returned to the queue manager to pass to the corresponding *activateSlave* method on the server queue manager, the code to do this is shown below:

```
/* Password entered by user is passed back to queue manager, which
   will send it to the corresponding activateSlave method on the server
   queue manager */

System.out.println("pwd: " + password);
String replyTxt = "From Master: " + password;
byte [] replyMsg = replyTxt.getBytes();
return replyMsg;
```

Example 9-4 Returning the password for authentication

3. On the *WASServerQm* queue manager, the *activateSlave* method of the *QueueAuthenticator* class is invoked

    i. The data passed to this method contains the password entered by the end user, code in the method validates the password and sends back a positive response, the code that does this is shown below:

```
if (recvMsg.indexOf("shazam") > 0){
try
{
        setAuthenticatedID( authID );
        replyMsg = "From Slave: Auth ok: ".getBytes();
        return replyMsg;
} /*
```

Example 9-5 Validating the password

    ii. If the password is incorrect, an exception is thrown which will result in the *activateMaster* method being re-invoked on the *ClientQm* queue manager, which will re-display the window asking for the password

    iii. The *setAuthenticatedID* method call tells the queue manager that authentication has been successfully established for the queue

4. On the *ClientQm* queue manager, the *slaveResponse* method in the *QueueAuthenticator* class is called, this method simply calls the *setAuthenticatedID* method to notify the *ClientQm* that access to the queue has been authenticated

9.4 Queue definitions

To run the example, set up the following queue definitions. It is assumed that you have setup the Chat Room Client example described in *Chat room – An MQSeries Everyplace* application on page 17.

On ClientQM: remote queue – YourCoQuery

This is a similar process you have been using to define queues in the *Chat Room* application.

Stop the chat room application if running, and use MQe_Explorer to load up the *ClientQm* queue manager.

From the expanded tree view, right click on the *Local queues* object, and then select the *New Queue* menu item. A window will appear, in which to enter the details of the queue you wish to define.

On the *General* tab, type 'YourCoQuery' into the *Name* field.

Change the queue type to one of 'Remote queue', by selecting that value from the drop down box in the *Type* field.

In the Queue qMgr box, select from the drop down box the name of the remote queue manager, in this case 'WASServerQm'.

Check the mode is 'Synchronous', by selecting that value from the drop down box in the *Mode* field.

Then click on the *Security* tab. In the *Authenticator* field type in this value:

```
itso.mqe.security.QueueAuthenticator
```

The screen should look similar to this:

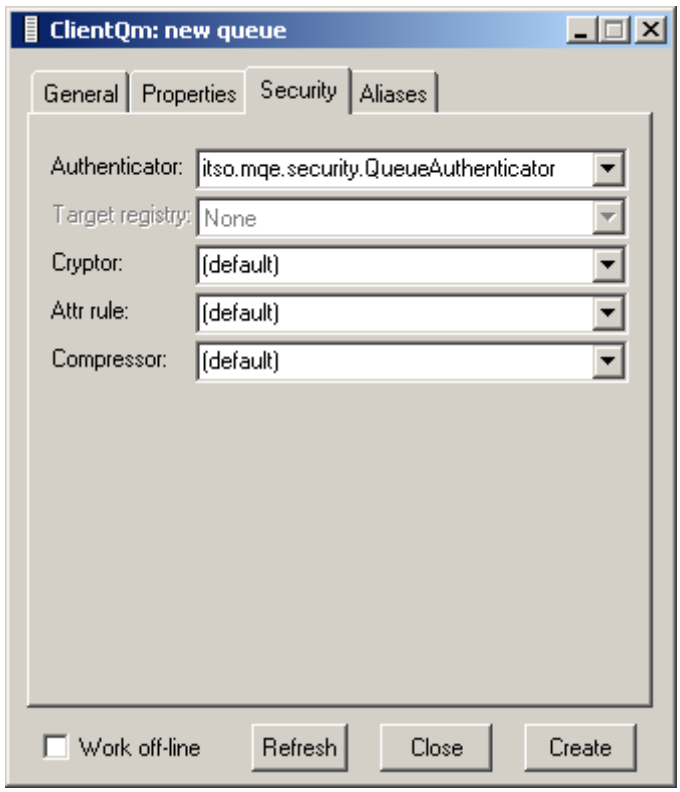

Figure 9-2 Specifying a customized authentication adapter

Then click the *Create* button. No confirmation window is displayed.

On WASServerQM: local queue – YourCoQuery

Stop the application server in WebSphere Application Server if the queue manager is active in the servlet. Then use MQe_Explorer to load up the *WASServerQm* queue manager.

From the expanded tree view, right click on the *Local queues* object, and then select the *New Queue* menu item. A window will appear, in which to enter the details of the queue you wish to define.

On the *General* tab, type 'YourCoQuery' into the *Name* field.

The queue type should be local, mode synchronous.

Then click on the *Security* tab. In the *Authenticator* field type in this value:

```
itso.mqe.security.QueueAuthenticator
```

Then click the *Create* button. No confirmation window is displayed.

9.5 Property file

Check that in the *MQe.property* file, the flag for the *YourCo* property is set to 'Yes'. The line in the property file should look like this:

```
YourCo=Yes
```

9.6 Additional beans

The YourCo application comes as supplied example with WebSphere Application Server. Additional beans and servlets were developed to demonstrate new functionality of WebSphere Everyplace Server. These additional beans are supplied with the *.zip* file for this SupportPac. They need to be copied to the directory containing the rest of the YourCo example.

This directory also needs to be added to the *classpath* in the Web Application definition in WebSphere Application Server:

C:\WebSphere\AppServer\hosts\default_host\WSsamples_app\servlets

These additional classes:

WebSphereSamples.YourCo.Timeout.TotalLeaveBean
WebSphereSamples.YourCo.Timeout.TotalLeaveServlet

need to be copied to this directory:

C:\WebSphere\AppServer\hosts\default_host\WSsamples_app\servlets\WebSphereSamples\YourCo\Timeout

This class:

WebSphereSamples.YourCo.Meeting.InviteesDBBean

needs to be copied to this directory:

C:\WebSphere\AppServer\hosts\default_host\WSsamples_app\servlets\WebSphereSamples\YourCo\Meeting

Refer to the installation instructions.

9.7 Running the YourCo example

To run this example, start the client side of the chat room application. Then click on the *YourCo Secure Query* button. You will then be prompted to enter a password before access to the *YourCoQuery* queue is allowed, the window is as shown below:

Figure 9-3 YourCoQuery password prompt

Enter the password ‘shazam’ into the password field and click on the *OK* button.

After a few seconds the reply message advising the total leave values by type will appear in the chat room window. The message will be similar to this:

From: WASServerQm : ChatClientQ msg: YourCo leave Totals: Vactional: 11 Personal: 14 Sick: 20

Example 9-6 YourCo query reply message

10 Integration with WebSphere Everyplace Suite

So far in this chapter, the examples of using MQSeries Everyplace with applications have been done on standard desktops using Windows 2000, over a standard LAN. However one of the reasons that MQSeries Everyplace was developed was to provide assured messaging capability over non-traditional networks, such as those now available for wireless connection.

The connectivity support provided by the Wireless Gateway component of WebSphere Everyplace Server, means that applications that use MQSeries Everyplace can use this connection support to allow them to run on wireless devices. While this chapter has only described using MQSeries Everyplace on Windows systems, the product does provide support for other devices such as palm type devices.

The Wireless Gateway consists of a server component that would typically be run within an organizations data center, and a client component installed on the wireless device. This client component handles the process of communicating with the server side of the Wireless Gateway.

One of the advantages of using the Wireless Gateway is that it can be configured to provide authentication and encryption services. Enabling authentication means that, when an end user establishes a wireless connection, the Wireless client will prompt them for their authentication details. Enabling encryption means that all data transferred between the client and the server is encrypted preventing unauthorized people from viewing the data.

These authentication and encryption services of the Wireless Gateway can be in addition to any authentication and encryption that applications or other products may use, that are communicating via the Wireless Gateway.

Applications using MQSeries Everyplace, when run on a device using the Wireless Gateway for handling the communication, require no modifications. Applications using MQSeries Everyplace do not handle any part of the communication process; rather MQSeries Everyplace does this. Additionally MQSeries Everyplace configuration does not require any special configuration to use the Wireless Client support.

An advantage that MQSeries Everyplace provides is that it can send messages using the HTTP protocol as well as via TCP/IP. This means that a site that has the typical firewall setup to allow in HTTP traffic through to back end web servers, does not need to change this setup to allow applications on wireless devices access into the system. Since the packets of data from the MQSeries Everyplace applications will be standard HTTP packets, coming in on the standard HTTP port 80, the firewall will not require modifications.

By using client type queue managers on the wireless device, MQSeries Everyplace will only be establishing connections from the outside world into the organizations site. No connections are established from within the organization site to devices outside.

WebTraffic Express

As MQSeries Everyplace can use the standard HTTP protocol, those requests that will invoke a servlet in WebSphere Application Server, can be initially routed to the WebTraffic Express component of WebSphere Everyplace Server. WebTraffic Express can then use the WebSeal Lite plugin, to verify with Policy Director if the request should be allowed to pass through to the backend server.

A sample authentication adapter is shipped with MQSeries Everyplace, which can be used to add to the HTTP request an authentication header, containing the user id and password of the end user. The user id and password are encoded using a base64 algorithm, just as is done in browsers, meaning that it is a trivial task to decrypt it. However enabling the encryption support of the Wireless Gateway means that the HTTP requests sent by MQSeries Everyplace would be encrypted using a much stronger algorithm, which protects the user id and password in the HTTP request.

The following diagram shows how the Chat Room application would be implemented across multiple devices with the Wireless Gateway handling the communication between the client and other queue managers.

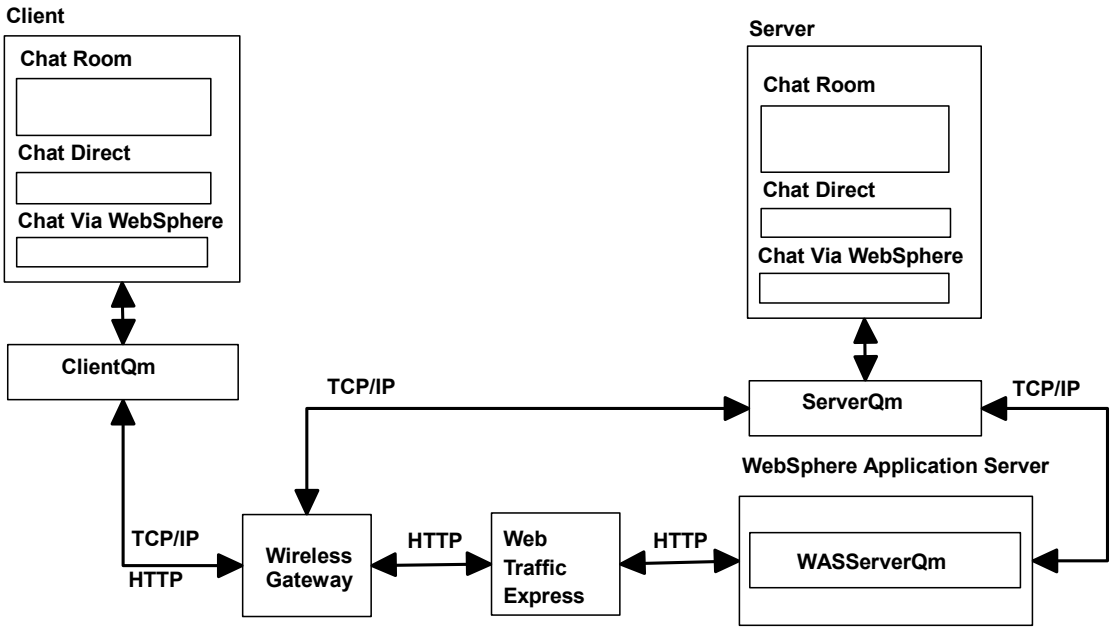

Figure 10-1 Integration with the Wireless Gateway

The IBM Redbook, *Enterprise Wireless Applications using IBM WebSphere Everyplace Server Service Provider and Enable Offerings*, SG24-6519, describes another feature of WebSphere Everyplace Suite, called Location Based Services. Location Based Services allows information about the location of the end user to be added to the HTTP request when received on the server side. Since MQSeries Everyplace could be configured to send it messages as HTTP requests, these requests could be routed through the system running the Location Based Services component.

Location information would be added to the HTTP request, which would then flow onto the servlet in WebSphere Application Server. The servlet, as well as passing the message to MQSeries Everyplace, could extract the location information and use this as required.

By using client type queue managers on the wireless device, MQSeries Everyplace will only be establishing connections from the outside world into the organizations site. No connections are established from within the organization site to devices outside.

10.1 Using the Wireless Client and Gateway

This section demonstrates how to use the Wireless Gateway in conjunction with MQSeries Everyplace.

The following diagram shows how we initially configured the chat room application to run across two Windows 2000 machines.

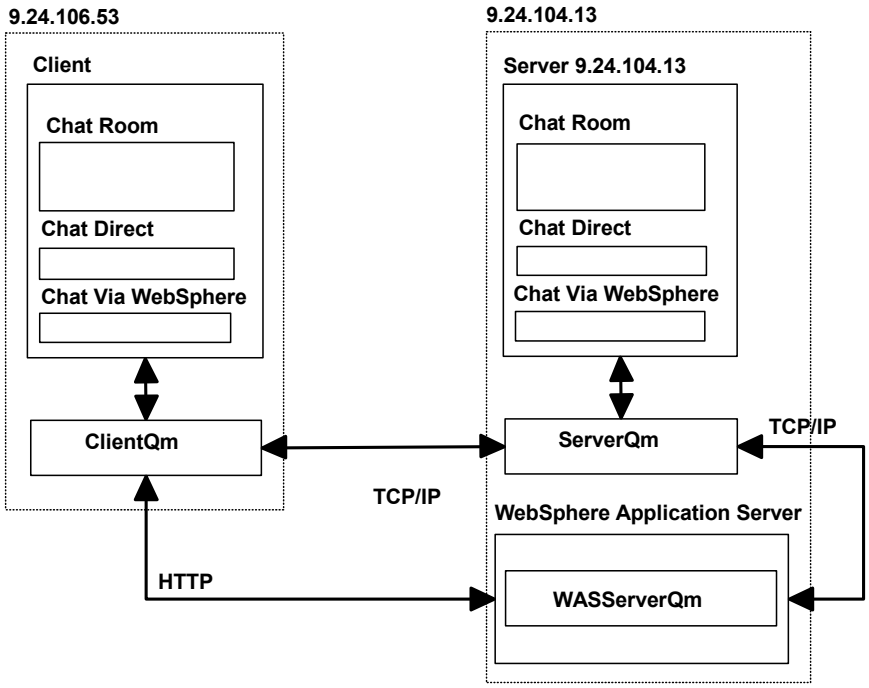

Figure 10-2 Chat room application over a standard LAN

The above diagram shows the client side running on a PC at address 9.24.106.53, while the server queue manager and WebSphere Application Server both run on the PC at address 9.24.104.13.

We now want to use the Wireless Gateway to handle communication between these two devices.

The Wireless gateway was setup on an AIX system, and the Wireless client installed on the client-side PC.

The Wireless Gatekeeper is the tool used to administer the Wireless Gateway. In the lab, we had the Wireless Gateway running on an AIX system. A sample screen shot is shown below.

Figure 10-3 Wireless gatekeeper GUI

On an AIX system, the gatekeeper is started by typing in ‘wgcfg’.

On the left hand side of the gatekeeper window is a tree structure showing the various objects being managed by the gateway. In the tree, the object labeled RS615001 represents the AIX system running the Wireless Gateway. Under this object are two objects relating to the Wireless gateway.

Mobile network connection

The first is an icon of a connection with a lightening bolt. This icon represents a Mobile Network Connection or MNC. This represents the interface to a network provider for the Wireless Gateway. Right click on this icon, and select properties. The right hand side of the window displays its associated properties. For this example we do not want to use the authorization facility of the Wireless gateway. To set this level of authentication, click on the tab labeled 'Security'. Then click on the radio button corresponding to 'No validation', then click on the *Apply* button to effect the change. The screen will look similar to this:

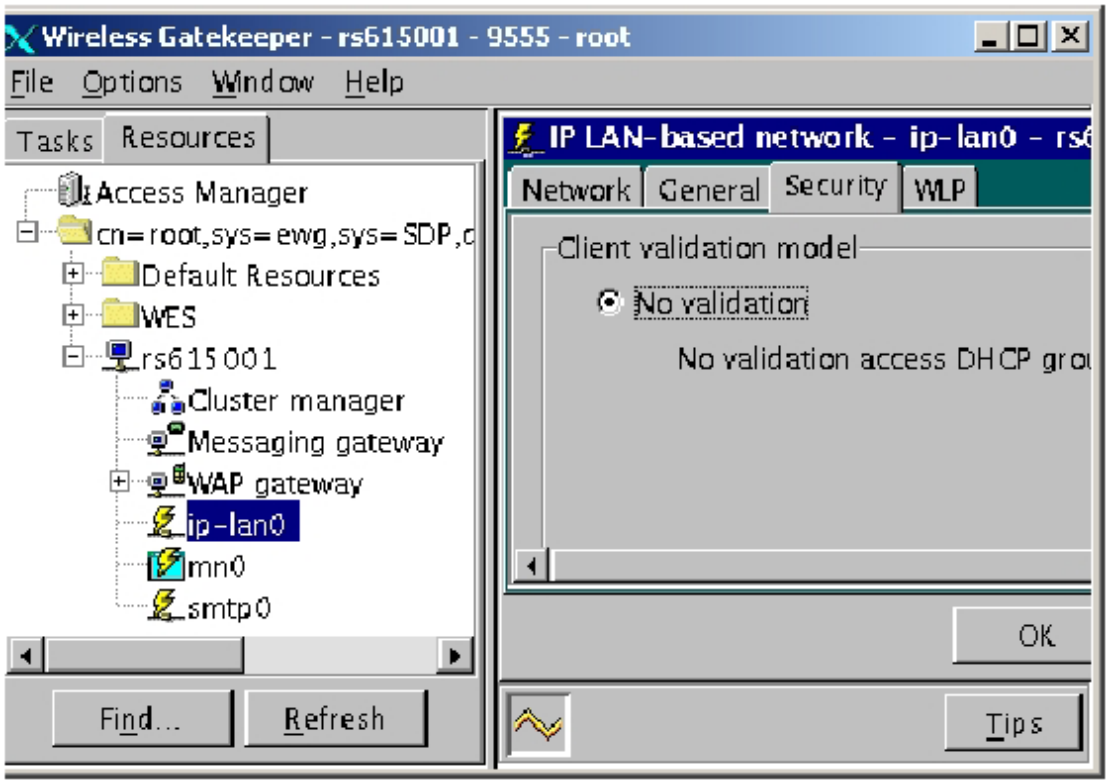

Figure 10-4 Selecting no authentication

Mobile network interface

The next object of interest is the one that appears as a light blue icon with a lightening bolt through it. This icon represents the Mobile Network Interface, or MNI. This interface defines an IP subnet, through which the Wireless Gateway routes traffic for Wireless Clients. When a device connects using the Wireless client, it will be allocated an address from this subnet. Right click on this icon, and select properties. The right hand side of the Gatekeeper window will then display the properties on the MNI. Click on the tab labeled 'Interface'. The field labeled 'IP address' is where you specify the IP subnet that will be used to support the clients.

Configuring this IP subnet correctly is a key issue when setting up a wireless gateway.

For our example we set up a virtual IP subnet at address 10.0.0.1. Thus the value entered in the 'IP Address' field in our case was '10.0.0.1'. The gatekeeper screen looked like this:

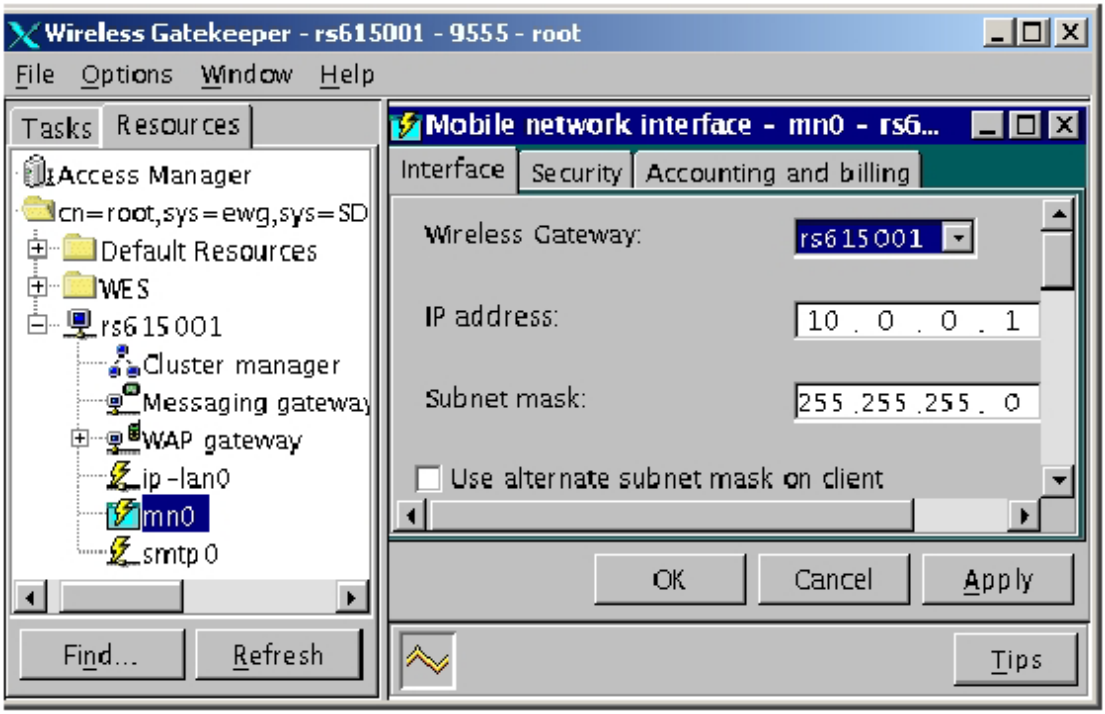

Figure 10-5 Configuring the Wireless Gateway client

Our configuration with the Wireless Gateway incorporated now looks as shown below:

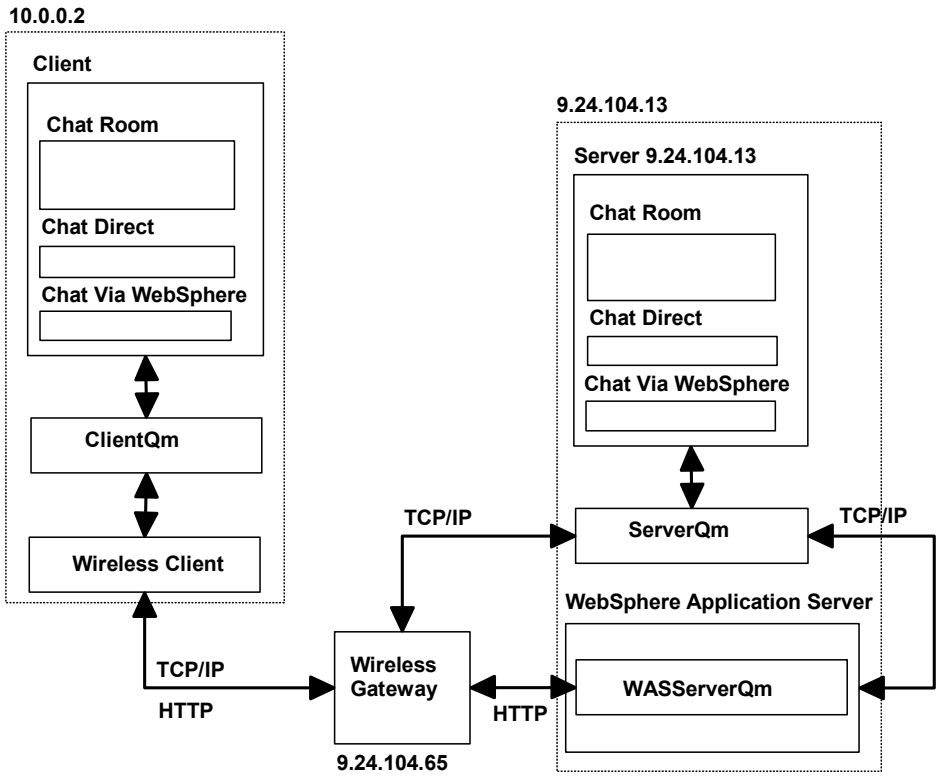

Figure 10-6 Incorporating the Wireless Gateway

Note that the client now has an IP address of 10.0.0.2. This is the IP address it has been allocated by the Wireless Gateway when it connects using the Wireless Client. If it reconnected at a later time, the address may change, for example to 10.0.0.5.

In this case we are running the wireless protocol over the LAN to demonstrate the use of the Wireless Gateway and client.

IP packets can now flow from the client through the gateway to the 9.24.104.13 machine. However an entry needs to be added to the route table on the 9.24.104.13 machine so that it knows where to send reply packets destined for the client address at 10.0.0.2. On the Windows 2000 machine at 9.24.104.13, open a DOS window and enter this command:

```
route ADD 10.0.0.0 MASK 255.255.255.0 9.24.104.65
```

This adds a temporary TCP/IP routing entry, that tells that system, to route packets destined for 10.0.0.* to 9.24.104.65, which is the Wireless Gateway.

Note this setup is for example purposes only. A production implementation would require a proper IP subnet and routing tables to be configured.

10.2 Trying out the Wireless Gateway

First set up the chat room application on two machines as depicted in *Figure 10-2 Chat room application over a standard LAN* on page 84 and ensure the application is working normally.

Then stop the chat room application on the client side. This must be done before starting the Wireless Client.

Install the Wireless Client software on the Windows 2000 machine on the client side. This is a straightforward process.

Then from the *Start* button, find the IBM Wireless Client, and select ‘Connections’. In the window displayed, create a connection definition if one has not already been defined. This creation process is straightforward. The most important item to know is the IP address of the Wireless Gateway that needs to be entered.

In the lab we created a connection called `France', as shown below:

Figure 10-7 Wireless connections

To establish a wireless connection, just double click on the *Gateway to France* icon. A window similar to the one shown below appears:

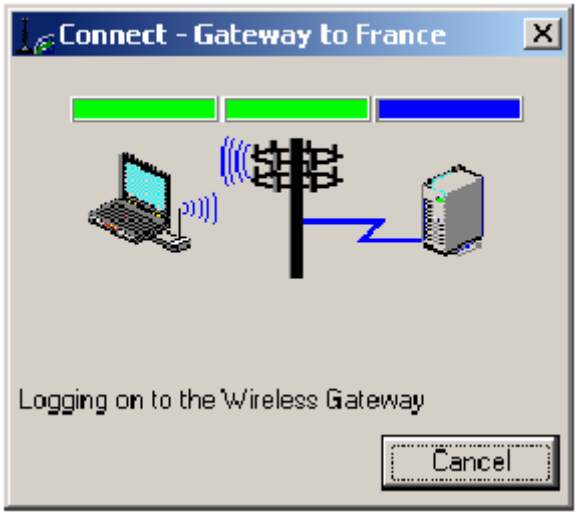

Figure 10-8 Connecting to the Wireless Gateway

When the three boxes all turn green, the connection is established, and the window disappears. On the right hand side of the Windows 2000 task bar, appears a small icon of a transmission tower.

To test your wireless connection is working open a DOS window. Type in 'IPCONFIG' and you will get a display similar to this:

```
C:\>ipconfig

Windows 2000 IP Configuration

Ethernet adapter {21959871-44F7-46A7-BE57-6501A133852C}:

Connection-specific DNS Suffix .        :
IP Address. . . . . . . . . . . .       : 10.0.0.3
Subnet Mask . . . . . . . . . . .       : 255.255.255.0
Default Gateway . . . . . . . . .       : 10.0.0.1

Token Ring adapter Local Area Connection:

Connection-specific DNS Suffix .        : itso.ral.ibm.com
IP Address. . . . . . . . . . . .       : 9.24.106.53
Subnet Mask . . . . . . . . . . .       : 255.255.255.0
Default Gateway . . . . . . . . .       :
```

Example 10-1 IP status

This output shows that you still have the LAN connection, but now have also been allocated a new IP address of 10.0.0.3, which is your wireless connection.

Then PING the address of the machine running your server-side queue managers. Watch the little transmission tower icon, you will see a lightening bolt flash to indicate that IP traffic is being sent.

Restart the client side of the chat room application, and send some messages and it should function as before.

This example demonstrates that MQSeries Everyplace applications can be deployed to run on wireless devices, with the Wireless Gateway providing the communication support.

10.3 Tracing

On the client

The Wireless client provides a tracing capability to assist with resolving communications problems. Right click on the small transmission tower icon, and select `Trace'. A window allowing you to set trace options appears. You can set trace to various levels as required.

Trace information is written to a file *arttrace.txt* which is located in the directory where the Wireless client was installed, which is a default install is *C:\Program Files\IBM\Wireless Client*.

This trace file contains formatted trace output, showing IP traffic that has occurred.

On the server

Various levels of logging and tracing can be enabled in the Wireless Gateway. Using the Wireless Gatekeeper does this. Right click on the icon representing the AIX system you wish to set logging for, select ‘Properties’, then in the panel on the right, click on the *Logging* tab.

This tab shows the logging and trace file names, and the level of logging and tracing that is active. These values can be adjusted as required.

11 OS/390

The OS/390 platform also provides excellent Java support, so we decided to try out MQSeries Everyplace on OS/390. We had access to an OS/390 system running Z/OS V1. We were able to successfully run the chat room application using MQSeries Everyplace on the OS/390[5] system. This section describes how this was done.

We only had time to set up the environment on OS/390 to allow messages to be sent from the client chat room to the server chat room. However it would only require the appropriate definitions to allow the server side to function fully. No Java programs required any modification or recompilation.

11.1 Requirements

OS/390 Z/OS contains an Open/Edition environment that provides a UNIX environment. A Windows server is required to allow the Open/Edition (Unix) Services to display the GUI window of the chat room application when it is run.

We installed a X-Windows server onto a Windows 2000 desktop. Then we opened a Telnet session to the OS/390 system, and issued this command to set the address of the machine to display the GUI on:

```
export DISPLAY=9.24.106.53:0.0
```

The MQSeries Everyplace product is shipped as a *.zip* file. There is no supplied facility in Open/Edition to unzip a zip file. However the Infozip product has been ported to run on OS/390 Open/Edition. It can be downloaded from here:

http://www-1.ibm.com/servers/eserver/zseries/zos/unix/bpxa1ty1.html

We downloaded this file, ftp'd it to Open/Edition in OS/390, then installed it, which involved simply un-tarring it.

This then provided us a way to unzip any *.zip* files we created on the Windows platform.

We then used WinZip to zip up the directory containing the MQSeries Everyplace product, ftp'd this file to the OpenEdition environment and unzipped it.

Then we zipped up the directory containing the chat room application packages, ftp'd this to Open/Edition and unzipped it.

Note: Like all Unix environments, Open/Edition is case sensitive, thus it is very important to ensure that case of the directory names matches the package names coded in the java programs.

After this transfer process was complete, the MQSeries Everyplace product was located at */u/.edward/mqe3/MQe*, while the Chat Room application was located at */u/edward/itso*.

11.2 Classpath

To be able to define the queue manager and run the application (based on where we had unzipped files to, as mentioned above) we set *classpath* using this command:

```
export CLASSPATH=/u/edward/mqe3/MQe/Java:/u/edward:.
```

[5] At the time of writing IBM was determining licensing issues with MQSeries Everyplace on OS/390 and its use is restricted at this time. Apply to IBM if you wish to deploy on this platform.

Also the *PATH* environment variable should reference the location of the Java executable, for example on our system we set the *PATH* as follows:

```
export PATH=/usr/lpp/java213/J1.3/bin
```

11.3 Configure ServerQm

The MQe_Explorer tool cannot be run from OS/390, though it can remotely administer such queue managers after they have been created. However the MQSeries Everyplace product comes with many example Java programs that can be used from a command line to perform queue manager administration tasks. We used these tools to setup the *ServerQm* on OS/390.

Note we only set up the server side to allow the client chat window to send messages to the server side.

Create .ini file

Firstly need to create the *.ini* file defining the initialization parameters for the *ServerQm* queue manager. Due to the way the supplied samples have been written, they expect that the *.ini* file will be in ASCII. While you can store the *.ini* file in ASCII in Open/Edition, you cannot edit it there.

Thus use notepad to code up the *.ini* file, then do a binary transfer of this file to the OpenEdition environment. We placed the *.ini* file in a directory called */u/edward/os390*.

The *.ini* file is shown below:

```
[Registry]
(ascii)LocalRegType=FileRegistry
(ascii)DirName=/u/edward/os390/ServerQm/Registry/
(ascii)Adapter=RegistryAdapter
[ChannelManager]
(int)MaxChannels=0
[QueueManager]
(ascii)Name=ServerQm
[Listener]
(int)TimeInterval=300
(ascii)Listen=FastNetwork::8082
(ascii)Network=FastNetwork:
[Alias]
(ascii)QueueManager=com.ibm.mqe.MQeQueueManager
(ascii)DefaultTransporter=com.ibm.mqe.MQeTransporter
(ascii)RegistryAdapter=com.ibm.mqe.adapters.MQeDiskFieldsAdapter
(ascii)MsgLog=com.ibm.mqe.adapters.MQeDiskFieldsAdapter
(ascii)PrivateRegistry=com.ibm.mqe.registry.MQePrivateSession
(ascii)FastNetwork=com.ibm.mqe.adapters.MQeTcpipHistoryAdapter
(ascii)FileRegistry=com.ibm.mqe.registry.MQeFileSession
(ascii)Server=examples.queuemanager.MQeServer
(ascii)ChannelAttrRules=examples.rules.AttributeRule
(ascii)Admin=examples.administration.console.Admin
(ascii)AttributeKey_2=com.ibm.mqe.attributes.MQeSharedKey
(ascii)AttributeKey_1=com.ibm.mqe.MQeKey
(ascii)DefaultChannel=com.ibm.mqe.MQeChannel
(ascii)Network=com.ibm.mqe.adapters.MQeTcpipHttpAdapter
```

Example 11-1 ServerQm .ini file for OS/390

Create ServerQm queue manager

We then issued this command to create the ServerQM queue manager:

```
java examples.install.SimpleCreateQM /u/edward/os390/ServerQm.ini
```

Basic test of ServerQm

We then ran of the supplied examples just to test the queue manager could be run successfully by issuing this command:

```
java examples.application.Example1 ServerQm /u/edward/os390/ServerQm.ini
```

Add chat room application

We then added the chat room application to the *.ini* file, and ftp'd that to Open/Edition. The lines added to the bottom of the *.ini* file are shown below:

```
[AppRunList]
(ascii)App1=itso.mqe.chatserver.RoomMgr
[App1]
(ascii)ClientQueue=ChatClientQ
(ascii)ChatRoomQ=ChatRoomQ
```

Example 11-2 Adding the chat room application to the .ini file

Define ChatRoomQ

We issued this command to define the local queue, *ChatRoomQ*, to the *ServerQm* queue manager:

```
java examples.administration.commandline.LocalQueueCreator ChatRoomQ null null null nolimit
nolimit ServerQm /u/edward/os390/ServerQm.ini
com.ibm.mqe.adapters.MQeDiskFieldsAdapter:/u/edward/os390/ServerQm
```

11.4 Modify ClientQm

We then used the MQe_Explorer to modify the connection definition to *ServerQm* in *ClientQm*. We changed the IP address to the IP address of the OS/390 system, then shutdown MQe_Explorer.

11.5 Start chat room on OS/390

We then started the *ServerQm* queue manager on OS/390 by issuing this command:

```
java examples.queuemanager.MQeServer /u/edward/os390/ServerQm.ini
```

Note, prior to doing this be sure you have set the *DISPLAY* environment variable in your Open/Edition telnet session, and that you have the X-Windows server running on the system were you want the GUI window to appear.

Once the above command was issued, the server-side GUI window of the chat room application appeared on our Windows desktop.

Start client side of chat room

We then started the client side of the chat room application, as explained in *Starting the chat room application* on page 65. The client side GUI window appeared. We then typed a message in on the client side, and it duly appeared in the server chat room window.

End of Document