WebSphere MQ Everyplace

Configuration Guide

Version 2.0.0.4

SC34-6283-01

WebSphere MQ Everyplace

Configuration Guide

Version 2.0.0.4

SC34-6283-01

Second Edition (April 2003)

This edition applies to WebSphere MQ Everyplace Version 2.0.0.4 (Program number: 5724-C77) and to all subsequent releases and modifications until otherwise indicated in new editions.

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

Contents

Figures

Tables

About this book

This book is a configuration guide for the WebSphere MQ Everyplace product. It contains information on how WebSphere MQ Everyplace can be set up to provide a specific configuration matching a user's business requirements. In many cases, example code is supplied.

The book is broadly divided into three parts:

- An introductory section provides an overview of how to use this guide and describes the basics of system administration using administration messages and the C administration API
- The creation and administration of the fundamental components of a WebSphere MQ Everyplace solution are described: queue managers, queues, connection adapters, listeners and bridge resources
- Advanced options are discussed, such as how to configure your system for security or how to enhance performance, as well as the intricacies of platform-specific configuration issues

This book is intended to be used in conjunction with:

- WebSphere® MQ Everyplace™ Introduction, SC34-6277-01
- WebSphere MQ Everyplace Application Programming Guide, SC34-6278-01
- WebSphere MQ Everyplace System Programming Guide, SC34-6274-01

The relevant books are available in softcopy form from the Book section of the online WebSphere MQ library. This can be reached from the WebSphere MQ Web site:

`http://www.ibm.com/software/mqseries/library/manualsa`

This document is continually being updated with new and improved information. For the latest edition, please see the MQSeries® family library web page at the Web site indicated above.

Who should read this book

This book is intended for anyone who wants to configure a solution using WebSphere MQ Everyplace systems and other members of the WebSphere MQ family of messaging and queueing products.

Prerequisite knowledge

This documentation assumes that the reader has an understanding of WebSphere MQ Everyplace as described in WebSphere MQ Everyplace Introduction, SC34-6277-01.

An initial understanding of the concepts of secure messaging is also required. If you do not have this understanding, you may find it useful to read the following WebSphere MQ book: WebSphere MQ An Introduction to Messaging and Queuing, GC33-0805-01.

This book is available in softcopy form from the online WebSphere MQ library:
http://www.ibm.com/software/mqseries/library/manualsa

Chapter 1. Introduction

This book provides the basic information necessary in order to configure WebSphere MQ Everyplace queue managers and networks. It is also designed to allow a user to customize a configuration matching his or her specific business requirements. It describes how individual WebSphere MQ Everyplace components can be created and administered and how components may be used together in various topologies.

The contents include information on:

- Creating and starting queue managers
- Defining connectivity between queue managers
- Establishing the routes taken by messages through a WebSphere MQ Everyplace network
- Exercising control over the protocols used
- Determining where messages are staged, if appropriate
- Configuring queue-level security
- Appreciating the advantages and disadvantages of the available WebSphere MQ Everyplace configuration options

This introduction provides a map of various routes through the rest of the guide depending on the type of configuration which the user hopes to achieve. Since these routes are described in terms of queue manager configurations, a brief description of the WebSphere MQ Everyplace queue manager and associated components follows.

Queue manager

A queue manager owns and controls WebSphere MQ Everyplace messages, queues, and connections (see below). It allows applications to access messages and queues. Each queue manager has a unique name that distinguishes it from any other WebSphere MQ Everyplace queue manager. Depending upon the needs of an application, queue managers can differ in their collection of queues, messages, connections, and other objects, and also in the role they play in a configuration.

WebSphere MQ Everyplace identifies three distinct roles for queue managers in addition to the basic queue manager functionality:

- **Client** A queue manager that supplies messages to, or gets messages from, a server
- **Server** A queue manager that provides services to many attached client queue managers
- **Gateway** A server queue manager that also has the capability to exchange messages with WebSphere MQ base messaging queue managers

Queue

A queue may be used to store, process, or move messages. Each queue belongs to a queue manager and applications can access queues through the queue manager. Each

queue has a unique name that distinguishes it from any other queue on that same queue manager. Local queues are not strictly mandatory, however you cannot do much without them.

Message

A message is a collection of data which can be stored in a queue or moved across a WebSphere MQ Everyplace network.

Connection

A connection provides its local queue manager with the information it needs to establish communication links with a remote queue manager. The name of a connection is the name of that remote queue manager. Only one connection definition can exist on a local queue manager for each remote queue manager name.

Channel

A channel is an entity allowing a queue manager to move messages to a remote queue manager.

Registry

The registry is the primary store for queue manager-related information. Each queue manager has its own registry. Every queue manager uses the registry to hold details of its properties and objects.

Queue manager configuration

No matter what role a queue manager performs, there is a basic amount of configuration required. This basic configuration results in what is here termed a 'Basic Queue Manager'. Depending upon the type of role intended for the queue manager, this Basic Queue Manager is extended, resulting in a Client Queue Manager, a Server Queue Manager or a Gateway Queue Manager. The following diagram attempts to summarize these configurations:

Table 1. Queue manager configuration

Basic Queue Manager	+	Connection definition and remote queue definition	=	Client queue manager
Basic Queue Manager	+	Listener	=	Server queue manager
Basic Queue Manager	+	Bridge functionality	=	Gateway queue manager
Basic Queue Manager	+	Security configuration, and so on		

In the following table, the necessary steps to configure each type of queue manager are itemized, together with the corresponding chapters of this manual. The Basic Queue

Manager configuration is a prerequisite of all other configurations; that is to say, any queue manager must first be configured as a Basic Queue Manager. Then, other types of functionality may be added as required.

Thus, to configure a Client, perform steps 1, 2, 3, 4 and 5; to configure a Server, perform steps 1, 2, 6 and 7; to configure a queue manager with both Server and Client functionality, perform steps 1 through 7 inclusive.

Table 2. Configuring clients, servers, and queue managers

Requisite steps	Chapter or chapters
Basic queue manager	
1. Create and start the queue manager	2: Administration using admin messages 3: Administration using admin API 4: Creating and starting queue managers
2. Create a local queue	5: Administering queue managers 6: Administering local queues
Client queue manager	
3. Create a connection definition to a server	10: Connection definition
4. Create a remote queue definition	7: Administering remote queues
5. Create a home server queue for triggered transmission (required for remote asynchronous queues)	8: Administering home server queues
Server queue manager	
6. Create a listener	11: Listener
7. Create a store-and-forward queue (optional)	9: Administering store-and-forward Queues
8. Add bridge functionality	12: Administering bridge resources

Chapters 13 through 18 provide additional configuration options: for advanced message routing, security, performance and platform specifics.

This book is not an application programming guide. It describes what a user needs to set up in order to configure a WebSphere MQ Everyplace system and the steps which must be followed in doing this. The specifics of coding practices and APIs are covered in the WebSphere MQ Everyplace Application Programming Guide, SC34-6278-01 and WebSphere MQ Everyplace System Programming Guide SC34-6274-01.

Chapter 2. Administration using administration messages

You can administer WebSphere MQ Everyplace resources using specialized messages called administration messages (admin messages). Using these messages allows you to administer resources locally or remotely. The native codebase, if configured with an administration queue (admin queue) responds to admin messages. However, it does not provide helper functions to create admin messages. For more information on this, refer to Chapter 4, “Administration using the administrator API”, on page 55. Java™ is administered by admin messages. C can be, but has an administration interface for local administration.

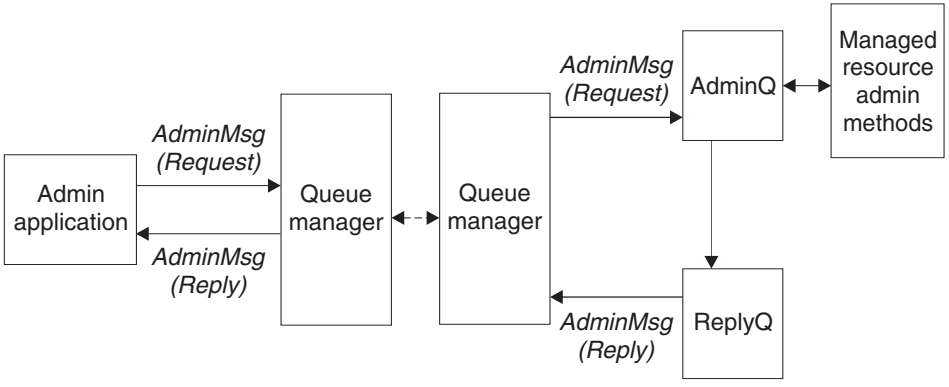

Figure 1. WebSphere MQ Everyplace administration using administration messages

The sequence of steps required in administering a resource using administration messages is as follows:

1. Create an admin queue on the resource performing the administration, or make sure that one exists.
2. Create an appropriate admin message for the resource being managed.
3. Set the required fields in the message.
4. Put the admin message to the appropriate admin queue.
5. Wait for an admin reply message on the appropriate admin reply queue, if a reply has been requested in the admin message.
6. Analyze the data in the admin reply message.

The administration queue

Before you can administer a queue manager or its resources using admin messages, you must start the queue manager and configure an admin queue on it. The admin queue’s role is to process admin messages in the sequence that they arrive on the queue. Only one request is processed at a time.

Java

In Java, the queue can be created using the `defineDefaultAdminQueue()` method of the MQeQueueManagerConfigure class. The name of the queue is `AdminQ` and applications can refer to it using the constant `MQe.Admin_Queue_Name`.

C

In the native codebase, an admin queue is created using the following API:

```
MQeAdminQParms params = ADMIN_Q_INIT_VAL;
rc = mqeAdministrator_AdminQueue_create(hAdmin, // handle to MQeAdministrator
           pExceptBlock, // handle to an exception block
           hQueueName, // the name of the queue to be created
           hQueueQMgrName, // the name of the queue's
                 //owning queue manager
           &params); // pointer to structure
                 // for configuring the
                 // queue of type MQeAdminQParms,
```

In particular, the constant string handle `MQE_ADMIN_QUEUE_NAME` can be used as the admin queue name. This is the equivalent of the constant `MQe.Admin_Queue_Name` in the Java codebase.

The `params` structure can be initialized to contain default values for all admin queue properties. The structure also contains an opFlags bit mask element that must be used to indicate which properties have been set to a value other than the default value. The above example accepts all of the default values, as specified using the `ADMIN_Q_INIT_VAL` constant.

The administration reply-to queue

This section describes the use of administration reply-to queues in Java and C.

Java

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

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

C

In the native codebase, as in the Java codebase, any queue can be specified as the admin reply-to queue. However, it is recommended that the default admin reply-to

queue name, MQE_ADMIN_REPLY_QUEUE_NAME, is used to name a queue dedicated to the role of admin reply-to queue. This name corresponds to MQe.Admin_Reply_Queue_Name in the Java codebase.

In practice, the native client is more likely to be receiving than to be sending admin messages. In this case, the client needs a remote asynchronous queue definition of the admin reply-to queue on the server as well as a home server queue matching a store-and-forward queue on the server to enable the admin and admin reply messages to be transferred.

Create the appropriate administration message

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

Java

In Java, there is a hierarchy of administration message types. For certain operations, the exact type of administration message is required, for example, to create a Home Server 'queue' you need a Home Server Queue administration message. For other operations, a more general administration message is appropriate, for example, to enquire upon a home server queue, you can use a queue administration message, or a remote queue administration message. If in doubt, use the exact type of administration message.

The following messages are provided for administration of WebSphere MQ Everyplace resources:

Table 3. Administration messages

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

Table 3. Administration messages (continued)

Message name	Purpose
MQeCommunicationsListenerAdminMsg	provides support for administering communications listeners

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

C

In the C codebase, all messages are MQeFields instances. This applies to admin messages and the admin message types are distinguished by a special field inserted into the fields object. The user has to create an admin message of the appropriate type from scratch, inserting all of the required fields described later in this chapter. Alternatively, for local administration, use the native administration API (see chapter 3). The native codebase can respond correctly to all administration messages but the native administration API is usually used for local administration. For these reasons, most of the examples in this chapter relate to the Java codebase only.

Set the required fields in a message

Administration messages convey the administration action required by a combination of data fields stored in the message. These fields have well defined names, types, and values, and you can set up the administration message using low level fields API. In Java, there are numerous helper methods to make this task less arduous.

The following sections describe the constituent fields of admin messages and admin reply messages.

The basic administration request message

Every request to administer an WebSphere MQ Everyplace resource takes the same basic form. Figure 2 on page 9 shows the basic structure for all administration request messages:

A request is made up of:

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

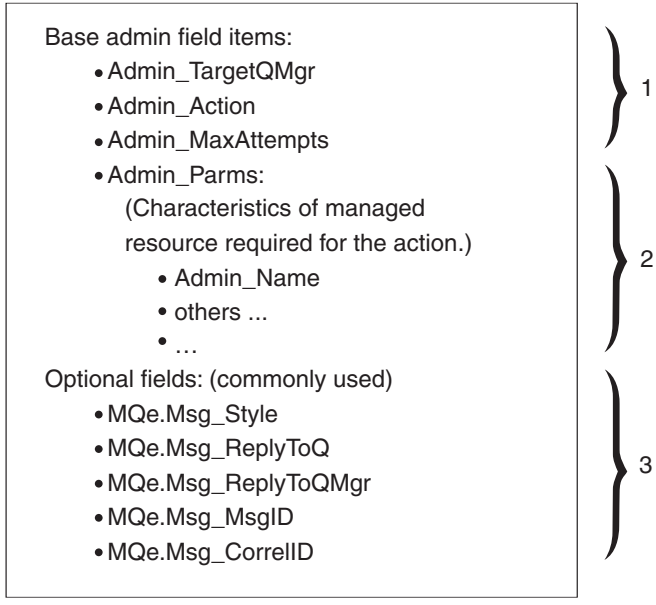

Figure 2. Administration request message

Base administration fields

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

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

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

Table 4. Administration actions

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

Table 4. Administration actions (continued)

Administration action	Purpose
Action_InquireAll	Inquire on all characteristics of a managed resource
Action_Update	Update one or more characteristics of a managed resource

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

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

Table 5. Setting the administration action field

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

Admin_MaxAttempts

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

Other fields

For most failures further information is available in the reply message. It is the responsibility of the requesting application to read and handle failure information. See “The basic administration reply message” on page 17 for more details on using the reply data.

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

Table 6. Setting administration request fields

Administration action	Field type	Set and get methods
Admin_Parms	MQeFields	MQeFields getInputFields()
Admin_Action	int	setAction (int action)
Admin_TargetQMgr	ASCII	setTargetQMgr(String qmgr)
Admin_MaxAttempts	int	setMaxAttempts(int attempts)

Fields specific to the managed resource

Admin_Parms

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

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

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

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

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

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

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

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

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

Other useful fields

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

Msg_Style

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

Msg_Style_Datagram
A command not requiring a reply

Msg_Style_Request
A request that would like a reply

Msg_Style_Reply
A reply to a request

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

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

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

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

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

Msg_MsgID
A byte array containing the message ID

Msg_CorrelID
A byte array containing the Correl ID of the message

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

Java

As this is a frequently performed process, this code example combines each step in the primeAdminMsg() method, that can be invoked in other chapters throughout this book (assuming that the method has been defined for the class in question).

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

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

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

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

  /*
   * Ensure matching response message is retrieved
   * - set up a fields object that can be used as a match parameter
   *   when searching and retrieving messages.
   */
  MQeFields msgTest = new MQeFields();
  msgTest.putArrayOfByte( MQe.Msg_CorrelID, new Byte{1, 2, 3, 4} );

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

Depending on how the destination administration queue is defined, delivery of the message can be either synchronous or asynchronous.

The next example is used to make an 'inquire all' on a queue manager. This method performs the steps required to address the admin message, request a reply, and add a unique marker to the message.

```
/* This method performs standard processing */
/* that primes an administration message so that */
/* we can handle it in a standard way */
/* This method sets the target queue manager */
/* (the queue manager upon which the admin */
/* action takes place. */
/* Requests that a reply message is sent to the */
/* admin reply queue on *the target queue manager. */
/* Incorporates a unique key in the message that */
/* can be used to retrieve the reply for this message.*/
/* The unique key is returned as a string,to be */
/* used by the routine extracting the reply. */

public static final String decorateAdminMsg(MQeAdminMsg msg,
                           String targetQMName)throws Exception {
    //set the target queue manager
    msg.setTargetQMgr(targetQMName);
```

```
    //indicate that we require a reply message
    msg.putInt(MQe.Msg_Style,MQe.Msg_Style_Request);
    //use default reply-to queue on the target queue manager.
    msg.putAscii(MQe.Msg_ReplyToQ,MQe.Admin_Reply_Queue_Name);
    msg.putAscii(MQe.Msg_ReplyToQMgr,targetQMName);
    //create a unique tag that we can identify the reply with
    String match ="Msg"+System.currentTimeMillis();
    msg.putArrayOfByte(MQe.Msg_CorrelID,match.getBytes());
    return match;
}
```

Put the administration message onto the target administration queue

The action defined in the admin message will only be performed when the message reaches the admin queue on the target queue manager. The target queue manager will need to have an admin queue. To get the message to a remote target queue manager, you will need to have all the appropriate connectivity in place. If the administration is to be done on the local queue manager, no connectivity is required. Message delivery is achieved by a simple put message call. Simply use the MQeQueueManager API call putMessage(), specifying the destination queue manager and the standard admin queue name. We can ignore the attribute, and confirmed parameters in our example, though they are available for more controlled access to the admin queue.

Java

```
//put the message to the right admin queue
LocalQueueManager.putMessage(targetQueueManagerName, MQe.Admin_Queue_Name,
                             msg,null,0L);
```

Wait for an administration reply message

Since administration is performed asynchronously, you will have to wait for the reply to the admin message in order to determine if the action was successful. Standard WebSphere MQ Everyplace message processing is used to wait for a reply or notification of a reply. In the Java codebase, for instance, the queue manager API call `waitForMessage()` can be used for this purpose.

There is a time lag between sending the request and receiving the reply message. The time lag may be small if the request is being processed locally or may be long if both the request and reply messages are delivered asynchronously. The following Java code fragment could be used to send a request message and wait for a reply:

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

/*
 * Send a completed admin message.
 *   Uses the simple putMessage method which is not assured if the
 *   the queue is defined for synchronous operation.
```

```
 */
public void sendRequest( MQeAdminMsg msg ) throws Exception
{
  myQM.putMessage( targetQMgr,
                   MQe.Admin_Queue_Name,
                   msg,
                   null,
                   0L );
}

/*
 * Wait ten seconds for a reply message. This method will wait for
 * a limited time on either a local or a remote reply to queue.
 *
 *
 */
public MQeAdminMsg waitForReply(MQeFields msgTest) throws Exception {
 int secondsElapsed = 0;
 MQeAdminMsg msg = null;
 try {
  msg = (MQeAdminMsg)myQM.getMessage(
         targetQMgr,
         MQe.Admin_Reply_Queue_Name,
         msgTest, null, 0L);
 } catch (MQeException e) {
  if (e.code() != MQe.Except_Q_NoMatchingMsg) {
   // if the exception is 'no matching
   //message then ignore it.  This
   // will result in a null return value.
   //Rethrow all other exceptions
   throw e;
  }
 }
 while (null == msg && secondsElapsed < 10) {
  Thread.sleep(1000);
  secondsElapsed++;
  try {
   msg = (MQeAdminMsg)myQM.getMessage(
          targetQMgr,
          MQe.Admin_Reply_Queue_Name,
          msgTest, null, 0L);
  } catch (MQeException e) {
   if (e.code() != MQe.Except_Q_NoMatchingMsg) {
    // if the exception is 'no matching message' then ignore it. This
    // will result in a null return value. Rethrow all other exceptions
    throw e;
   }
  }
 }
 return msg;
}
```

This method is a simple wrapper for the `MQeQueueManager` API call `waitForMessage()`, that sets up a filter to select the required admin reply, and casts any message obtained to an admin message.

```
/**
*Wait for message -waits for a message to arrive on the admin reply queue
*of the specified target queue manager.Will wait only for messages with the
*specified unique tag return message,or return null if timed out */

public static final MQeAdminMsg waitForRemoteAdminReply(
                                        MQeQueueManager localQueueManager,
                                        String remoteQueueManagerName,
                                        String match)throws Exception {
    //construct a filter to ensure we only get the matching reply
    MQeFields filter =new MQeFields();
    filter.putArrayOfByte(MQe.Msg_CorrelID,match.getBytes());
    //now wait for the reply message
    MQeMsgObject reply =localQueueManager.waitForMessage(
                                              remoteQueueManagerName,
                                              MQe.Admin_Reply_Queue_Name,
                                              filter,
                                              null,
                                              0L,
                                              10000);//wait for 10 seconds
    return (MQeAdminMsg)reply;
}
```

Set the required fields in the message

This section applies to the C codebase only. Since administration is performed asynchronously, you have to wait for the reply to the administration message in order to determine if the action was successful. You therefore need to request a reply, the default is to send no reply, and specify where to send the reply message. The destination for the reply message should be a convenient local queue. Remember that the administration code needs to send the reply message to the destination specified, and so may need connection definitions and listeners set up. It is easiest to get the administration reply message sent to the administration reply queue on the machine on which the administration is performed. The connectivity used to deliver the administration message to the target queue manager can then be used to retrieve the administration reply message from the target queue manager. This is the technique we use in the following examples.

Another useful task you can perform at this stage is to add an identifying field to the administration request message, so that you can easily identify the matching reply. You do this by adding a byte array field called MQe.Msg_CorrelID to the message. The administration code ensures that this field is copied into the reply message. If you wished you could then use this to correlate the administration action with the administration response.

Analyze the data in the administration reply message

Administration reply messages contain information about the success or failure of the attempt to perform the administration request. There are three levels of success:

- Total success - the action happened as requested. For enquire requests the messages contains the data requested.
- Total failure - the action failed. The message contains a reason why the action failed.
- Partial failure - some portion of a composite request failed. For example an attempt to update five fields might be successful for three, but unsuccessful for two. The fields that failed, and the reason for their failure is contained in the message.

Successful reply

If the administration action is successful then the return message contains a byte field called `MQeAdminMsg.Admin_RC` with a value of `MQeAdminMsg.RC_Success`.

Total failure

If the administration action is a complete failure then the return message contains a byte field called MQeAdminMsg.Admin_RC with a value of MQeAdminMsg.RC_Fail. It also contains a String field called `MQeAdminMsg#Admin_Reason` which contains a description of the failure.

Partial failure

If the administration action is a partial failure then the return message contains a byte field called `MQeAdminMsg.Admin_RC` with a value of `MQeAdminMsg.RC_Mixed`. The String field called `MQeAdminMsg.Admin_Reason` which only contains a general explanation 'errors occurred'. For more detail, access the field called `MQeAdminMsg.Admin_Errors`. The MQeFields object contains any errors related to subproblems that occur when a request fails with a return code of `RC_Fail or RC_Mixed`. For each attribute in error, there is a corresponding field in this MQeFields object. If the field that was processed was an array then the corresponding error field is of type ASCII array. If the field that was processed was not an array then the corresponding error field is of type ASCII.

For example if an update request was made to change 4 attributes of a resource and 2 of the updates were successful and 2 failed, this field would contain information detailing the reason for the 2 failures.

Each error is typically a `toString()` representation of the exception that caused the failure. If the exception is of type `com.ibm.mqe.MQeException` the string includes the `MQeException` code at the start of the string as ″Code=nnn″.

The basic administration reply message

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

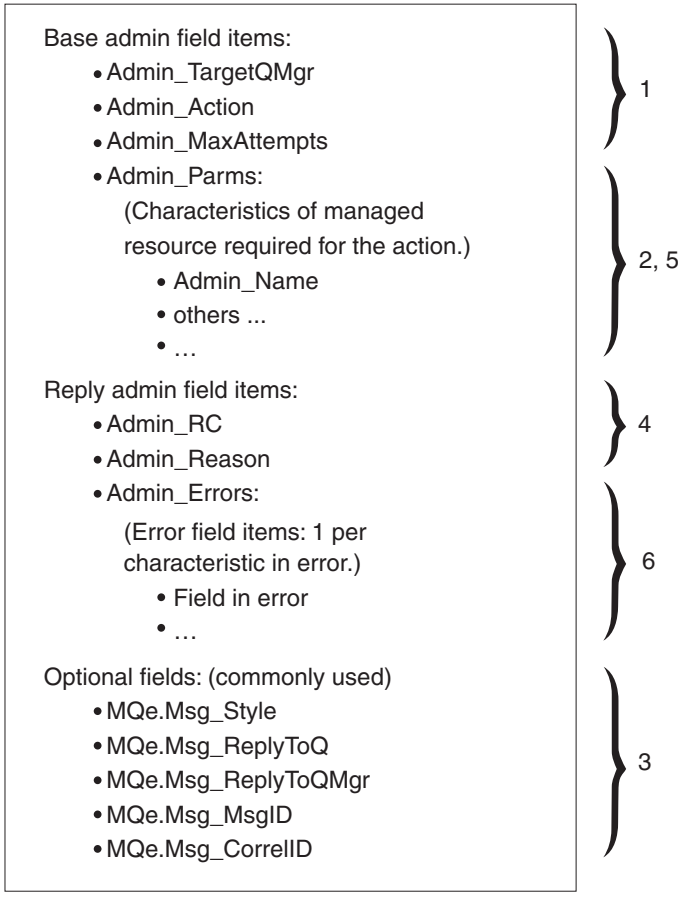

Figure 3. Administration reply message

A reply is made up of:

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

The first three items are describe in “The basic administration request message” on page 8. The reply specific fields are described in the following sections.

Outcome of request fields

Admin_RC field

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

MQeAdminMsg.RC_Success
The action completed successfully.

MQeAdminMsg.RC_Failed
The request failed completely.

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

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

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

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

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

Table 7. Getting administration reply fields

Administration field	Field type	Get method
Admin_RC	int	int getAction()
Admin_Reason	Unicode	String getReason()
Admin_Parms	MQeFields	MQeFields getOutputFields()
Admin_Errors	MQeFields	MQeFields getErrorFields()

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

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

Table 8. Enquiring on queue parameters

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

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

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

Table 9. Request and reply message to update a queue

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

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

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

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

Java

This method shows how you might analyze a reply message, and return a boolean to indicate whether the action was successful or not. We take the opportunity to print out any error messages to the console.

```
/**
*Reply true if the given admin reply
*message represents a successful
*admin action.Return false otherwise.
*A message indicating success
*or failure will be printed to the console.
*If the admin action was not successful then the reason will be printed
*to the console
*/
public static final boolean isSuccess(MQeAdminMsg reply)
             throws Exception {
    boolean success =false;
    final int returnCode =reply.getRC();
    switch (returnCode){
      case MQeAdminMsg.RC_Success:
        System.out.println("Admin succeeded");
        success =true;
        break;
      case MQeAdminMsg.RC_Fail:
  /* all on one line */
        System.out.println("Admin failed,reason:"+
              reply.getReason());
        break;
      case MQeAdminMsg.RC_Mixed:
        System.out.println("Admin partially succeeded:\n"
                           +reply.getErrorFields());
        break;
    }
    return success;
}
```

Administration of managed resources

As described in previous sections, WebSphere MQ Everyplace has a set of resources that can be administered with admin messages. These resources are known as managed resources.

Example administration console

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

This example demonstrates the following WebSphere MQ Everyplace administration features:

- Management of both local and remote queue managers
- Administration of all WebSphere MQ Everyplace managed resources

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

This is provided solely as a programming example, ***it is not expected to be used outside a development and test environment***. It should be noted that this example works with other examples such as trace, and the client queue manager, and it is also subclassified to provide an administration example for the WebSphere MQ bridge.

The main console window

To start the console use the command:

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

This displays the following window:

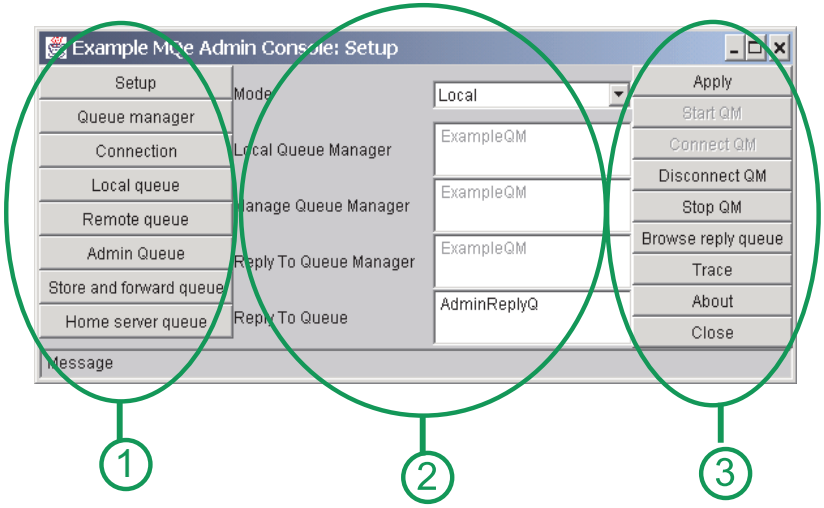

Figure 4. Administration console window

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

1. Type of resource to manage

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

2. Base administration parameters

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

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

Local queue manager
The name of the local queue manager that is initiating the administration actions. This is set automatically when a queue manager is started with the **Start QM** button.

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

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

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

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

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

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

Queue browser

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

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

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

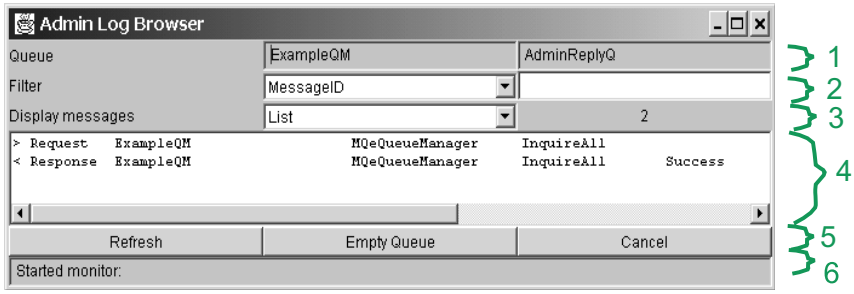

Figure 5. Reply-to queue window

This window has several sections:

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

2. Message filter

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

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

3. Message view type

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

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

Full: The contents of all messages on the queue.

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

The number of messages currently being viewed is also displayed.

4. Message display panel

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

5. Actions

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

Refresh

Clears the display and then displays the current contents of the queue. If the queue being browsed is a local queue, a monitor is automatically started. This monitor refreshes the display when new messages are added to the queue. If the queue being browsed is remote then it is not possible to automatically refresh the window

when new messages are added. In this case, the **Refresh** button can be used to get the latest contents of the queue.

Empty Queue
Deletes all messages from the queue.

Cancel Closes the queue browser window.

6. Message
Error and status messages are displayed here.

Action windows

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

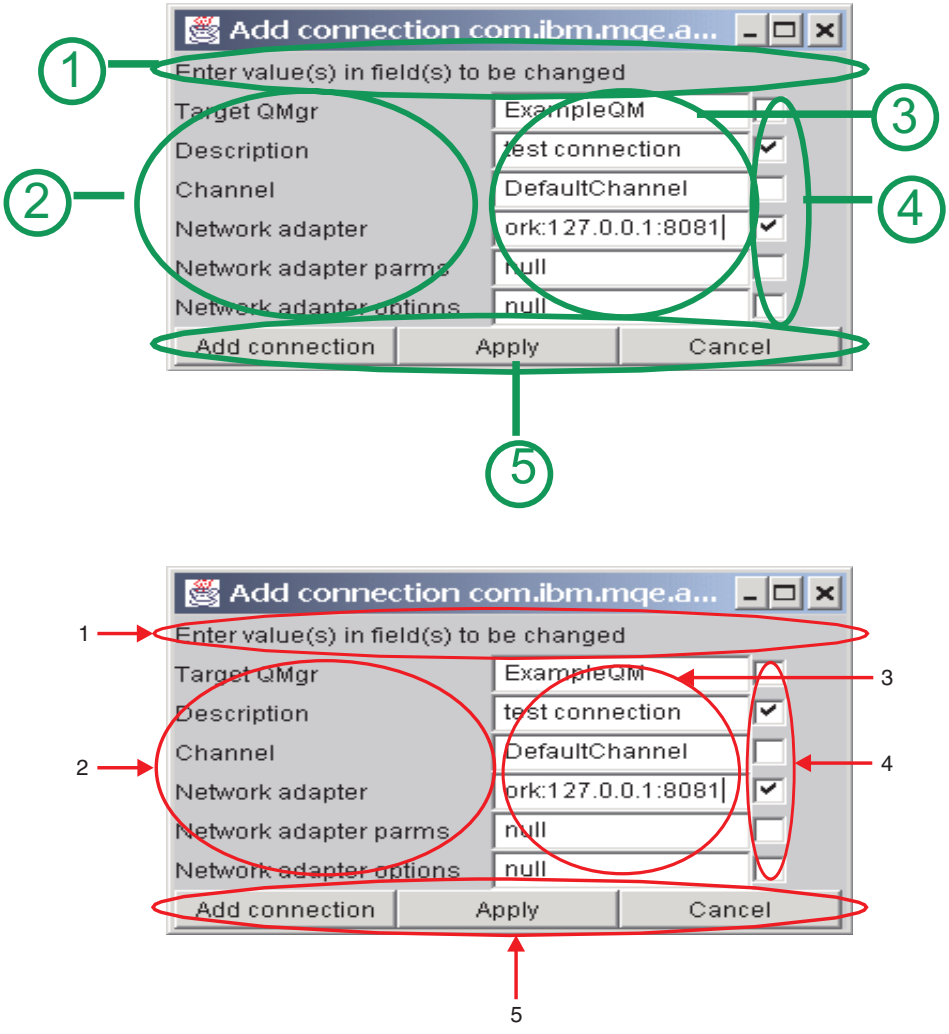

Figure 6. Action window

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

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

2. Names of parameter
Action parameter names.

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

4. Send field
The check box for each field is automatically selected when a value is

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

5. Action buttons

For each administration action there are three buttons:

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

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

Cancel Close the action window without sending the administration message.

Reply windows

You can view the outcome of an administration request with the administration log browser as described in “Queue browser” on page 23. To see the details of the result of the request, double click on the reply message in the list view.

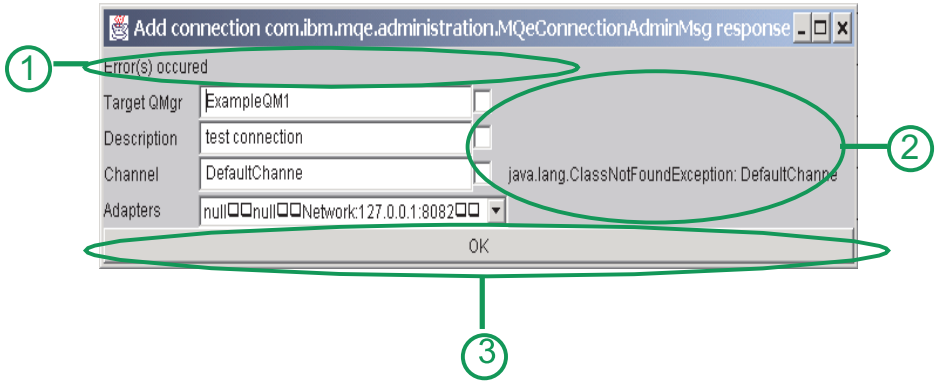

Figure 7. Reply window

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

1. Message

Displays the return code and result of the action.

2. Detailed errors

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

3. Action buttons

OK Close the action reply window.

Java

example (use enquire all on queue manager)

Decorating the queue manager in Java

This method is implemented in class examples.config.BasicAdministration. It performs the steps described above to address the administration message, request a reply, and add a unique marker to the message.

```
    /**
      * This method performs standard processing that
  * decorates an administration message
      * so that we can handle it in a standard way.
      * <p>This method:
      * <p> Sets the target queue manager
    * (the queue maanger upon which
      * the administration action takes place.
      * <p> Requests that a reply message is sent
  * to the administration reply queue on
      * the target queue manager.
      * <p> Incorporates a unique key in the message
  * that can be used to retrieve
      * the reply for this message.
  * The unique key is returned as a string, to be
     * used by the routine extracting the reply.
     */
    public static final String decorateAdminMsg(MQeAdminMsg msg,
                              String targetQMName) throws Exception {

        // set the target queue manager
        msg.setTargetQMgr(targetQMName);

        // indicate that we require a reply message
        msg.putInt(MQe.Msg_Style, MQe.Msg_Style_Request);

        // use default reply-to queue on the target queue manager.
        msg.putAscii(MQe.Msg_ReplyToQ, MQe.administration_Reply_Queue_Name);
        msg.putAscii(MQe.Msg_ReplyToQMgr, targetQMName);

        // create a unique tag that we can identify the reply with
        String match = "Msg" + System.currentTimeMillis();
        msg.putArrayOfByte(MQe.Msg_CorrelID, match.getBytes());

        return match;
    }
```

Putting the administration message in Java

Simply use the MQeQueueManager API call `putMessage()`, specifying the destination queue manager and the standard administration queue name. We can ignore the attribute, and confirmed parameters in our example, though they are available for more controlled access to the administration queue.

```
// put the message to the right administration queue
localQueueManager.putMessage(targetQueueManagerName,
                             MQe.Admin_Queue_Name,
                             msg, null, 0L);
```

Waiting for the administration reply in Java

This method is implemented in class `examples.config.BasicAdministration`. It is a simple wrapper for the MQeQueueManager API call `waitForMessage()`, that sets up a filter to select the required administration reply, and casts any message obtained to an administration message.

```
    /**
      * Wait for message - waits for a message to
  * arrive on the administration reply queue
      * of the specified target queue manager.
  * Will wait only for messages with the
      * specified unique tag
      * return message, or null if timed out
      */
    public static final MQeAdminMsg waitForRemoteAdminReply(
                    MQeQueueManager localQueueManager,
                    String remoteQueueManagerName,
                    String match) throws Exception {
        // construct a filter to ensure we only get the matching reply
        MQeFields filter = new MQeFields();
        filter.putArrayOfByte(MQe.Msg_CorrelID, match.getBytes());

        // now wait for the reply message
        MQeMsgObject reply = localQueueManager.waitForMessage(
            remoteQueueManagerName,
            MQe.Admin_Reply_Queue_Name,
            filter,
            null,
            0L,
            10000);          // wait for 10 seconds
        return (MQeAdminMsg)reply;
    }
```

Analyzing the reply message in Java

This method is implemented in class `examples.config.BasicAdministration`. It shows how you might analyze a reply message, and return whether the reply indicates that the action was successful or not. We take the opportunity to print out any error messages to the console.

```
   /**
     *  Reply true if the given administration
  * reply message represents a successful
      * administration action.  Return false otherwise.
  * A message indicating success
```

```
    * or failure will be printed to the console.
    * If the administration action was not successful
* then the reason will be printed
    * to the console
   */
  public static final boolean isSuccess(MQeAdminMsg reply)
              throws Exception {
      boolean success = false;
      final int returnCode = reply.getRC();
      switch (returnCode) {
          case MQeAdminMsg.RC_Success:
              System.out.println("Admin succeeded");
              success = true;
              break;
          case MQeAdminMsg.RC_Fail:
              System.out.println("Admin failed, reason:
           "+ reply.getReason());
              break;
          case MQeAdminMsg.RC_Mixed:
              System.out.println("Admin partially succeeded:\n"
                                      +reply.getErrorFields());
              break;
      }
      return success;
  }
```

Java

This method is implemented in class `examples.config.QueueManagerAdmin`. It shows how to use the primitives in the BasicAdministration class to update a queue manager description, and to report the success of the action.

```
   /**
     * Update the description field of the
* specified queue manager to the specified
    * string.  Use the supplied queueManager
* reference as the access to the
    * MQe network.
    *
    * @param queueManager (MQeQueueManager): access point to the MQe network
    * @param queueManagerName (String): name of queue manager to modify
    * @param (String): new description for queue manager
    */
  public static final boolean updateQueueManagerDescription(
                                       MQeQueueManager queueManager,
                                       String targetQueueManagerName,
                                       String description)
                                               throws Exception {
      // create administration message
      MQeQueueManagerAdminMsg msg = new MQeQueueManagerAdminMsg();

      // request an update
      msg.setAction(MQeAdminMsg.Action_Update);

       // set the new value of the parameter
 //into the input fields in the message
```

```
      // the field name is the attribute name,
// and the field value is the new
      // value of the attribute.  The type is specified
// by the administration message.
      // In this case, the field name is 'description',
// the value is the new
      // description, an the type is Unicode.
     msg.getInputFields().putAscii(
                         MQeQueueManagerAdminMsg.QMgr_Description,
                         description);

     // set up for reply etc
     String uniqueTag = BasicAdministration.decorateAdminMsg(
                               msg, targetQueueManagerName);

     // put the message to the right administration queue
     queueManager.putMessage(targetQueueManagerName,
                            MQe.Admin_Queue_Name,
                            msg, null, 0L);

     // wait for the reply message
     MQeAdminMsg reply = BasicAdministration.waitForRemoteAdminReply(
                           queueManager,
                           targetQueueManagerName,
                           uniqueTag);

     return BasicAdministration.isSuccess(reply);
 }
```

C

Refer to the example on how to analyze administration reply (use enquire all on queue manager).

Administration from the command line

WebSphere MQ Everyplace includes some tools that enable the administration of WebSphere MQ Everyplace objects from the command line, using simple scripts. The following tools are provided:

QueueManagerUpdater
Creates a device queue manager from an ini file, and sends an administration message to update the characteristics of a queue manager.

IniFileCreator
Creates an ini file with the necessary content for a client queue manager.

LocalQueueCreator
Opens a client queue manager, adds a local queue definition to it, and closes the queue manager.

HomeServerCreator
Open a server queue manager, adds a home-server queue, and closes the queue manager.

ConnectionCreator
Allow a connection to be added to an WebSphere MQ Everyplace queue manager without programming anything in Java.

RemoteQueueCreator
Opens a device queue manager for use, sends it an administration message to cause a remote queue definition to be created, then closes the queue manager.

MQBridgeCreator
Creates an WebSphere MQ bridge on an WebSphere MQ Everyplace queue manager.

MQQMgrProxyCreator
Creates a WebSphere MQ queue manager proxy for a bridge.

MQConnectionCreator
Creates a connection definition for a WebSphere MQ system on a proxy object.

MQListenerCreator
Creates a WebSphere MQ transmit queue listener to pull messages from WebSphere MQ.

MQBridgeQueueCreator
Creates an WebSphere MQ Everyplace queue that can reference messages on a WebSphere MQ queue.

StoreAndForwardQueueCreator
Creates a store-and-forward queue.

StoreAndForwardQueueQMgrAdder
Adds a queue manager name to the list of queue managers for which the store-and-forward queue accepts messages.

The following files are also provided:

Example script files
Two example .bat files, and a runmqsc script to demonstrate setting up a fictitious network configuration, involving a branch, a gateway, and a WebSphere MQ system.

Rolled-up Java example
An example of how a batch file can be rolled-up into a Java file for batch-language independence.

Example of use of command-line tools

The command-line tools can be used to create an initial queue manager configuration using a script, and without needing to know how to program in the Java programming language.

The following example demonstrate how to use these tools to configure the network topology shown in the following figure.

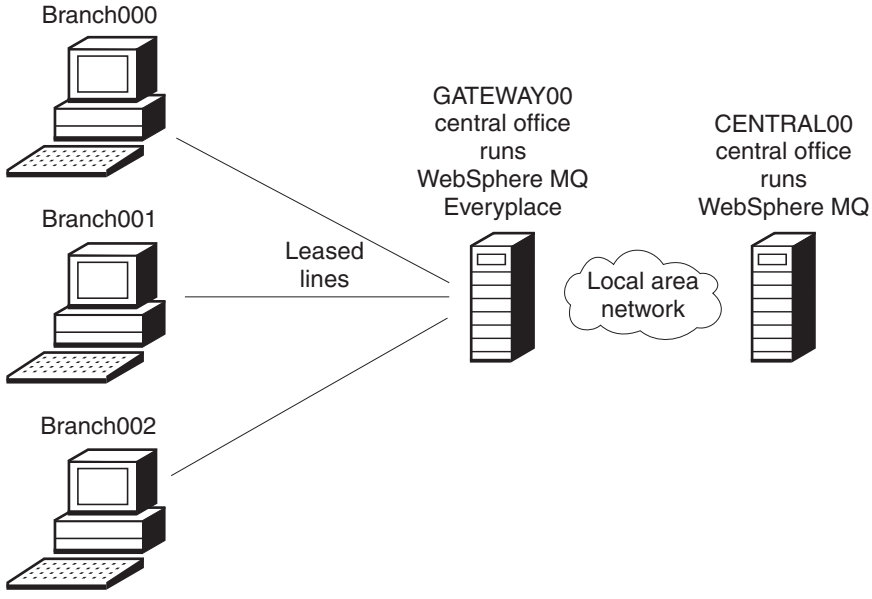

Figure 8. WebSphere MQ Everyplace administration scenario

In this scenario:

- The branch offices need to send sales information to the central site for processing by applications on the WebSphere MQ server
- Each branch has a single machine with DNS names `BRANCH000`, `BRANCH001`, and `BRANCH002` respectively. These machines all run WebSphere MQ Everyplace each having a single queue manager names `BRANCH000QM`, `BRANCH001QM`, and `BRANCH002QM` respectively.
- The central office machine `GATEWAY00` runs a single gateway queue manager `GATEWAY00QM`
- The central office machine `CENTRAL00` runs WebSphere MQ with a single queue manager `CENTRAL00QM`
- When a sale occurs, a message is sent to the WebSphere MQ queue manager `CENTRAL00QM`, into a queue called `BRANCH.SALES.QUEUE`.
- The messages are encoded in a byte array at the branch, and sent inside an MQeMQMsgObject.
- The WebSphere MQ system must be able to send messages back to each branch queue manager.
- The topology must also be able to cope with the addition of a Firewall later between the branches and the gateway.
- The WebSphere MQ-bound queue traffic should use the 56-bit DES cryptor.

Script files required

The following scripts are needed to configure this network topology.

Central.tst
Used with the runmqsc script to create relevant objects on CENTRAL00QM

CentralQMDetails.bat
Used to describe the CENTRAL00QM to other scripts

GatewayQMDetails.bat
Used to describe the GATEWAY00QM to other scripts

CreateGatewayQM.bat
Used to create the gateway queue manager

CreateBranchQM.bat
Used to create a branch queue manager

These .bat files can all be found in the installed product, in MQe\Java\Demo\Windows.

Note: Although the example scripts provided are in the Windows® .bat file format, they could be converted to work equally well in any scripting language available on your system.

WebSphere MQ Everyplace and WebSphere MQ objects defined by the scripts

The following objects are created by the scripts, to provide the branch-to-central routing:

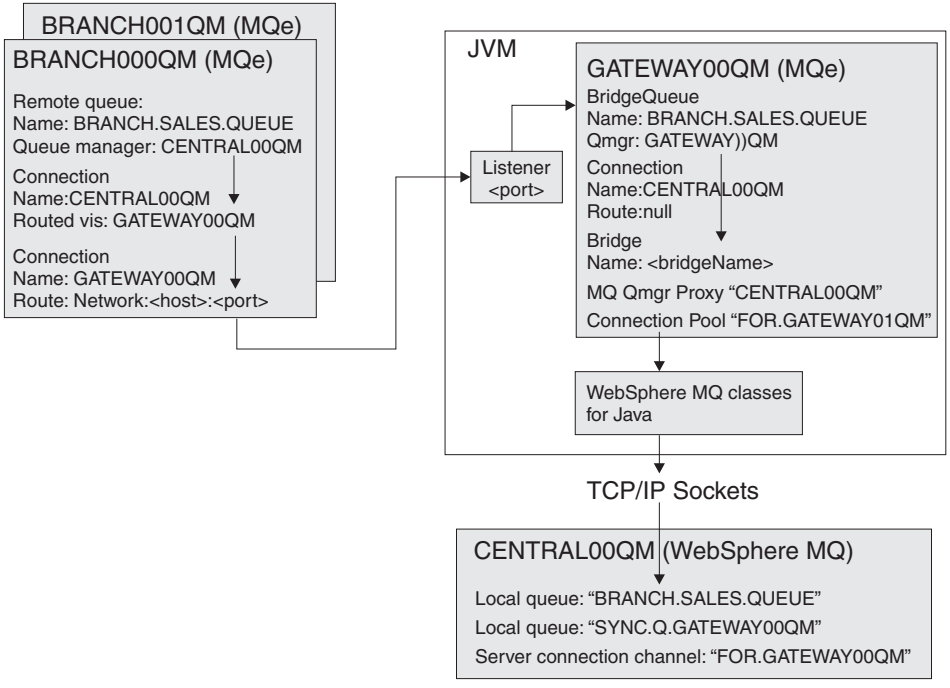

Figure 9. Branch to central routing

The following objects are created by the scripts to provide the central-to-branch routing:

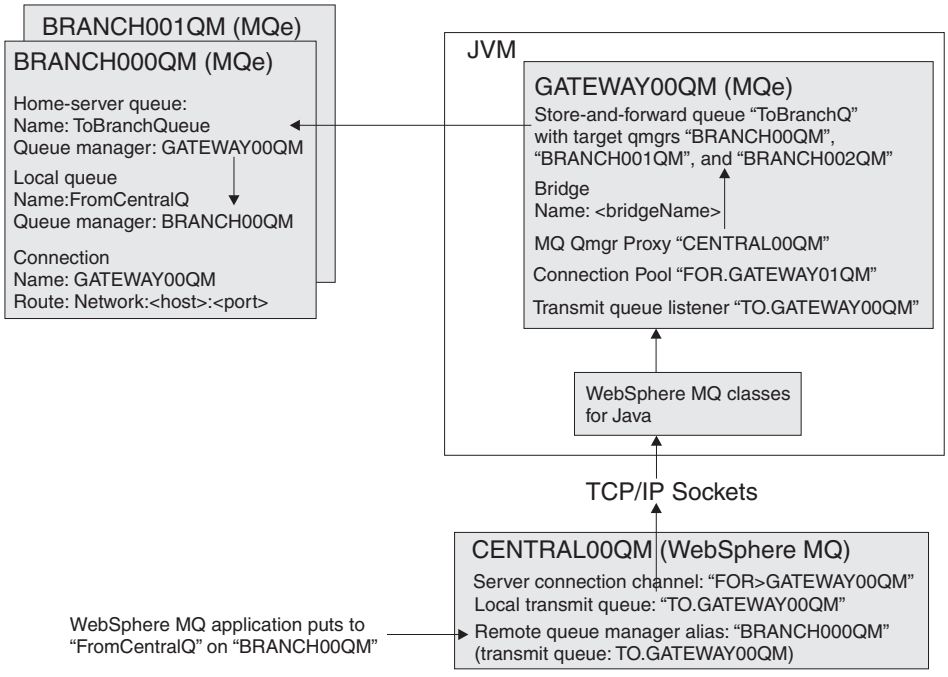

Figure 10. Central to branch routing

How to use the script files

Follow these procedures to create the required objects and operate the example scenario, using the supplied script files.

Edit the JavaEnv.bat .

Make sure you have edited the JavaEnv.bat file to set your required working environment.

Create a command-line session

Create a command-line session, and invoke the JavaEnv.bat to make the settings available in the current environment.

Gather hardware required

Locate all the hardware on which you will be installing the network topology.

Gather the machine names of those machines available to you, and note them down. If you have only one machine available, you can still use the scripts to deploy the example network topology, as you can specify the same hostname for each queue manager.

Create a WebSphere MQ queue manager

By default, the scripts assume this is called `CENTRAL00QM` listening on port 1414 for client channel connections.

Describe the WebSphere MQ queue manager

Edit and review the CentralQMDetails.bat file to make sure that its details match those of the WebSphere MQ queue manager you have just created. All values, except the name of the machine on which the WebSphere MQ queue manager sits, are defaulted in the script file.

Describe the gateway queue manager

Edit and review the GatewayQMDetails.bat file to make sure that details of the gateway queue manager are decided on, and available for the other .bat files to use.

The default name of the gateway queue manager created by the scripts is `GATEWAY00QM`. You will need to set the machine name, and port number it will listen on. This port must be available for use.

Tip: On Windows machines, use the command **netstat -a** to get a list of ports currently in use.

Review the central.tst file

Read the central.tst file, make sure it won't create any WebSphere MQ objects you are unhappy with on your WebSphere MQ queue manager.

Distribute all the scripts to all machines

Copy all of the scripts to all of the machines on which you will be running WebSphere MQ Everyplace queue managers.

This step spreads knowledge to all the machines in your network, of the host names, port numbers, and queue manager names that you have decided to use. If any of these files are changed, delete all WebSphere MQ Everyplace queue managers and restart from this point in the instructions.

Run the central.tst script on your new WebSphere MQ queue manger

The central.tst script is in a format used by the **runmqsc** sample program supplied with WebSphere MQ.

Pipe the central.tst file into **runmqsc** to configure your WebSphere MQ queue manger For example:

```
runmqsc CENTRAL00QM < Central.tst
```

Use the WebSphere MQ Explorer to view the resultant WebSphere MQ objects that are created.

Milestone: You have now set up your WebSphere MQ system.

Run the CreateGatewayQM script

The CreateGatewayQM script uses the details in the CentralQMDetails and GatewayQMDetails scripts to create a gateway queue manager.

The script needs no parameters.

Check for the test message

The script that creates the queue manager sends a test message to the WebSphere MQ system.

Use the WebSphere MQ Explorer tool to look at the target queue (`BRANCH.SALES.QUEUE` by default) to make sure a test message arrived. The body of the test message contains the string `ABCD`

Milestone: You have now set up your WebSphere MQ Everyplace gateway queue manager.

Keep the gateway queue manager running

During the running of the CreateGatewayQM script, an example server program is invoked to start the gateway queue manager, and keep it running. An AWT application runs, displaying a window on the screen.***Do not close this window***.

All the time this window is active, the WebSphere MQ Everyplace gateway queue manager it represents is also active. Closing the window closes the WebSphere MQ Everyplace gateway queue manager and breaks the path from the branch queue managers to the WebSphere MQ queue manager.

Create a branch queue manager

If your branch queue manager needs to run on a different machine, you may need to edit the JavaEnv.bat file to set up your local environment.

Create a command-line session, and call JavaEnv.bat as before to set up your environment.

Use the CreateBranchQM script to create a branch queue manager. The syntax of the command is :

```
CreateBranchQM.bat branchNumber portListeningOn
```

Where:

branchNumber

Is a 3-digit number, padded with leading zeros, indicating which branch the queue manager is being created for. For example, 000, 001, 002...

portListeningOn

Is a port on which the device branch queue manager listens on for administration requests. For example, 8082, 8083...

Note: The port must not already be in use
Hint: On Windows machines, use the **netstat -a** command to view the list of ports in use.

During the script, a test message is sent to your WebSphere MQ system. Use the WebSphere MQ Explorer to make sure the test message arrived successfully. The body of the test message contains the string `ABCD`.

At the end of the script, an example program is used to start the WebSphere MQ Everyplace queue manager. An AWT application

runs, displaying a window on the screen.As with the gateway queue manager, ***do not close this window*** until you wish to close the queue manager.

Explore the branch queue manager
The branch queue manager is set up with a channel manager and listener, on the port you specified when you created it, and the Primary Network connection is HttpTcpipAdapter. As a result, you can use the MQe_Explorer to view the queue managers. Refer to “How to use the MQe_Explorer to view the configurations”.

Milestone: You now have a branch queue manager set up.

Note: An WebSphere MQ Everyplace queue manager should be named uniquely. Never create two queue managers with the same name.

Start the MQe_Explorer.exe program. Stop one of the branch queue managers, say BRANCH002QM Open the BRANCH002QM.ini file, and navigate from there.

How to use the MQe_Explorer to view the configurations

To use the MQe_Explorer to view your configuration:

1. Start the MQe_Explorer.exe program.
2. Stop one of the branch queue managers, say `BRANCH002QM`
3. Open the BRANCH002QM.ini file, and navigate from there.

Chapter 3. Administration

This section describes some aspects of administration in WebSphere MQ Everyplace applications. Information is provided under the following sections:

Administration of managed resources

Security and administration

Example administration console

Administration from the command line

Administration of managed resources

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

Queue managers

The complete management life cycle for most managed resources can be controlled with administration messages. This means that the managed resource can be brought into existence, managed and then deleted with administration messages. This is not the case for queue managers. Before a queue manager can be managed it must be created and started.

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

Connections

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

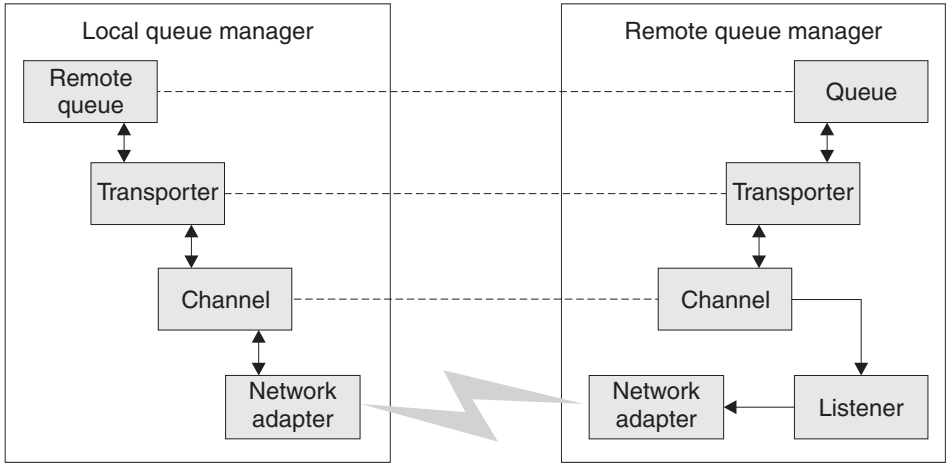

Figure 11. Queue manager connections

Communication happens at different levels:

Transporter:
Logical connection between two queues

Channel:
Logical connection between two systems

Adapter:
Protocol specific communication

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

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

  /*
   * Prime message with who to reply to and a unique identifier
   */
```

```
  MQeFields msgTest = primeAdminMsg( msg );

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

  /*
   * Set the admin action to create a new queue
   * The connection is setup to use a default channel. This is an alias
   * which must have be setup on the queue manager for the connection to
   * work.
   */
  msg.create( adapter,
              parms,
              options,
              channel,
              description );

  return msg;
}
```

Connecting queue managers in client to server mode

You can connect queue managers in client to server mode. In a client to server configuration, one queue manager acts as a client and the other runs in a server environment. A server allows multiple simultaneous incoming connections (channels). To accomplish this the server must have components that can handle multiple incoming requests. See the WebSphere MQ Everyplace Application Programming Guide for a description of how to run a queue manager in a server environment.

Figure 12 on page 42 shows the typical connection components in a client to server configuration.

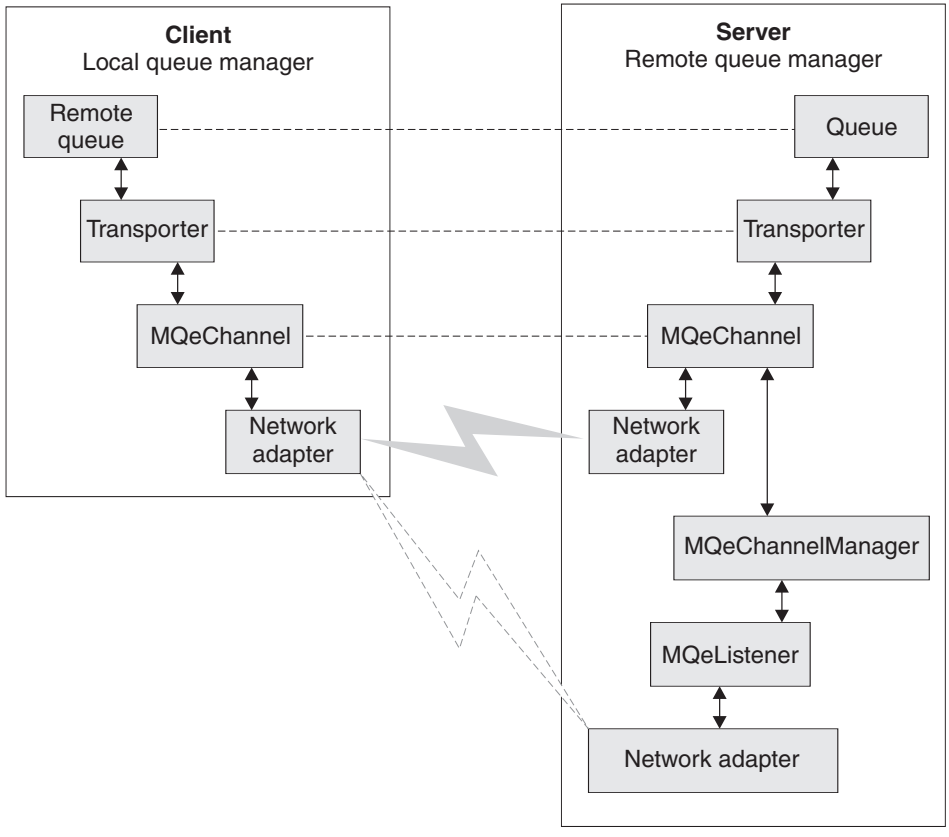

Figure 12. Client to server connections

Note: MQeChannelManager and MQeListener are deprecated in version 2.0.

You use MQeConnectionAdminMsg to configure the client portion of a connection. The channel type is `com.ibm.mqe.MQeChannel`. Normally an alias of `DefaultChannel` is configured for MQeChannel. The following code fragment shows how to configure a connection on a client to communicate with a server using the HTTP protocol.

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

```
  String options     = null;
  String channel     = "DefaultChannel";
  String description = "client connection to ServerQM";

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

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

}
```

Adapters

For details of the adapters supplied with WebSphere MQ Everyplace see the Chapter 2, Adapters, of the WebSphere MQ Everyplace System Programming Guide and Chapter 9 of the WebSphere MQ Everyplace Java Programming Reference.

Routing connections

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

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

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

Aliases

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

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

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

The following snippet of code shows how to add connection aliases to a message:

```
/**
 * Setup an admin msg to add aliases
  to a queue manager (connection)
 */
public MQeConnectionAdminMsg addAliases(  String queueManagerName
                                     String aliases[] )
             throws Exception
{
  /*
   * Create an empty connection admin message
   */
  MQeConnectionAdminMsg msg = new MQeConnectionAdminMsg();

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

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

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

  return msg;
}
```

Queues

The queue types provided by WebSphere MQ Everyplace are described briefly in Chapter 5, *Queues*, of the WebSphere MQ Everyplace Application Programming Guide. The simplest of these is a local queue that is implemented in class MQeQueue and is managed by class MQeQueueAdminMsg. All other types of queue inherit from MQeQueue. For each type of queue there is a corresponding administration message that inherits from MQeQueueAdminMsg. The following sections describe the administration of the various types of queues.

Local queue

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

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

The diagram below shows an example of a queue manager configured with a local queue. Queue manager `qm1` has a local queue named `invQ`. The queue manager name characteristic of the queue is qm1, which matches the queue manager name. Figure 13 shows a local queue:

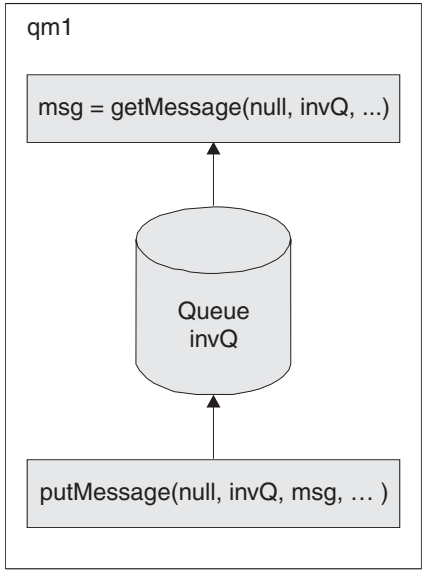

Figure 13. Local queue

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

adapter class:adapter parameters
or
adapter alias:adapter parameters

For example:

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

WebSphere MQ Everyplace Version 2.0.0.4 provides two adapters, one for writing messages to disk and one for storing them in memory. By creating an appropriate adapter, messages can be stored in any suitable place or medium (such as DB2® database or writable CDs).

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

If you do not provide message store information when creating a queue, it defaults to the message store that was specified when the queue manager was created.

Take the following into consideration when setting the *Queue_FileDesc* field:

- Ensure that the correct syntax is used for the system that the queue resides on. For instance, on a windows system use "\" as a file separator on UNIX® systems use "/" as a file separator. In some cases it may be possible to use either but this is dependent on the support provided by the JVM (Java Virtual Machine) that the queue manager runs in. As well as file separator differences, some systems use drive letters like Windows NT® whereas others like UNIX do not.
- On some systems it is possible to specify relative directories (".\") on others it is not. Even on those where relative directories can be specified, they should be used with great caution as the current directory can be changed during the lifetime of the JVM. Such a change causes problems when interacting with queues using relative directories.

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

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

  /*
   * Prime message with who to reply to and a unique identifier
   */
```

```
  MQeFields msgTest = primeAdminMsg( msg );

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

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

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

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

  /*
   *  Put the admin message to the admin queue (not assured delivery)
   */
  localQM.putMessage( qMgrName,
                      MQe.Admin_Queue_Name,
                      msg,
                      null,
                      0);
}
```

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

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

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

Aliases: Queue names can have aliases similar to those described for connections in "Aliases" on page 43. The code fragment in the connections section alias example shows how to setup aliases on a connection, setting up aliases on a queue is the same except that an MQeQueueAdminMsg is used instead of an MQeConnectionAdminMsg.

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

Action_Update

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

Action_Delete

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

If the request requires that the queue is not in use, or that it has zero messages, the administration request can be retried, either when the queue manager restarts or at regular time intervals. See "The basic administration request message" on page 8 for details on setting up an administration request retry.

Home-server queue

Home-server queues are implemented by the MQeHomeServerQueue class. They are managed with the MQeHomeServerQueueAdminMsg class which is a subclass of MQeRemoteQueueAdminMsg. The only addition in the subclass is the `Queue_QTimerInterval` characteristic. This field is of type int and is set to a millisecond timer interval. If you set this field to a value greater than zero, the home-server queue checks the home server every `n` milliseconds to see if there are any messages waiting for collection. Any messages that are waiting are delivered to the target queue. A value of `0` for this field means that the home-server is only polled when the MQeQueueManager.triggertransmission method is called

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

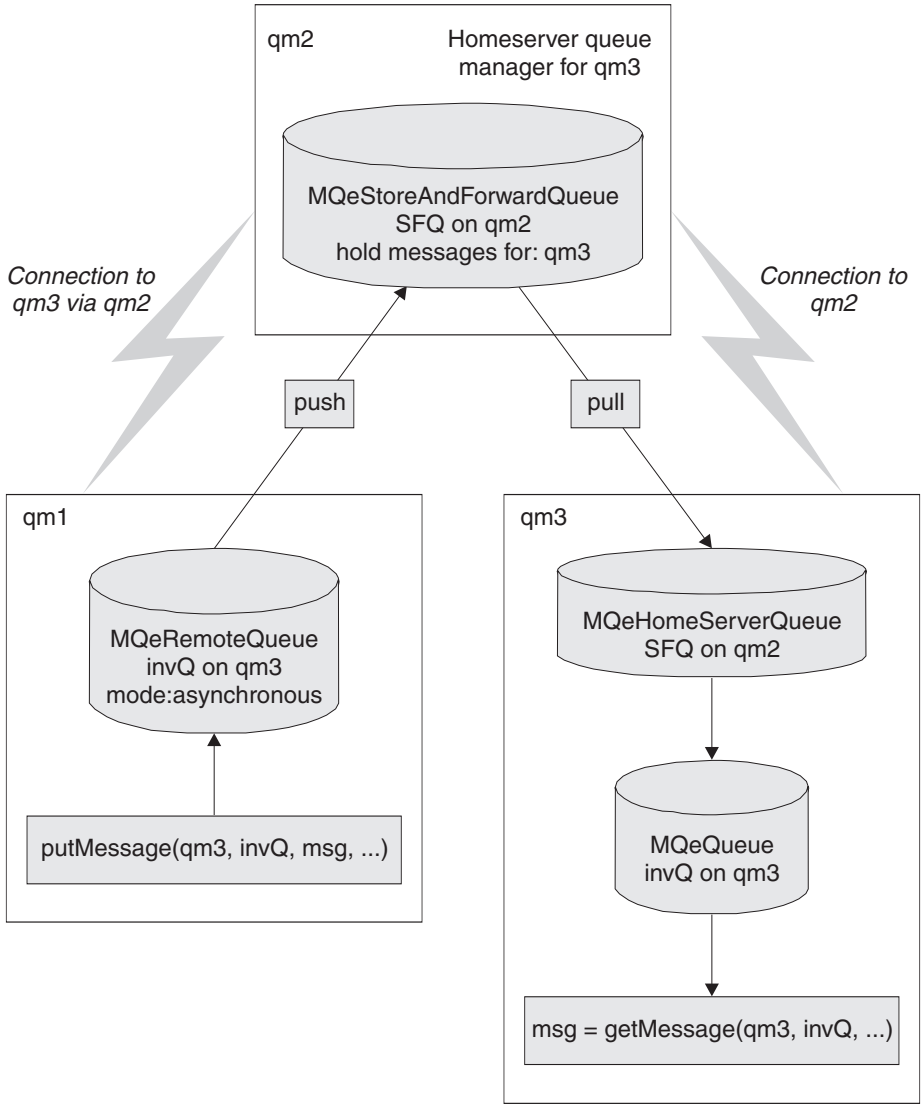

Figure 14. Home-server queue

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

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

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

Figure 14 on page 49 shows an example of a queue manager qm3 that has a home-server queue SFQ configured to collect messages from its home-server queue manager qm2.

The configuration consists of:

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

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

WebSphere MQ bridge queue

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

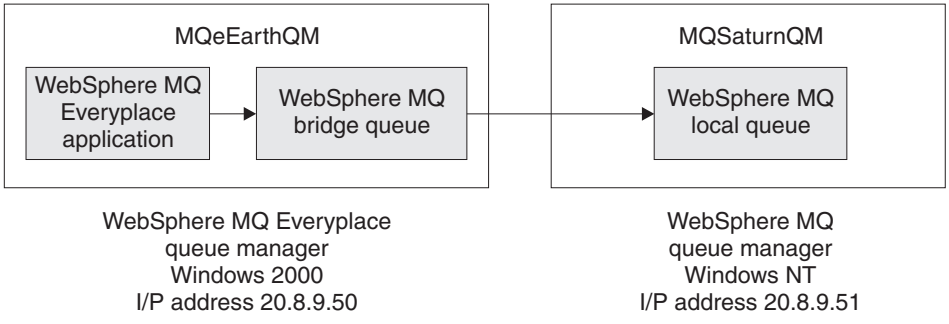

Figure 15. WebSphere MQ bridge queue

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

The definition of the bridge queue requires that bridge, WebSphere MQ queue manager proxy, and client connection names are specified to uniquely identify a client connection object in the bridge object hierarchy. Refer to Figure 47 on page 134 for more information. This information identifies how the WebSphere MQ bridge accesses the WebSphere MQ queue manager, to manipulate a WebSphere MQ queue.

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

For a complete list of the characteristics used by the WebSphere MQ bridge queue, refer to *MQeMQBridgeQueueAdminMsg* in the *com.ibm.mqe.bridge* section of *WebSphere MQ Everyplace Java Programming Reference*.

Table 10 details the list of operations supported by the WebSphere MQ bridge queue, once it has been configured:

Table 10. Message operations supported by WebSphere MQ—bridge queue

Type of operation	Supported by WebSphere MQ bridge queue
getMessage()	yes*
putMessage()	yes
browseMessage()	Yes*
browseAndLockMessage	no
Note: * These functions have restrictions on their use. Refer to Chapter 4, *Messaging*, of the WebSphere MQ Everyplace Application Programming Guide.	

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

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

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

If you do not wish to use synchronous operations against the WebSphere MQ bridge queue, you may set up an asynchronous remote queue definition that refers to the WebSphere MQ bridge queue. Refer to Chapter 7, *Message Delivery*, of the WebSphere MQ Everyplace Application Programming Guide for information on asynchronous message delivery. Alternatively, you can set up a store-and-forward queue, and home-server queue. These two alternative configurations provide the

application with an asynchronous queue to which it can put messages. With these configurations, when your `putMessage()` method returns, the message may not necessarily have passed to the WebSphere MQ queue manager.

An example of WebSphere MQ bridge queue usage is described in “Configuration example” on page 145.

Administration queue

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

If a message fails because the resource to be administered is in use, it is possible to request that the message be retried. “The basic administration request message” on page 8 provides details on setting up the maximum number attempts count. If the message fails due to the managed resource not being available and the maximum number of attempts has not been reached, the message is left on the queue for processing at a later date. If the maximum number of attempts has been reached, the request fails with an MQeException. By default the message is retried the next time the queue manager is started. Alternatively a timer can be set on the queue that processes messages on the queue at specified intervals. The timer interval is specified by setting the long field *Queue_QTimerInterval* field in the administration message. The interval value is specified in milliseconds.

Security and administration

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

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

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

Additional information on implementing queue rules can be found in the WebSphere MQ Everyplace System Programming Guide.

Chapter 4. Administration using the administrator API

To create and administer Queue Managers and their associated objects (queues etc.), the Java API uses the MQeQueueManagerConfigure class and admin messages. In the C API, admin activities are performed using an Administrator API. The native codebase responds to admin messages correctly but no provision is provided for creating them and hence the Administrator API is the recommended method for local administration.

This chapter explains the basics of the Administrator API, under the following headings:

- Creating an administrator handle
- Using the administrator handle
- Freeing the administrator handle

The Administrator API is used in later chapters of this book to perform simple tasks. For complete documentation on the Administrator API and all the available options, refer to the WebSphere MQ Everyplace C Programming Reference.

Creating an administrator handle

Before any administration can take place, an administrator handle must be created using the `mqeAdministrator_new` API call. The prototype for the call is:

```
MQERETURN mqeAdministrator_new(MQeExceptBlock*   pExceptBlock,
                               MQeAdministratorHndl* phAdmin,
                               MQeQueueManagerHndl   hQueueMgr)
```

The first parameter is a pointer to a valid exception block. The second parameter is a pointer to an administrator handle, which is filled in with a valid handle upon successful return from the function. The third parameter is an optional queue manager handle. If the queue manager to be administered already exists, it must be created using the `mqeQueueManager_new` function, and the queue manager handle returned must be passed to the `mqeAdministrator_new` call.

To create a queue manager, `NULL` must be passed as the third parameter to the `mqeAdministrator_new` call. If `NULL` is used, pass the `mqeAdministrator_free` or `mqeAdministrator_QueueManager_create` call. Once the `mqeAdministrator_QueueManager_create` call has been executed, the administrator handle can be used as normal.

Using the administrator handle

Once an Administrator Handle has been created any of the `mqeAdministrator` calls can then be used, the calls are all of the form:

```
MQERETURN mqeAdministrator_Object_action(
                               MQeAdministratorHndl hAdministrator,
                               MQeExceptBlock*   pExceptBlock,
                               ...)
```

Where object is the type of object to be administered, for example, a queue manager, local queue, synchronous remote queue, and so on, and action is the operation to be performed, for example, create, delete, inquire, update, and so on.

Note: Some Actions are only available for some object types.
Example calls:

If `NULL` is used to create an `MQeAdministratirHndl`, the next administration API call can only be one of MQeAdministrator_free or MQeAdministrator_create_QueueManager. Once the queue manager has been created, all the administration APIs are available for use.

```
 mqeAdministrator_LocalQueue_create
/* create a local queue */

 mqeAdministrator_AdminQueue_inquire
/* inquire on a local queue */
```

Many of the APIs, particularly the inquire and update calls, have arguments which are structures containing multiple elements some of which may or may not be filled in. In order to accommodate this functionality, such structures contain an element called ″opFlags″, a set of bits to indicate which elements of the structure are set. Also supplied are macros which initialize these opFlag structures to appropriate values and macros for each bit which can be set.

For instance, if you wanted to inquire on a local queue but you were only interested in the description and the Maximum Message Size fields, then you would do the following:

```
MQeLocalQParms lqParms = LOCAL_Q_INIT_VAL;
lqParms.opFlags |= QUEUE_DESC_OP;
lqParms.opFlags |= QUEUE_MAX_MSG_SIZE_OP;
/* Note that the | function is being used */

/* call inquire function */
```

Similarly, if you wanted to test which elements are filled in when such a structure is returned from a function, you would do the following:

```
if(lqParms.opFlags & QUEUE_DESC_OP)
{ /* description is set*/
}
if(lqParms.opFlags & QUEUE_MAX_MSG_SIZE_OP)
{ /* max msg size is set*/
}
```

Freeing the administrator handle

When the application has finished with the administrator handle it should be destroyed using the `mqeAdministrator_free` call. This allows the system to free up any resources which are in use by the administrator. Once an administrator handle has been freed, it must not be used in any of the `mqeAdministrator_*` API calls - if the handle is used, the behavior is indeterminate, but is likely cause an access violation. If further

administration actions are to be performed, the handle can be recreated with the mqeAdministrator_new call.

```
rc = mqeAdministrator_new(&exceptBlock,
                          &hAdministrator,
                          NULL);
if(MQERETURN_OK == rc)
{  /* mqeAdministrator_QueueManager_create  */
   /* further mqeAdministrator calls */
   /* ... */
   rc = mqeAdministrator_free(hAdministrator,
                              &exceptBlock);
} hAdministrator = NULL;
```

Figure 16. Creating an Administrator Handle for a new Queue Manager

We recommend that once a handle has been freed it is accidentally set to NULL. If this handle is then accidentally reused, the API returns an error.

```
/* mqeQueueManager_new(...,&hQueueManager,...) */
/* ... */
rc = mqeAdministrator_new(&exceptBlock,
                          &hAdministrator,
                          hQueueManager);
if(MQERETURN_OK == rc)
{
   /* further mqeAdministrator calls */
   /* ... */
   rc = mqeAdministrator_free(hAdministrator,
                              &exceptBlock);
}
```

Figure 17. Creating an Administrator Handle for an existing Queue Manager

Table 11. Common reason and return codes

Return codes	Reason codes	Notes
MQERETURN_ADMINISTRATION_ERROR	MQEREASON_INVALID_QMGR_NAME	Name has invalid character or is NULL
	MQEREASON_INVALID_QUEUE_NAME	Name has invalid character or is NULL
MQERETURN_INVALID_ARGUMENT	MQEREASON_API_NULL_POINTER	Pointer is NULL
	MQEREASON_WRONG_TYPE	Wrong type handle has been passed, for example, QueueManager hndl instead of MQeFields
MQERETURN_QUEUE_ERROR	MQEREASON_QMGR_QUEUE_EXISTS	Queue already Exists

Table 11. Common reason and return codes (continued)

Return codes	Reason codes	Notes
	MQEREASON_QMGR_QUEUE_NOT_EMPTY	Queue is not empty
MQERETURN_QUEUE_MANAGER_ERROR	MQEREASON_UNKOWN_QUEUE	Queue does not exist
	MQEREASON_UNKOWN_QUEUE_MANAGER	Queue manager does not exist
MQERETURN_NOTHING_TO_DO	MQEREASON_DUPLICATE	Name already in use
	MQEREASON_NO_SUCH_QUEUE_ALIAS	The queue alias specified does not exist

Chapter 5. Creating and starting queue managers

WebSphere MQ Everyplace queue managers, irrespective of their role within the WebSphere MQ Everyplace network, require some information to be held in permanent storage. This is the responsibility of WebSphere MQ Everyplace. Iif there is additional information that must persist between invocations of an application, this is the responsibility of the application.

Information held within the registry contains Queue Manager configuration details, for example:

- Information on where messages, queues, remote queue definitions, channel timeout, aliases, adapters, and the message store are held and how to access them
- Connection definitions
- Security information
- Various bridge related objects

The following persistent information, useful to an application, is referred to in this manual as environmental data:

- Registry information, class, path, storage adapter class, and registry type. This information is used to locate an existing registry, allowing WebSphere MQ Everyplace to start an existing queue manager, or to create a new queue manager registry.
- Class manager information, for example class and name.
- Queue manager type.

Creating and starting simple queue managers

The simplest WebSphere MQ Everyplace queue manager is a queue manager that uses a registry based upon the internal default values. The queue manager could be created without any queues, but its functionality would be severely limited. The example we create contains four standard queues:

- Admin queue - so that administration can be performed
- Admin reply queue - a standard place to store replies from administration actions
- System default queue - a useful general purpose local queue
- Dead letter queue - a place for undeliverable messages

The simplest queue manager has no security and has a registry stored in the local file system. The steps to achieve are:

- Create a registry on disk
- Create and start a queue manager using the registry
- Stop the queue manager

These actions are described for both the Java codebase and the C codebase, with example code for each. The example Java code is shipped as

examples.config.CreateQueueManager. For C example code, refer to the HelloWorld compilation section and the transport-c file in the Broker example.

Creating a simple queue manager in Java

Registries are created in Java by using the class com.ibm.mqe.MQeQueueManagerConfigure. An instance of this class is created, and activated by passing it some initialization parameters. The parameters are supplied in the form of an MQeFields object. Within this MQeFields are contained two sub fields, one holding information about the registry, and one holding information about the queue manager being created. As we are creating a very simple queue manager, we only need to pass two parameters, the queue manager name, in the queue manager parameters, and the registry location, in the registry parameters. We can then use the MQeQueue ManagerConfigure to create the standard queues.

First, create three fields objects, one for the QueueManager parameters, one for the Registry parameters. The third fields object, parms, is used to contain both the QueueManager and Registry fields objects.

```
MQeFields parms = new MQeFields();
MQeFields queueManagerParameters = new MQeFields();
MQeFields registryParameters = new MQeFields();
```

The QueueManager name needs to be set. Use the MQeQueueManager.Name as the Field Label constant.

```
queueManagerParameters.putAscii(MQeQueueManager.Name, queueManagerName);
```

The location of the persistent registry needs to be specified. Do this in the Registry Paramters field object. Use the MQeRegistry.DirName as the Field Label constant.

```
registryParameters.putAscii(MQeRegistry.DirName, registryLocation);
```

The QueueManager and registry parameters can now be set embedded the main fields object.

```
parms.putFields(MQeQueueManager.QueueManager, queueManagerParameters);
parms.putFields(MQeQueueManager.Registry, registryParameters);
```

An instance of MQeQueueManagerConfigure can be created now. This needs the parameters fields object, plus a String indentifing the details of the queue store to use.

```
MQeQueueManagerConfigure qmConfig =
new MQeQueueManagerConfigure(parms, queueStore);
```

The four common types of queues can now be created via four convenience methods as follows:

```
qmConfig.defineQueueManager();
qmConfig.defineDefaultSystemQueue();
qmConfig.defineDefaultDeadLetterQueue();
qmConfig.defineDefaultAdminReplyQueue();
qmConfig.defineDefaultAdminQueue();
```

Finally the MQeQueueManagerConfigure object can be closed.

```
qmConfig.close();
```

Starting a simple queue manager in Java

Starting the simplest queue manager is facile, as we only need to provide the queue manager name and registry location to the queue manager constructor. This starts and activates the queue manager, and when the constructor returns the queue manager is running.

Figure 18. Start queue manager Java example

```
MQeQueueManager qm = newMQeQueueManager(queueManagerName, registryName);
```

There are other ways to start a queue manager that allow us to pass more parameters, in order to take advantage of some advanced features, which are explained in later chapters.

Stopping a queue manager in Java

There are 2 ways to close down a QueueManager.

- closeQuiese
- closeImmediate

closeQuiese

This closes Queue Manager, specifying a delay to allow existing internal processes to finish normally. Note that this delay is only implemented as a series of 100ms pause and retry cycles. Calling this method will prevent any new activity, such as transmitting a message, from being started, but will allow activities already in progress to complete. The delay is a suggestion only, and various JVM dependant thread scheduling factors could result in the delay being greater. If the activities currently in progress finish sooner, then the method will return before the expiry of the quiesce duration.

If at the expiry of this period the queue has not closed, it is forced to close.

This method closes down the queue manager. One of the close methods should be called by WebSphere MQ Everyplace applications when they have finished using the queue manager.

After this method has been called, no more event notifications will be dispatched to message listeners. It is conceivable that messages may complete their arrival after this method has been called (and before it finishes). Such messages will not be notified. Application programmers should be aware of this, and not assume that every message arrival will generate a message event.

```
MQeQueueManager qmgr = new MQeQueueManager();
MQeMsgObject msgObj = null;
try {
 qmgr.putMessage(null, "MyQueue", msgObj, null, 0);
} catch (MQeException e) {// Handle the exception here
}
qmgr.closeQuiesce(3000); // close QMgr
```

closeImmediate

This closes Queue Manager immediately. One of the close methods should be called by WebSphere MQ Everyplace applications when they have finished using the queue manager.

After this method has been called, no more event notifications are dispatched to message listeners. Messages might complete their arrival after this method has been called, and before it finishes. Such messages are not notified. Application programmers should be aware of this, and not assume that every message arrival will generate a message event.

```
MQeQueueManager qmgr = new MQeQueueManager();
MQeMsgObject msgObj = null;
try {
 qmgr.putMessage(null, "MyQueue", msgObj, null, 0);
} catch (MQeException e) {// Handle the exception here
}
qmgr.closeImmediate();  // close QMgr
```

Creating a simple queue manager in C

Stage 1: Create the admin components

All local administration actions can be accomplished using the MQeAdministrator. This allows you to create new QueueManagers and new Queues, and perform many other actions. For all calls, a pointer to the exception block is required, along with a pointer for the QueueManager handle.

Stage 2: Create a QueueManager

To create a QueueManager, two parameters structures are required. One contains the parameters for the QueueManager, the other for the registry. In this simple case the default values are suitable, with the addition of the location of the registry and queue store.

The call to the administrator will create the QueueManager. Note that the QueueManager name is passed into the call. A QueueManager Hndl is returned.

```
if ( MQERETURN_OK == rc ) {

   MQeQueueManagerParms qmParams  = QMGR_INIT_VAL;
   MQeRegistryParms     regParams = REGISTRY_INIT_VAL;

   qmParams.hQueueStore           = hQueueStore;
   qmParams.opFlags               = QMGR_Q_STORE_OP;
   regParams.hBaseLocationName    = hRegistryDir;

   display("Creating the Queue Manager\n");
   rc = mqeAdministrator_QueueManager_create(hAdministrator,
                                             &exceptBlk,
                                             &hQueueManager,
                                             hLocalQMName,
                                             &qmParams,
                                             &regParams);

}
```

Figure 19. Create queue manager C example

Starting a simple queue manager in C

This process involves two steps:

1. Create the queue manager item.
2. Start the queue manager.

Creating the queue manager requires two sets of parameters, one set for the queue manager and one for the registry. Both sets of parameters are initialized. The *queue store* and the registry require directories.

Note: All calls require a pointer to ExceptBlock and a pointer to the queue manager handle.

```
 if (MQERETURN_OK == rc) {

MQeQueueManagerParms qmParams  = QMGR_INIT_VAL;
MQeRegistryParms     regParams = REGISTRY_INIT_VAL;
qmParams.hQueueStore           = hQueueStore;
qmParams.opFlags               = QMGR_Q_STORE_OP;

/* ... create the registry parameters -
  minimum that are required */
regParams.hBaseLocationName     =  hRegistryDir;
display("Loading Queue Manager from registry \n");
rc = mqeQueueManager_new( &exceptBlock,
                        &hQueueManager,
                           hLocalQMName,
                          &qmParams,
                          &regParams);
}
```

You can now start the queue manager and carry out messaging operations:

```
 /* Start the queue manager  */

if ( MQERETURN_OK == rc ) {
  display("Starting the Queue Manager\n");
  rc = mqeQueueManager_start(hQueueManager,
          &exceptBlock);
}
```

Stopping a queue manager in C

Following the removal of the message from the queue, you can stop and free the queue manager. You can also free the strings that were created. Finally, terminate the session:

```
(void)mqeQueueManager_stop(hQueueManager,&exceptBlock);
(void)mqeQueueManager_free(hQueueManager,&exceptBlock);

/* Lets do some clean up */
(void)mqeString_free(hFieldLabel,&exceptBlock);
```

```
(void)mqeString_free(hLocalQMName,&exceptBlock);
(void)mqeString_free(hLocalQueueName,&exceptBlock);
(void)mqeString_free(hQueueStore,&exceptBlock);
(void)mqeString_free(hRegistryDir,&exceptBlock);


(void)mqeSession_terminate(&exceptBlock);
```

Configuring a queue manager using memory only

This section applies only to queue managers using the Java programming interface.

It is sometimes required that applications have a queue manager which exists in memory only. WebSphere MQ Everyplace Version 2.0 provides the ability to configure and use a queue manager using memory resources only, without the need to persist any information at all to disk.

A WebSphere MQ Everyplace queue manager normally uses two mechanisms to store data:

- Configuration information is stored via a registry to an adapter.
- Messages are stored via a message store, which in turn uses an adapter to store data.

The default is the MQeDiskFieldsAdapter, which persists information to disk.

Using the MQeMemoryFieldsAdapter instead of the MQeDiskFieldsAdapter for both of these tasks allows the queue manager to be defined, used to transmit and store messages, and deleted all without accessing a disk.

In-memory WebSphere MQ Everyplace queue managers have the following characteristics:

- Functionally they can do everything other WebSphere MQ Everyplace queue managers can do.
- Nothing is stored to disk.
- Messages and configuration stored to registries or queues are nonpersistent. They are lost if all instances of the MQeMemoryFieldsAdapter are garbage collected, or in the event of the JVM being shut down.
- The same steps are required to configure the in-memory queue manager, except they are required every time the JVM is started.
- Transient queue managers which are created, used, and destroyed can be easier to implement, with no clean-up problems if the JVM terminates abnormally.

Solutions that find this particular configuration of an WebSphere MQ Everyplace queue manager useful have the following properties:

- Disk space is not available or nonexistent, for example in Java applets.
- Message traffic is synchronous only to remote queue managers.

- The application requires no local message store which cannot be recovered from elsewhere if the JVM is terminated.
- The highest performance is required. Memory operations are much faster than disk operations, so configuring a queue manager using purely memory resources normally increases performance of queue manager configurations which, otherwise store information to disk. Using too much memory can result in thrashing, and synchronous remote queues ususally run at the same speed on a memory-hosted or disk-hosted queue manager.
- Creation and sending of messages for which no replies are required, though in-memory queue managers can obtain replies, you would normally leave replies on persistent queue managers and browse or get them using a synchronous remote queue.

An example of the configuration technique can be seen in the examples.queuemanager.MQeMemoryQM class. Note that the MQeMemoryFieldsAdapter is instantiated explicitly at the start, and a reference is held until the point where the queue manager, and messages it contains are no longer required.

Note also that it is still important that In-memory queue managers have names which are unique within the messaging network.

Chapter 6. Administering queue managers

This chapter explains how to administer queue managers. Further chapters explain how to administer Local Queues, Remote Queue Definitions, Store and Forward Queues, and Home Server Queues.

General notes

The queue manager is the central component of WebSphere MQ Everyplace, it provides the main programming interface for application programs as well as owning queues, communication and WebSphere MQ bridge subsystems. Java and C differ significantly in the area of creating and deleting queue managers. In Java, general qmgr administration is performed using admin messages, but creating and deletion is performed using the MQeQueueManagerConfigure class. In C, all administration is performed using the administrator API.

Java

Queue managers are created and deleted using the `MQeQueueManagerConfigure` class. General queue manager administration is performed using the `MQeQueueManagerAdminMsg` class which inherits from `MQeAdminMsg`.

The following actions are applicable to queue managers:

- `MQeAdminMsg.Action_Inquire`
- `MQeAdminMsg.Action_InquireAll`
- `MQeAdminMsg.Action_Update`

The `MQeAdminMsg.Admin_Name` field in the administration message is used to identify the queue manager. The method setName(String) can be used set this field in the administration message.

Note: For all administration messages, information relating to the destination queue manager, reply queue and so on, must be set. This is referred to in the examples below as priming the administration message.

The examples show how to create the admin to achieve the required result. These message need to then be sent, and the admin reply messages checked as required.

C

All administration is done via the administration API. These APIs are of the form:

```
MQERETURN MQEPUBLISHED mqeAdministrator_QueueManager_action();
```

Where action can be one of the following:

create Create a Queue Manager

delete Delete a Queue Manager

update Updates the properties of a queue manager

inquire Inquires the properties of a queue manager

addAlias
Adds a Queue Manager Alias

removeAlias
Removes a Queue Manager Alias

listAliasNames
Lists all the aliases present for this qmgr.

isAlias Determines if a qmgr name is an alias or a real qmgr.

For the create update and inquire calls a structure is passed in for various parameters.

Queue Manager attributes

Queue Managers have a number of attributes, which are listed below. Information about these attributes is passed either via API parameters or configuration structures/MQeFields objects.

The first list shows all the possible queue manager attributes and indicates which are available in the code bases.

Table 12. Queue Manager attributes

Attribute	Description	Java	Native	Read/Write
Bridge Capable	Determines if the qmgr has MQBridge functionality	Yes	Yes (but always false)	Read
Channel Attribute Rule	The attribute rule to be used by this queue manager's channels	Yes	No	Read/Write
Channel Timeout	The timeout to be used by this queue manager's outgoing channels	Yes	Yes	Read/Write
Communications Listeners	The list of listeners defined on this queue manager	Yes	No	Read
Connections	The list of connections known by this queue manager	Yes	Yes	Read

Table 12. Queue Manager attributes (continued)

Attribute	Description	Java	Native	Read/Write
Description	A free-format textual description of this qmgr.	Yes	Yes	Read/Write
Maximum Transmission Threads	The maximum number of background transmission threads supported by this qmgr.	Yes	No	Read/Write
Queues	The list of queues owned by this queue manager	Yes	Yes	Read
Queue Store	The location where this queue manager will store its queues	Yes	Yes	Read/Write
Qmgr Rules	The rules class which will be used by this queue manager	Yes	Yes	Read/Write

Java

The parameters in Java are passed in using MQeFields objects. The values are passed using field elements of specific types.

The field names are as follows. All the symbolic names are pubic static final static strings in the MQeQueueManagerAdminMsg class.

Table 13. Java Parameters passed in using MQeFields

Element type	Field name constants	
	Symbolic	Value
boolean	QMgr_BridgeCapable	bridge_capable
ascii	QMgr_ChnlAttrRules	chnlattrrules
long	QMgr_ChnlTimeout	chnltimeout
fields array	QMgr_CommsListeners	commsls
fields array	QMgr_Connections	conns
unicode	QMgr_Description	desc
int	QMgr_Maximum TransmissionThreads	maximumTransmissionThreads

Table 13. Java Parameters passed in using MQeFields (continued)

Element type	Field name constants	
	Symbolic	Value
fields array Each element contains a fields object containing {QMgr_QueueName, QMgr_QueueQMgrName, QMgr_QueueType}	QMgr_Queues	queues
ascii	QMgr_QueueStore	queueStore
ascii	QMgr_Rules	rules

C parameters

All the C parameters are passed in using a parameter structure. This structure needs to be initialized before it can be used - set it to QMGR_INIT_VAL.

Table 14. Parameter structures for C

Element Type	Element Name	Notes
MQEINT32	opFlags	Flags to indicate what parts of this structure have been set/requested
MQeStringHndl	hDescription	
MQeStringHndl	hQueueManagerRules	
MQEINT64	channelTimeOut	
MQeStringHndl	hQueueStore	
MQeVectorHndl	hQueues	
MQeVectorHndl	hConnections	
MQEBOOL	bridgeCapable	Valid values {MQE_TRUE, MQE_FALSE}

Create a queue manager

When creating a queue manager, a number of parameters can be specified.

Java

First, create the QueueAdminMsg object. This needs to be primed using code to setup the origin queueManagerAdmin reply etc.

```
MQeFields parms = new MQeFields();
MQeFields queueManagerParameters = new MQeFields();
queueManagerParameters.putAscii(MQeQueueManager.Name, "MyQmgrName");
parms.putFields(MQeQueueManager.QueueManager, queueManagerParameters);

MQeFields registryParameters = new MQeFields();
registryParameters.putAscii(MQeRegistry.DirName, "c:\MyRegLocation");
parms.putFields(MQeQueueManager.Registry, registryParameters);

String queueStore = "MsgLog:" + java.io.File.separator + "queues";
MQeQueueManagerConfigure qmConfig = new MQeQueueManagerConfigure(parms, queueStore);

qmConfig.defineQueueManager();
qmConfig.defineDefaultSystemQueue();
qmConfig.defineDefaultDeadLetterQueue();
qmConfig.defineDefaultAdminReplyQueue();
qmConfig.defineDefaultAdminQueue();
qmConfig.close();
```

Figure 20. Creating the QueueAdminMsg object

C API

The information for the queue is passed in via a structure to the API. Two important points are:

- The structure is initialized using QMGR_INIT_VAL
- The properties that are set, are indicated using the opFlags elements of the structure. Each property has a corresponding bit mask – these need to be bitwise ORed together.

```
MQeQueueManagerParms qmParams  = QMGR_INIT_VAL;
MQeRegistryParms  regParams = REGISTRY_INIT_VAL;

/* String parameters for the location of the msg store */
qmParams.hQueueStore = hQueueStore;

/* Indicate what parts of the structure have been set */
qmParams.opFlags = QMGR_Q_STORE_OP;

/* ... create the registry parameters - minium that are required */
regParams.hBaseLocationName  =  hRegistryDir;

rc = mqeAdministrator_QueueManager_create(hAdministrator,
                                          &exceptBlk,
                                          &hQueueManager,
                                          hLocalQMName,
                                          &qmParams,
                                          &regParams);
```

Delete a queue manager

Java

```
MQeFields parms = new MQeFields();
MQeFields queueManagerParameters = new MQeFields();
queueManagerParameters.putAscii(MQeQueueManager.Name, "MyQmgrName");
parms.putFields(MQeQueueManager.QueueManager, queueManagerParameters);

MQeFields registryParameters = new MQeFields();
registryParameters.putAscii(MQeRegistry.DirName, "c:\MyRegLocation");
parms.putFields(MQeQueueManager.Registry, registryParameters);

String queueStore = "MsgLog:" + java.io.File.separator + "queues";
MQeQueueManagerConfigure qmConfig =
   new MQeQueueManagerConfigure(parms, queueStore);

qmConfig.deleteDefaultAdminReplyQueue();
qmConfig.deleteDefaultAdminQueue();
qmConfig.deleteDefaultDeadLetterQueue();
qmConfig.deleteDefaultSystemQueue();
qmConfig.deleteQueueManager();
qmConfig.close();
```

Figure 21. Deleting a queue manager in Java

C API

In order to delete a queue manager:

- The queue manager must be stopped
- All queues must be deleted
- All connection definitions must be deleted

Note there is no parameter structure here – just a Queue Manager handle.

```
rc =  mqeAdministrator_QueueManager_delete(hAdministrator,
                                           pExceptBlock);
if ( EC(&exceptBlk)  == MQERETURN_QUEUE_MANAGER_ERROR )
   {
      if(ERC(&exceptBlk) == MQEREASON_QMGR_ACTIVATED)
      {
      /* qmgr not been stopped - take appropriate actions */
      }
      else if(ERC(&exceptBlk) == MQEREASON_QMGR_QUEUE_EXISTS)
      {
      /* queues exist - take appropriate actions */
      }
      else if(ERC(&exceptBlk) == MQEREASON_CONNECTION_DEFINITION_EXISTS)
      {
      /* connection defs exist - take appropriate actions */
      }
      else
      {
      /* unknown error */
      }
   }
```

Figure 22. Deleting a queue manager in C

Inquire and Inquire all

In general, when inquiring on objects in WebSphere MQ Everyplace, it is possible to ask for particular parameters which are of interest using inquire, or just ask for all information using inquireAll.

Java or Administration message

In the Java interface, the presence of a field in the request message means that see the value the field that field failed in.

```
//inquire

//Request the value of description
try {
   MQeAdminMsg msg = (MQeAdminMsg) new MQeQueueManagerAdminMsg();
//Prime admin message with targetQM name, reply to queue, and so on
//refer to chapter 2 for details

   parms = new MQeFields();
   parms.putUnicode(MQeQueueManagerAdminMsg.QMgr_Description, null);

   //set the name of the queue to inquire on
   msg.setName("ExampleQM");

   //Set the action required and its parameters into the message
   msg.inquire(parms);
//Put message to target admin queue (code not shown)
```

```
//refer to chapter two for details
} catch (Exception e) {
   System.err.println("Failure ! " + e.toString());
}

//inquire all
try {
     MQeAdminMsg msg = (MQeAdminMsg) new MQeQueueManagerAdminMsg();

     //set the name of the queue to inquire on
     msg.setName("ExampleQM");

     //Set the action required and its parameters into the message
     msg.inquireAll(new MQeFields());
 } catch (Exception e) {
     System.err.println("Failure ! " + e.toString());
 }
```

C API

The example below shows how to inquire on the list of queues. This is the most complex inquire that can be performed as a vector of structures is returned. All these structures must be freed as shown below. This queue info structure contains three strings and a MQeQueueType. The first two strings are the QueueQueueManager Name and the QueueName. Both of these strings must be freed. Then there is the Java Class Name - this is a constant string and therefore need not be freed. Finally there is a primitive MQeQueueType.

The Queue Info structure must be freed using the mqeMemory_free function. Please see the API Reference for more information on the mqeMemory function.

As well as information on queues, a vector of connection definitions can be returned. This should also be freed when it has been processed.

```
MQeQueueManagerParms qmParms = QMGR_INIT_VAL;
qmParms.opFlags |= QMGR_QUEUES_OP;
rc = mqeAdministrator_QueueManager_inquire(hAdministrator,
                                           &exceptBlk,
                                           &qmParms);
if (MQERETURN_OK == rc) {
   /* This has returned a Vector of information */
   /* blocks about the queues */
   MQeVectorHndl hListQueues = qmParms.hQueues;
   MQEINT32 numberQueues;

   rc = mqeVector_size(hListQueues,&exceptBlk,&numberQueues);
   if (MQERETURN_OK == rc) {
      MQEINT32 count;
      /* Loop round the array to get the information */
      /* about the queues */
      for (count=0;count<numberQueues;count++) {
         MQeQMgrQParms *pQueueInfo;
         rc = mqeVector_removeAt(hListQueues,
                                 &exceptBlk,
                                 &pQueueInfo,
```

```
                                count);
            if (MQERETURN_OK == rc) {
               /* Queue QueueManager - FREE THIS STRING when done */
               MQeStringHndl hQMgrName = pQueueInfo->hOwnerQMgrName;
               /* QueueName - FREE THIS STRING*/
               MQeStringHndl hQueueName = pQueueInfo->hQMgrQName;
              /* A Constant String matching the Java Class Name */
              /* for this queue one of
               * MQE_QUEUE_LOCAL
               * MQE_QUEUE_REMOTE
               * MQE_QUEUE_ADMIN
               * MQE_QUEUE_HOME_SERVER
               */
               MQeStringHndl hQueueClassName = pQueueInfo->hQueueType;

               /* Will be set from MQeQueueType */
               MQeQueueType queueType = pQueueInfo->queueExactType;

               (void)mqeMemory_free(&exceptBlk,pQueueInfo);

            }
         }
      }

      /* the vector needs to be freed as well */
      mqeVector_free(hListQueues,&exceptBlk);
   }
```

Update

Java/Administration message

```
//Set name of resource to be managed
try {
   MQeAdminMsg msg = (MQeAdminMsg) new MQeQueueManagerAdminMsg();

   msg.setName("ExampleQM");

   //Change the value of description
   parms = new MQeFields();
   Parms.putUnicode(MQeQueueManagerAdminMsg.QMgr_Description,
                    "Change description ...");

   //Set the action required and its parameters into the message
   msg.update(parms);
} catch (Exception e) {
   System.err.println("Failure ! " + e.toString());
}
```

C API

This shows how to update the description. Note that the queues and so on, can not be updated, via this API - they must be done via the specific Queue update methods.

Updates of the Description, ChannelTimeout and QueueStore are allowed. QueueStore changes will only take effect for any news queues that are created.

```
MQeQueueManagerParms qmParms = QMGR_INIT_VAL;
qmParms.opFlags | = QMGR_DESC_OP;
qmParms.hDescription = hNewDescription;
rc = mqeAdministrator_QueueManager_update(hAdministrator,
                                          &exceptBlk,
                                          &qmParms);
```

Add alias

Note that it is not possible to chain aliases together. So QM1 can't be an alias for QM2, which itself is an alias for QM3.

Java

In Java, queue manager aliases are manipulated using the MQeConnectionAdminMsg. Refer to the Connection section for more information.

C API

The real name of the queue manager is hRealTargetQMname, and the alias to this is hAliasName. Note that these strings will be duplicated internally, so could be freed if not required elsewhere.

```
rc =  mqeAdministrator_QueueManager_addAlias(hAdministrator,
                                             &exceptBlk,
                                             hAliasName,
                                             hRealTargetQMName);
```

Remove alias

Java

In Java, queue manager aliases are manipulated using the MQeConnectionAdminMsg. Refer to the Connection section for more information.

C API

Removes the Alias hAliasName. An error is returned if this is not present.

```
rc = mqeAdministrator_QueueManager_removeAlias(hAdministrator,
                                               &exceptBlk,
                                                hAliasName);
```

List alias names

Java

In Java, queue manager aliases are manipulated using the MQeConnectionAdminMsg. Refer to the Connection section for more information.

C API

Lists all aliases, into a new MQeVector. These are the Alias names. Note the vector being freed, its contents will automatically be freed.

```
MQeVectorHndl hAliasList;

rc = mqeAdministrator_QueueManager_listAliasNames(hAdministrator,
                                                  &exceptBlk,
                                                  &hAliasList);
if (MQERETURN_OK == rc) {
   /* do processing */
   rc = mqeVector_free(hAliasList,&exceptBlk);
}
```

IsAlias

Java

In Java, queue manager aliases are manipulated using the MQeConnectionAdminMsg. Refer to the Connection section for more information.

C API

```
MQEBOOL isAlias;
rc =  mqeAdministrator_QueueManager_isAlias(hAdministrator,
                                            &exceptBlk,
                                             hName,
                                            &isAlias);
if (isAlias==MQE_TRUE) {
   /* name is alias */
}
```

Chapter 7. Administering local queues

This chapter explains how to administer local queues, from creation to deletion. Further chapters will explain how to administer Remote Queue Definitions, Store and Forward Queues, and Home Server Queues. A lot of the concepts and methods of administering queues are the same. These concepts will be explained in this chapter.

General notes

Local queues, as the name suggests, are local to the owning queue manager. The name of a queue is formed from the target queue manager name, for a local queue this is the name of the queue manager that owns the queue and a unique name for the queue on that queue manager. These two components of a queue name have Ascii values. The method `setName(String, String)` can be used to set the `QueueName` and the owning `QueueManagerName` in the administration message.

Java

The simplest type of queue is a local queue, managed by class `MQeQueueAdminMsg`. For other types of queue there is a corresponding administration message that inherits from `MQeQueueAdminMsg`. The `MQeQueueAdminMsg` inherits from the `MQeAdminMsg`.

The following actions are applicable on queues:

- `MQeAdminMsg.Action_Create`
- `MQeAdminMsg.Action_Delete`
- `MQeAdminMsg.Action_Inquire`
- `MQeAdminMsg.Action_InquireAll`
- `MQeAdminMsg.Action_Update`
- `MQeQueueAdminMsg.Action_AddAlias`
- `MQeQueueAdminMsg.Action_RemoveAlias`

Note: For all administration messages, information relating to the destination queue manager must be set. This is referred to in the examples below as priming the administration message. The examples show how to create the administration message to achieve the require result. These messages need to then be sent, and the admin reply messages checked as required.

C codebase

All administration is done via the administration API, which are of the form:

```
MQERETURN MQEPUBLISHED mqeAdministrator_queuetype_action();
```

Where action can be on of the following:

create	Create a Queue
delete	Delete a Queue
update	Update the properties of a queue

inquire	Inquire upon the properties of a queue
listAliasName	List all the Queue Aliases
addAlias	Add a Queue Alias
removeAlias	Remove a Queue Alias

QueueType can be one of the following:

- LocalQueue
- SyncRemoteQueue
- AsyncRemoteQueue
- AdminQueue
- HomeServerQueue

For the create, update, and inquire calls, a structure is passed in as a parameter. There is a general structure for elements that are applicable to all queues. For more specialized forms of queues, such as HomeServer, there are structures which are composed of a reference to the general structure plus additional information. For more information, refer to Chapter 4, “Administration using the administrator API”, on page 55.

Local queue properties

Queues have a number of properties, which are listed below. Information about these properties is passed either via discrete API parameters or configuration structures (MQeFields) objects.

The first list shows all the possible queue properties and indicates which are available in the code bases. All other queues will have these properties also.

Table 15. Queue properties available in each code base

Property	Description	Java	Native	Read/Write
Queue name	Identifies the name of the local queue	Yes	Yes	Read (write on create)
Local qMgr	The name of the local queue manager owning the queue	Yes	Yes	Read (write on create)
Adapter	The class (or alias) of a storage adapter that provides access to the message storage medium (see Storage adapters on page 116)	Yes	No – only one adapter in codebase	Read

Table 15. Queue properties available in each code base (continued)

Property	Description	Java	Native	Read/Write
Alias	Alias names are optional alternative names for the queue (see below)	Yes	Yes	Read/Write
Attribute rule	The attribute class (or alias) associated with the security attributes of the queue (for more details see later in this chapter)	Yes	No	Read/Write
Authenticator	The authenticator class (or alias) associated with the queue (for more details see later in this chapter)	Yes	No	Read/Write
Class	The class (or alias) used to realize the local queue	Yes	No	Read
Compressor	The compressor class (or alias) associated with the queue (for more details see later in this chapter)	Yes	No	Read/Write
Cryptor	The cryptor class (or alias) associated with the queue (for more details see later in this chapter)	Yes	No	Read/Write
Description	An arbitrary string describing the queue	Yes	Yes	Read/Write
Expiry	The time after which messages placed on the queue expire	Yes	Yes	Read/Write

Table 15. Queue properties available in each code base (continued)

Property	Description	Java	Native	Read/Write
Maximum depth	The maximum number of messages that may be placed on the queue	Yes	Yes	Read/Write
Maximum message length	The maximum length of a message that may be placed on the queue	Yes	Yes	Read/Write
Message store	The class (or alias) that determines how messages on the local queue are stored	Yes	No – only one message store available	Read (write on create)
Path	The location of the queue store	Yes	Yes	Read
Priority	The default priority associated with messages on the queue	Yes	Yes	Read/Write
Rule	The class (or alias) of the rule associated with the queue; determines behavior when there is a change in state for the queue	Yes	No – rules handled on global level	Read/Write
Target registry	The target registry to be used with the authenticator class (that is, None, Queue, or Queue manager)	Yes	No	Read/Write

Java

The parameters in Java are passed in using MQeFields objects. The values are passed using field elements of specific types.

The field names are as follows. All the symbolic names are public static final static Strings on the `MQeQueueAdminMsg` class.

Table 16.

Element type	Field name constants		Notes
	Symbolic	Value	
Unicode	Queue_CreationDate	qcd	
Int	Queue_CurrentSize	qcs	
Unicode	Queue_Description	qd	
Long	Queue_Expiry	qe	
Ascii	Queue_FileDesc	qfd	
Int	Queue_MaxMsgSize	qms	If no limit, use Queue_NoLimit (which is -1)
Int	Queue_MaxQSize	qmqs	If no limit, use Queue_NoLimit (which is -1)
Ascii	Queue_Mode	qm	Possible values are given by the constants: Queue_Asynchronous Queue_Synchronous
Byte	Queue_Priority	qp	Between 0 and 9 inclusive
Ascii array	Queue_QAliasNameList	qanl	
Ascii	Queue_QMgrName	qqmn	
Ascii	Queue_AttrRule	qar	
Ascii	Queue_Authenticator	qau	
Ascii	Queue_Compressor	qco	
Ascii	Queue_Cryptor	qcr	
Byte	Queue_TargetRegistry	qtr	Possible values are given by the constants: Queue_RegistryNone Queue_RegistryQMgr Queue_RegistryQueue
Ascii	Queue_Rule	qr	

C parameters

All the C parameters are passed in using a parameter structure. This structure needs to be initialized before it can be used by setting it to LOCAL_Q_INIT_VAL.

Table 17. C parameters

Element type	Element name	Description
MQEINT32	opFlags	Flags to indicate what parts of this structure have been set/requested
MQeStringHndl	hDescription	Description of the queue

Table 17. C parameters (continued)

Element type	Element name	Description
MQeStringHndl	hFileDesc	File Description for the Message Store (Read/Create/Write)
MQeVectorHndl	hQAliasNameList	Alias List
MQEINT64	queueExpiry	Queue Expiry
MQEINT64	queueCreationDate	Queue Creation Date
MQEINT32	queueMaxMsgSize	Queue Max Message Size
MQEINT32	queueMaxQSize	Maximum Number of messages on the queue
MQEINT32	queueCurrentSize	Current size of the Queue (all msg states)
MQEBOOL	queueActive	Indication of the Queue's state
MQEBYTE	queuePriority	Priority of messages on the queue

Create a local queue

When creating a queue, a number of parameters can be specified. In this example a queue is created, with a maximum size of 200 messages, expiry time of 20000 ms, and a description.

Administration message

First of all create the `QueueAdminMsg` object. This needs to be primed using the code introduced in Chapter 4, "Administration using the administrator API", on page 55, to set up the origin queue manager administration reply.

```
/* Create an empty queue admin message and parameters field */
MQeQueueAdminMsg msg = new MQeQueueAdminMsg();

MQeFields parms = new MQeFields();

/** Prime message with who to reply to and a unique identifier */

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

/* Add any characteristics of queue here, otherwise */
/* characteristics will be left to default values. */
parms.putUnicode( MQeQueueAdminMsg.Queue_Description,  description);

parms.putInt32(MQeQueueAdminMsg.Queue_MaxQSize,200);
parms.putInt32(MQeQueueAdminMsg. Queue_Expiry, 20000);_

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

Figure 23. Create a local queue

Once the Admin message has been created, it needs to be sent to the local admin queue, as described in Chapter 4, “Administration using the administrator API”, on page 55.

Constructor

C API

The information for the queue is passed in via a structure to the API. Two important points are:

- The structure is initialized using `LOCAL_Q_INIT_VAL`
- The properties that are set are indicated using the opFlags elements of the structure. Each property has a corresponding bit mask, which needs to be ORed together. Omitting the `QUEUE_DESC_OP` means that the queue does not have its description set, even though a value was present in the structure.

```
MQeLocalQParms localQParms = LOCAL_Q_INIT_VAL;

localQParms.queueMaxQSize  = 200;
localQParms.queueExpiry    = 20000;
localQParms.queueDescription = hDescription;
//this is an MQeStringHndl

localQParms.opFlags = QUEUE_MAX_Q_SIZE_OP | QUEUE_EXPIRY_OP | QUEUE_DESC_OP;

      rc = mqeAdministrator_LocalQueue_create(hAdministrator,
                                              &exceptBlk,
                                              hLocalQueueName,
                                              hLocalQMName,
                                              &localQParms);
   }
```

Figure 24. Create a local queue in C

Delete

Before a queue is deleted, it must be empty. Create a new administration message and set the delete action.

Administration message

```
/* Create an empty queue admin message and parameters field */
MQeQueueAdminMsg msg = new MQeQueueAdminMsg();

MQeFields parms = new MQeFields();

/** Prime message with who to reply to and a unique identifier */

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

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

Figure 25. Deleting a queue in Java

C API

Deletion of queue requires that the queue be empty of messages.

Note there is no parameter structure here – just the QueueName and QueueManager name.

```
 rc = mqeAdministrator_LocalQueue_delete(hAdministrator,
                                         &exceptBlk,
                                         hLocalQueueName,
                                         hLocalQMName);
if (   EC(&exceptBlk)  == MQERETURN_QUEUE_ERROR
      && ERC(&exceptBlk) == MQEREASON_QMGR_QUEUE_NOT_EMPTY) {
      /* queue not empty - take appropriate actions */
    }
   }
```

Figure 26. Deleting a queue in C

Add alias

Queues can be known by multiple names or aliases. If you try to add an alias that already exists, you will get an error back.

Administration message

To add an alias name to a queue, use the addAlias method on the MQeQueueAdminMsg. With admin messages multiple add and remove alias operations can be done in one admin message.

```
/* Create an empty queue admin message and parameters field */
MQeQueueAdminMsg msg = new MQeQueueAdminMsg();

/** Prime message with who to reply to and a unique identifier
 *  and set the name of the QueueManager and Queue
 */

/* Add a name that will be the alias of this queue */
msg.addAlias( "Fred" );

/* Set the admin action to update the queue */
msg.update( parms );
```

Figure 27. Adding an alias to a queue in Java

C API

Use the `addAlias()` method to add an alias name. Note that aliases have to be added one at a time. For other types of queues, such as Remote Queues, the format of the API remains the same, just change `LocalQueue` to, for example, `SyncRemoteQueue`.

```
rc = mqeAdministrator_LocalQueue_addAlias(hAdministrator,
                                          &exceptBlk,
                                          hLocalQueueName,
                                          hLocalQMName,
                                          hAliasName);
if (   EC(&exceptBlk) == MQERETURN_NOTHING_TO_DO
    && ERC(&exceptBlk) ==MQEREASON_DUPLICATE ) {
    /* already has alias */
}
```

Figure 28. Adding an alias to a queue in C

List aliases

Use the `listAlias()` method to list the aliases that you use.

Administration message

To get a list of Alias Names using Administration Messages, use the inquire action and specify a field of Queue_QAliasNameList in the parameters Fields Object.

C API

A list of aliases can be obtained from the C API by using the following API. Note that the Vector must be freed after use.

```
if (MQERETURN_OK == rc) {
    MQeVectorHndl hVectorAliases;
    rc = mqeAdministrator_LocalQueue_listAliasNames(hAdministrator,
                                                    &exceptBlk,
                                                    hLocalQueueName,
                                                    hLocalQMName,
                                                    &hVectorAliases);


    /* process the aliases vector here */

    rc = mqeVector_free(hVectorAliases,&exceptBlk);

}
```

Figure 29. Obtaining a list of aliases in C

Remove alias

Note that removing an alias could potentially alter the routing of messages. Therefore, this operation should be treated with care.

Administration message

```
/* Create an empty queue admin message and parameters field */
MQeQueueAdminMsg msg = new MQeQueueAdminMsg();

/* Prime message with who to reply to and a unique identifier
/* and set the name of the QueueManager and Queue  */

/* Specify the alias of the queue to be removed */
msg.removeAlias( "Fred" );

/* Set the admin action to update the queue */
msg.update( parms );
```

Figure 30. Removing an alias in Java

C API

```
rc = mqeAdministrator_LocalQueue_removeAlias(hAdministrator,
                                             &exceptBlk,
                                             hLocalQueueName,
                                             hLocalQMName,
                                             hAliasName);
if (   EC(&exceptBlk) == MQERETURN_NOTHING_TO_DO
    && ERC(&exceptBlk) ==
      MQEREASON_NO_SUCH_QUEUE_ALIAS ) {
    /* alias doesn't exist */
}
```

Figure 31. Removing an alias in C

Update

Some of the properties of a queue can be updated. Note that these are only those properties which are marked as writable in the above table. A similar technique is used to update and inquire upon other types of queues, such as remote and home server queues.

Administration message

The parameter field object needs to be set with field elements that need to be updated.

```
/* Create an empty queue admin message and parameters field */
MQeQueueAdminMsg msg = new MQeQueueAdminMsg();

/** Prime message with who to reply to and a unique identifier
 *  and set the name of the QueueManager and Queue
 */
MQeFields params = new MQeFields();


/* Add a new description for the queue */
msg.putAscii(MQeQueueAdminMsg.Queue_Descrpition,"New Description");
/* Set the admin action to update the queue */
msg.update( parms );
```

Figure 32. Updating the propeties of a queue in Java

C API

In a similar manner to creating the Queue, the parameter structure needs to be set with the details to update. For example, to update the description of the queue:

```
MQeLocalQParms localQParms = LOCAL_Q_INIT_VAL;

localQParms.queueDescription = hDescription; //MQeStringHndl

localQParms.opFlags |= QUEUE_DESC_OP;

      rc = mqeAdministrator_LocalQueue_update(hAdministrator,
                                              &exceptBlk,
                                              hLocalQueueName,
                                              hLocalQMName,
                                              &localQParms);
   }
```

Figure 33. Updating the propeties of a queue in C

Inquiry

It is possible to inquire upon the properties of queue. This is achieved by using the inquire action. The details that are required are set. If using an administration message, the administration reply message contains a fields object with the required information. When using the API, a structure will be filled out with the requested information.

Administration message

There are two ways of inquiring on a queue, either inquire or inquireAll. `InquireAll` will return a Fields object in the admin reply message.

```
/* Create an empty queue admin message and parameters field*/
 MQeQueueAdminMsg msg = new MQeQueueAdminMsg();
```

```
/*Prime message with who to reply to and a unique identifier
 * Set the admin action to get all characteristics of queue manager.
 */
msg.inquireAll(new MQeFields());

/* get message back from the admin reply queue to match */
/* and retrieve the results from the reply message */
```

The fields object that is returned in the administration reply message is populated with all of the properties of the queue. To get access to a specific value use the field labels as in the property table above. For example, to get at the queue description, assuming `respMsg` is the administration reply message:

```
// all on one line
String description = respMsg.getOutputFields().
  getAscii(com.ibm.mqe.administration.Queue_Description)
```

Instead of requesting all the properties of a queue, only certain ones can be requested and returned. If, for example, only the description is required the following can be used.

```
MQeFields requestedProperties = new MQeFields();
requestedProperties.putAscii(Queue_Description);
msg.inquire(requestedProperties)

/* Retrieve the administration reply */
/* message from the relevant queue */
/* Then retrieve the returned MQeFields */
/* object from this message */
MQeFields  outputFields = respMsg.getOutputFields();
```

Figure 34. Inquiring on a queue in Java

Output fields now contains the field Queue_Description only.

C API

The API takes the same parameter structure that the other APIs such as create take. To specify the elements that are of interest set the opFlags accordingly. To get, for example, the queue maximum depth, expiry, and description, set the opflags as follows:

```
MQeLocalQParms params = LOCAL_Q_INIT_VAL;

params.opflags = QUEUE_MAX_Q_SIZE_OP | QUEUE_EXPIRY_OP | QUEUE_DESC_OP;

rc = mqeAdministrator_LocalQueue_inquire(hAdministrator,
                                         &exceptBlk,
                                         hQueueName,
                                         hQueueMgrName,
                                         &params);
if (MQERETURN_OK == rc) {
   MQEINT64 queueExpiry = params.queueExpiry;
   MQEINT32 queueMaxSize = params.queueMaxQSize;
   MQeStringHndl queueDescription = params. hDescription;
}
```

Figure 35. Inquiring on a queue in C

Message store or storage adapter specification

The queue uses a queue store adapter to handle its communications with the storage device. Adapters are interfaces between WebSphere MQ Everyplace and hardware devices, such as disks or networks, or software, such as databases. Adapters are designed to be pluggable components, allowing the queue store to be easily changed.

All types of queue other than those that are remote and synchronous require a message store to store their messages. Each queue can specify what type of store to use, and where it is located. The queue characteristic Queue_FileDesc is used to specify the type of message store and to provide parameters for it. The file descriptor takes the form:

- adapterClass:adapterParameters or
- adapterAlias:adapterParameters

For example assuming MsgLog has been defined as an WebSphere MQ Everyplace alias:

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

A number of storage adapters are provided and include:

- MQeDiskFieldsAdapter to store messages on a file system
- MQeMemoryFieldsAdapter to store messages in memory
- Other storage adapters can be found in package com.ibm.mqe.adapters

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

If a message store is not defined when creating a queue, the default is to use the message store that was specified when the queue manager was created.

Note that under the C codebase, there is only one supplied message store, and one adapter, therefore the format of the QueueStore is fixed the MsgLog is left as a placeholder for future expansion.

Examples where this option would be used are:

- When you want to use the MemoryFieldsAdapter, to store data in memory and not on disk
- Alternative Message Stores are provided, such as the ShortFilename message store for 4690

Take the following into consideration when setting the *Queue_FileDesc* field:

- Ensure that the correct syntax is used for the system that the queue resides on. For instance, on a windows system use "\" as a file separator on UNIX systems use "/" as a file separator. In some cases it may be possible to use either but this is dependent on the support provided by the JVM (Java Virtual Machine) that the queue manager runs in. As well as file separator differences, some systems use drive letters like Windows NT whereas others like UNIX do not.
- On some systems it is possible to specify relative directories (".\") on others it is not. Even on those where relative directories can be specified, they should be used with great caution as the current directory can be changed during the lifetime of the JVM. Such a change causes problems when interacting with queues using relative directories.

Chapter 8. Administering remote queues

This chapter describes the administration of remote queue definitions.

Terminology

Consider two QueueManagers, QM_A and QM_B. There is a queue on QM_B called Queue_One – which is a local queue on QM_B. Initially this is only accessible to the QM_B, QM_A has no access to it. In order to get access to Queue_One QM_A needs a Remote Queue Definition, usually abbreviated to RemoteQueue. When referring to the Remote Queue Definition, the term Queue QueueManager is used to refer to QM_B, that is, the QueueQueueManager is the QueueManager upon which the LocalQueue referenced by the RemoteQueueDefinition resides.

In summary, remote queues are references to queues that reside on a queue manager that is remote to where the remote queue is defined. The remote queue has the same name as the target queue but the remote queue definition identifies the owning or target queue manager of the real queue.

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

For asynchronous queues if max message size on the remote definition is greater than that on the real queue, the message is accepted for storage on the remote queue but may be rejected when moved to the real queue. The message is not lost, it remains on the remote queue but cannot be delivered.

If the security characteristics for a synchronous queue do not match, WebSphere MQ Everyplace negotiates with the real queue to decide what security characteristics should be used. In some cases, the message put is successful, in others an attribute mismatch exception is returned.

Administering remote queues

The contants provided for setting the Transport and Transporter XOR parameter are provided for backward compatability. The structure for Asynchronous Remote Queues is the same, apart from the name.

```
typedef struct MQeRemoteAsyncQParms
{
   MQeQueueParms     baseParms;
/**< Queue Parms Structure - for general parameters */
   MQeStringHndl     hQTransporterClass;
 /**< Transport Class (Read/Write) */
} MQeRemoteAsyncQParms;
```

Synchronous and asynchronous

The difference between the two types of remote queue definition has been covered in the WebSphere MQ Everyplace Application Programming Guide. To summarize:

Synchronous

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

Asynchronous

Asynchronous remote queues are queues that move messages to remote queues but cannot remotely retrieve messages. When message are put to the remote queue, the messages are temporarily stored locally. When there is network connectivity, transmission has been triggered and rules allow, an attempt is made to move the messages to the target queue. Message delivery will be once-only assured delivery. This allows applications to operate on the queue when the device is off-line. Consequently, asynchronous queues require a message store in order that messages can be temporarily stored at the sending queue manager whilst awaiting transmission.

Note: In the Java codebase, the mode of an instance of the MQeRemoteQueue class is set to Queue_Synchronous or Queue_Asynchronous to indicate whether the queue is synchronous or asynchronous. In the native codebase, two distinct sets of APIs are used to create and administer synchronous and asynchronous remote queues.

Figure 36 on page 97 shows an example of a remote queue set up for synchronous operation and a remote queue setup for asynchronous operation.

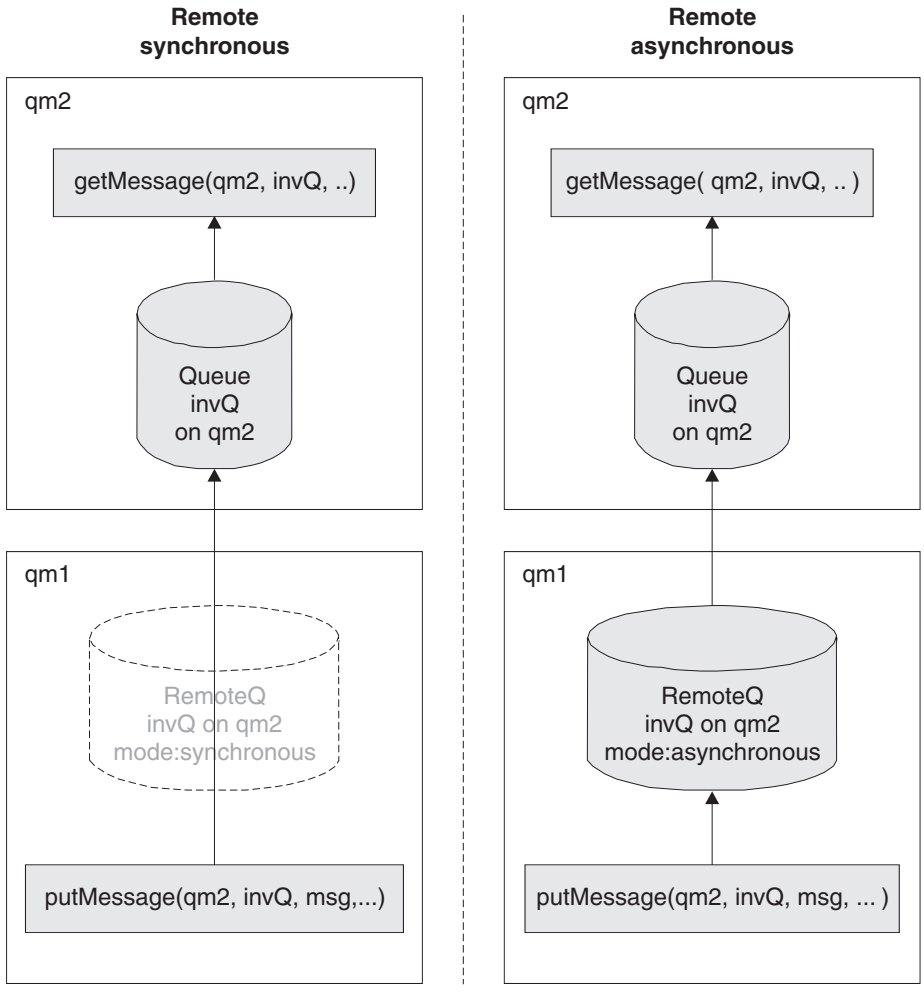

Figure 36. Remote queue

In both the synchronous and asynchronous examples queue manager qm2 has a local queue invQ.

In the synchronous example, queue manager qm1 has a remote queue definition of queue invQ. invQ resides on queue manager qm2. The mode of operation is set to synchronous.

An application using queue manager qm1 and putting messages to queue qm2.invQ establishes a network connection to queue manager qm2 (if it does not already exist) and the message is immediately put on the real queue. If the network connection cannot be established then the application receives an exception that it must handle.

In the asynchronous example, queue manager qm1 has a remote queue definition of queue invQ. invQ resides on queue manager qm2. The mode of operation is set to asynchronous.

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

Setting the operation mode

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

Asynchronous queues require a message store to temporarily store messages. Definition of this message store is the same as for local queues.

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

Creating a remote queue

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

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

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

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

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

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

    parms.putByte( MQeQueueAdminMsg.Queue_Mode,
                queueMode);

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

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

  /*
   *  Put the admin message to the admin
  queue (not assured delivery)
   *  on the target queue manager
   */
  localQM.putMessage( targetQMgr,
                      MQe.Admin_Queue_Name,
                      msg,
                      null,
                      0);
}
```

For synchronous operation, the queue characteristics for inclusion in the remote queue definition can be obtained using *queue discovery*.

Creating a C parameter structure

The parameter structure of the Synchronous Remote Queue, contains two elements. The first is a parameter structure of the same type as that used for local queues (MQeQueueParms). The second is the Transporter for use with this Queue. The remote Queue shares the properties of the local queue, hence the reason for the local queue structure.

Note that the opFlags parameter for specifying what elements of the structure have been set is in the MQeQueueParms structure.

```
typedef struct MQeRemoteSyncQParms
{
   MQeQueueParms     baseParms;
 /*< Queue Parms Structure –for general parameters */
   MQeStringHndl     hQTransporterClass;
/*< Transporter Class (Read/Write) */
} MQeRemoteSyncQParms;
```

Create synchronous

First create the remote queue administration message.

```
MQeRemoteQueueAdminMsg msg = new AdminMsg();
MQeFields params = new MQeFields();
```

Then prime the administration message, as explained in Chapter 2, “Administration using administration messages”, on page 5. Then set the queue queue manager name.

```
msg.setName(queueQMgrName, queueName);

params.putUnicode(descriptiorn);

/* set this to be a synchronous queue */
params.putByte(MQeQueueAdminMsg.Queue_Mode,
    MQeQueueAdminMsg.Queue_Synchronous);
```

Now, set the administration action to create the queue.

```
msg.create(params);

/* send the message */
```

C codebase

This is the C API to create a sync queue. It is very similar to the Local Queue creation. Options for description, max size etc can be set just as for the local queue.

```
 MQeRemoteSyncQParms remoteSyncQParms = REMOTE_SYNC_Q_INIT_VAL;

      rc = mqeAdministrator_SyncRemoteQueue_create(hAdministrator,
                  &exceptBlk,
                  hQueueName,
                  hServerName,
                  &remoteSyncQParms);
```

Create asynchronous

```
MQeRemoteQueueAdminMsg msg = new MQeRemoteQueueAdminMsg();
MQeFields params = new MQeFields();

/* Prime the admin message */
```

```
msg.setName(queueQMgrName, queueName);

params.putUnicode(description);

/* set this to be an asynchronous queue */
params.putByte(MQeQueueAdminMsg.Queue_Mode,
   MQeQueueAdminMsg.Queue_Asynchronous);

/* Assuming that MsgLog is an established */
/* Alias set the QueueStore location */
params.putAscci(MQeQueueAdminMsg.Queue_FileDesc,
"MsgLog:c:\queuestore");

/* Set the administration action to create the queue */
msg.create(params);

/* send the message */
```

C codebase

This is the C API to create a async queue. It is very similar to the Local Queue creation. Options for description, max size etc can be set just as for the local queue.

```
MQeRemoteAsyncQParms remoteAsyncQParms = REMOTE_ASYNC_Q_INIT_VAL;

rc = mqeAdministrator_AsyncRemoteQueue_create(hAdministrator,
           &exceptBlk,BROKERTRADE_Q_NAME,
           SERVER_QM_NAME, &remoteAsyncQParms);
```

Transporter

One of the parameters of Remote Queue Definition is the transport that is in use. This can be modified if required. Usually it is set to the DefaultTransporter, com.ibm.mqe.MQeTransporter. Note that his can not be modified once the Queue has been created.

Queue aliases

The administration of aliases is the same as for LocalQueues, as the MQeRemoteQueueAdminMsg is a subclass of the MQeQueueAdminMsg.

Under C use the following APIs in the same way as for a local queue.

```
mqeAdministrator_SyncRemoteQueue_addAlias
mqeAdministrator_SyncRemoteQueue_removeAlias

mqeAdministrator_AsyncRemoteQueue_addAlias
mqeAdministrator_AsyncRemoteQueue_removeAlias
```

Chapter 9. Administering home server queues

A home-server queue definition identifies a store-and-forward queue on a remote queue manager. The home-server queue then pulls any messages that are destined for the home-server queue's local queue manager, off the store-and-forward queue. Multiple home-server queue definitions may be defined on a single queue manager, where each one is associated with a different remote queue manager.

Home-server queues normally reside on a device and are typically set to pull messages from a server whenever the device connects to the network. When a message is pulled from the server, the message is then put on the correct target local queue. If the target queue does not exist then a rule is called which allows the message to be placed on a dead letter queue.

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

- The queue name must match the name of the store-and-forward queue
- The queue manager attribute of the queue name must be the name of the home-server queue manager
- The queue manager where the home-server queue resides must have a connection configured to the home-server queue manager where the store-and-forward queue resides..

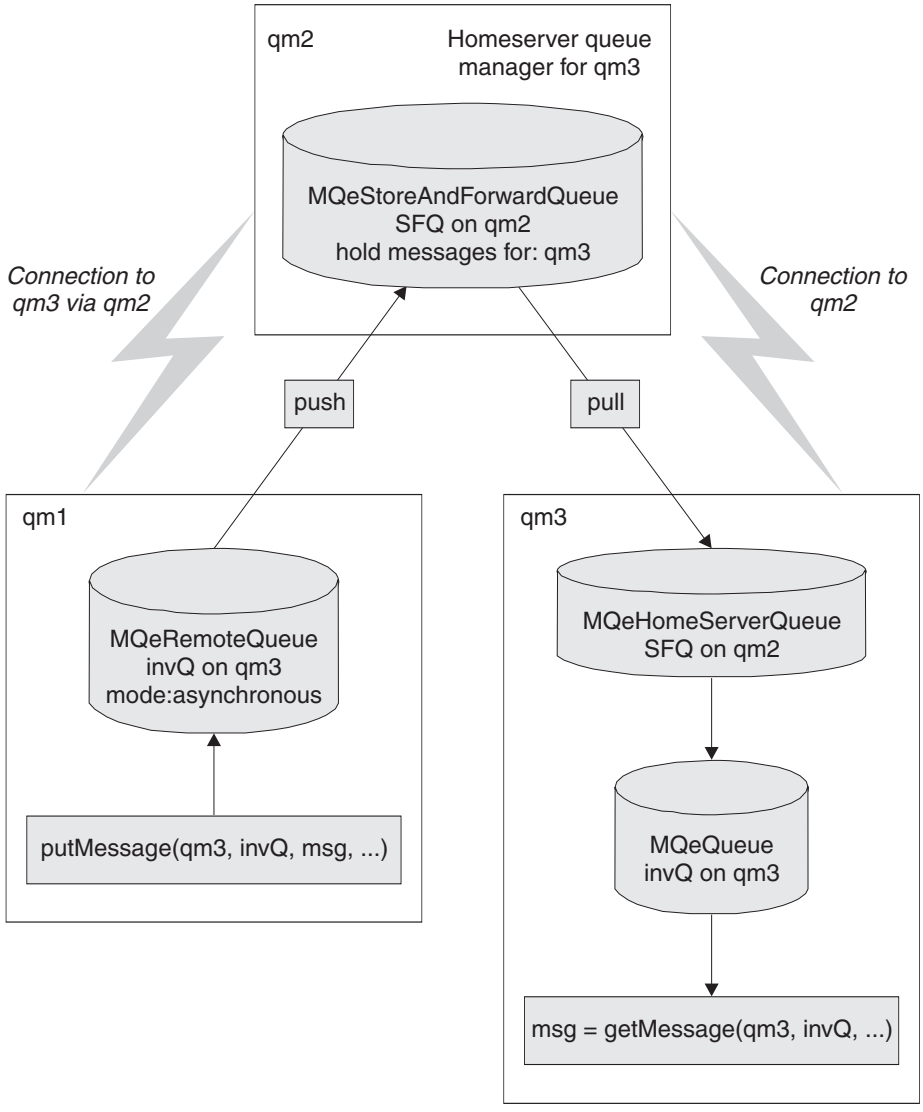

Figure 37. Home-server queue

The above diagram shows an example of a queue manager qm3 that has a home-server queue SFQ configured to collect messages from its home-server queue manager qm2. The configuration consists of:

- A home server queue manager qm2
- A store and forward queue SFQ on queue manager qm2 that holds messages for queue manager qm3
- A queue manager qm3 that normally runs disconnected and cannot accept connections from queue manager qm2

- Queue manager qm3 has a connection configured to qm2
- A home server queue SFQ that uses queue manager qm2 as its home server

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

Administration messages

The Java class extends `MQeRemoteQueueAdminMsg` which provides most of the `MQeHomeServerQueueAdminMsg` administration capability for remote queues. This class adds additional actions and constants for managing home server queues.

Home-server queues are implemented by the `MQeHomeServerQueue` class. They are managed with the `MQeHomeServerQueueAdminMsg` class which is a subclass of `MQeRemoteQueueAdminMsg`. The only addition in the subclass is the *`Queue_QTimerInterval`* characteristic. This field is of type int and is set to a millisecond timer interval. If you set this field to a value greater than zero, the home-server queue checks the home server every `n` milliseconds to see if there are any messages waiting for collection. Any messages that are waiting are delivered to the target queue. A value of `0` for this field means that the home-server is only polled when the MQeQueueManager.triggertransmission method is called

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

Message transmission

Java

A home server queue can be requested to check for pending messages:

- By setting a poll interval in field `Queue_QTimerInterval`, that causes a regular check for messages on the server whilst connectivity is available. When network connectivity is not available or a network outage occurs, the polling will stop and not restart until the queue is triggered using the `MQeQueueManager.triggerTransmission()` method.
- When the `MQeQueueManager.triggerTransmission()` method is called.

Home-server queues have an important role in enabling devices to receive messages over client-server channels particularly in environments where it is not possible for a server to establish a connection to a device.

For information on basic administration concepts, refer to Chapter 2, “Administration using administration messages”, on page 5. Also for information on managing queues, that is `MQeQueueAdminMsg` and `MQeRemoteQueueAdminMsg`, refer to Chapter 7, “Administering local queues”, on page 79 and Chapter 8, “Administering remote queues” , on page 95.

C

The C codebase does not have background threads. Therefore, the HomeServerQueue will only pull down messages from a Store and Forward Queue when `mqeQueueManager_triggerTransmission` is called. The trigger transmission method will only return when an attempt has been made to transmit all messages.

Creating

Administration message

The home server queue is created in a similar manner to other queues. It is generally recommended not to use a time interval but to control the transmission using `triggerTransmission`.

C API

```
if (MQERETURN_OK == rc) {
   MQeHomeServerQParms homeServerQParms = HOME_SERVER_Q_INIT_VAL;

   rc = mqeAdministrator_HomeServerQueue_create(hAdministrator,
                                                &exceptBlk,
                                                hQueueName,
                                                hServerName,
                                                &homeServerQParms);
```

Figure 38.

Administration is performed using the following APIs.

`mqeAdministration_HomeServerQueue_action()`

The MQeHomeServerQParms structure is used to pass parameters. Note that the first element is the MQeRemoteSyncQParms structure. This maps onto the MQeHomeServerQueueAdminMsg inheriting function from the MQeRemoteQueueAdminMsg.

```
typedef struct MQeHomeServerQParms
{
   MQeRemoteSyncQParms     remoteQParms;
/**<Remote Queue Parameters to be filled in  */
   MQEINT64              qTimerInterval;
/**<Time Interval - for Java compatibility only*/
} MQeHomeServerQParms;
```

Chapter 10. Administering store and forward queues

This chapter will explain how to administer store and forward queues, from creation to deletion, proceeding through the different administration actions that can be performed upon them.

General notes

Since there is no concept of a store and forward queue in C all of the following information relates to the Java codebase. The store and forward queue is managed by class MQeStoreAndForwardQueueAdminMsg which inherits from MQeQueueAdminMsg.

Store-and-forward queue

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

- Forward messages either to the target queue manager, or to another queue manager between the sending and the target queue managers. In this case the store-and-forward queue pushes messages either to the next hop or to the target queue manager
- Hold messages until the target queue manager can collect the messages from the store-and-forward queue. This can be accomplished using a *home-server* queue, as described in Chapter 9, “Administering home server queues”, on page 103. Using this approach messages are *pulled* from the store-and-forward queue.

Store-and-forward queues are implemented by the MQeStoreAndForwardQueue class. They are managed with the MQeStoreAndForwardQueueAdminMsg class, which is a subclass of MQeRemoteQueueAdminMsg. The main addition in the subclass is the ability to add and remove the names of queue managers for which the store-and-forward queue can hold messages.

Apart from the characteristics shared by all remote queues, a store-and-forward queue object also has a property identifying its set of target queue managers. The string field `Queue_QMgrNameList`, with the value `"qqmnl"`, identifies the field in an administration message representing the set of target queue managers. The value of this field is set or retrieved using `putAsciiArray()` and `getAsciiArray()` methods.

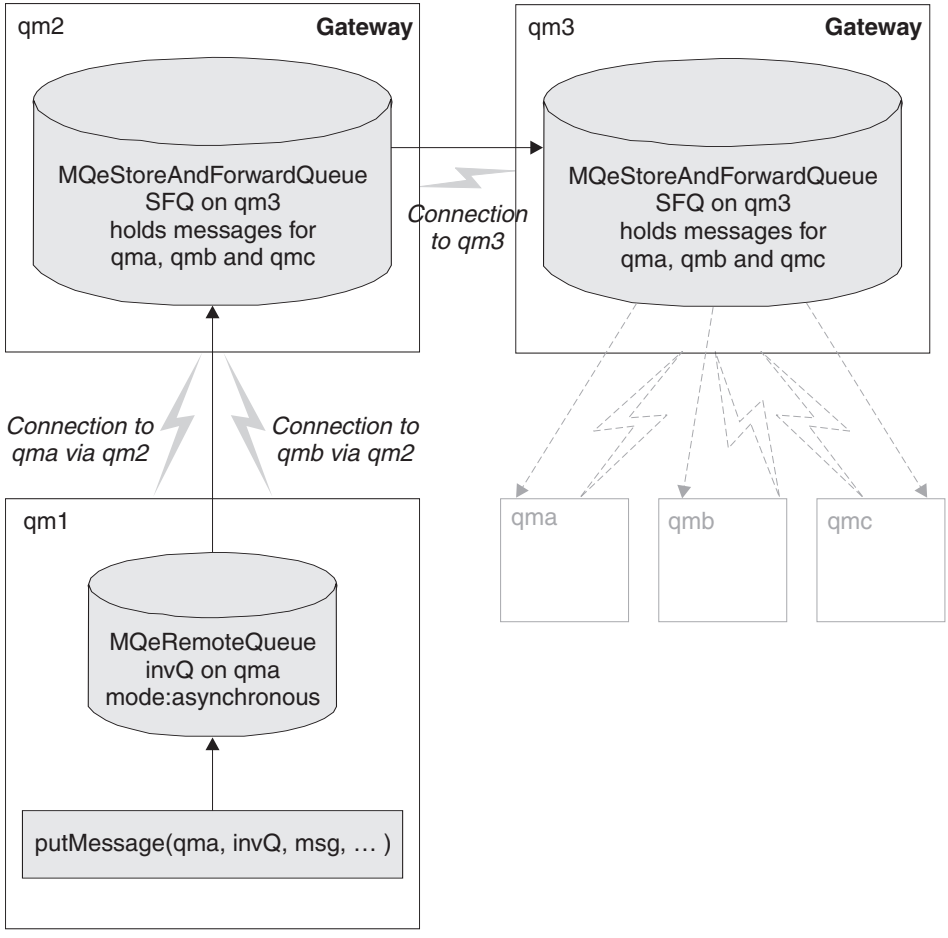

Figure 39. Store-and-forward queue

Each store-and-forward queue has to be configured to handle messages for any queue managers for which it can hold messages. Use the **Action_AddQueueManager** action, described earlier in this section, to add the queue manager information to each queue.

If you want the store-and-forward queue to push messages to the next queue manager, the queue manager name attribute of the store-and-forward queue must be the name of the next queue manager. A connection with the same name as the next queue manager must also be configured. The store-and-forward queue uses this connection as the transport mechanism for pushing messages to the next hop.

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

Figure 39 on page 108 shows an example of two store and forward queues on different queue managers, one setup to push messages to the next queue manager, the other setup to wait for messages to be collected:

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

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

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

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

In Figure 39 on page 108 queue manager qm1 has been configured to allow messages to be put to queue invQ on queue manager qma. The configuration consists of:

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

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

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

5. Queue manager qm3 has a store-and-forward queue qm3.SFQ that can hold messages destined for queue manager qma so the message is stored on that queue
6. Messages for qma remain on the store-and-forward queue until they are collected by queue manager qma. See Chapter 9, “Administering home server queues”, on page 103 for how to set this up.

Create

There are no extra parameters other than those used in creating a remote queue that can be specified for creating a store and forward queue. In this example a queue with a description is created.

Administration message

As with all queues the first action is to create the appropriate admin message object. This then needs to be followed by priming the message using the code introduced in Chapter 2, “Administration using administration messages”, on page 5.

```
/* Create an empty store and forward queue dmin message and parameters field */
MQeStoreAndForwardQueueAdminMsg msg = new MQeStoreAndForwardQueueAdminMsg();

MQeFields parms = new MQeFields();

/* Prime message stating who to reply to and a unique identifier */
/* Refer to Chapter 2, Administration using administration messages, */
/* for a definition of the user helper method primeAdminMsg(); */
primeAdminMsg( msg );

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

/* Add any characteristics of the queue here, otherwise */
/* characteristics will be left to default values. */
parms.putUnicode( MQeQueueAdminMsg.Queue_Description, description);

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

Figure 40.

Once the admin message has been created, it needs to be sent to the local admin queue.

Delete

In this example the constructor is used to set the QueueName and the QueueManager name. This is an alternative to using the setName() method on the admin message.

Administration message

As with all queues deletion requires that the queue be empty of messages. Note that there is no parameter structure here – just the QueueName and QueueManager name.

```
/* Create an empty store-and-forward */
/* queue admin message */
/* all on one line*/
MQeStoreAndForwardQueueAdminMsg msg =
    new MQeStoreAndForwardQueueAdminMsg (qMgrName, queueName);

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

/* Set the admin action to delete a queue */
msg.delete(new MQeFields() );
```

Figure 41.

Add queue manager

You can add and delete queue manager names with the `Action_AddQueueManager` and `Action_RemoveQueueManager` actions. You can add or remove multiple queue manager names with one administration message. You can put names directly into the message by setting the ascii array field `Queue_QMgrNameList`. Alternatively, you can use the `addQueueManager()` and `removeQueueManager()` methods. Each of these methods takes one queue manager name, but you can call the method repeatedly to add multiple queue managers to a message.

This action is specific to store and forward queues. In this example multiple queue manager names are added to a String array (queueManagerNames) and set into the fields object. The action and fields object are added to the message.

Administration message

```
/* Create an empty store and forward */
/* queue admin message and parameters field*/
/* all on one line*/
MQeStoreAndForwardQueueAdminMsg msg =
  new MQeStoreAndForwardQueueAdminMsg (qMgrName, queueName);

MQeFields parms = new MQeFields();

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

/* Add any characteristics of queue here, otherwise */
/* characteristics will be left to default values.*/
parms.putAsciiArray(MQeStoreAndForwardQueueAdminMsg.Queue_QMgrNameList,queueManagerNames);


/* Set the admin action to add a queue manager to a queue */
msg.putInt(MQeAdminMsg.Admin_Action,
  MQeStoreAndForwardQueueAdminMsg.Action_AddQueueManager);

/* Put the fields object into the message */
msg.putFields(MQeAdminMsg.Admin_Parms, parms);
```

Figure 42.

Remove queue manager

This action is specific to store and forward queues. In this example the helper method `removeQueueManager()` is used to remove a single queue manager.

Administration message

```
/* Create an empty store and forward queue admin message*/
MQeStoreAndForwardQueueAdminMsg msg =
  new MQeStoreAndForwardQueueAdminMsg (qMgrName, queueName);

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

/* Set the admin action to remove a queue manager */
msg.removeQueueManager(queueManagerName);
```

Figure 43.

Update

In this example the description and of a store and forward queue and the maximum number of messages allowed on the queue are updated.

Administration message

```
/* Create an empty store and forward */
/* queue admin message and parameters field */
MQeStoreAndForwardQueueAdminMsg msg = new MQeStoreAndForwardQueueAdminMsg ();

MQeFields parms = new MQeFields();

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

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

/* Add any characteristics of queue here, otherwise */
/* characteristics will be left to default values.*/
parms.putUnicode(MQeQueueAdminMsg.Queue_Description, description);
parms.putInt(MQeQueueAdminMsg.Queue_MaxQSize,10);

/* Set the admin action to update */
msg.update(parms);
```

Figure 44.

Inquire

In this example the list of queue manager names of a store and forward queue are inquired.

Administration message

```
/* Create an empty store and forward queue admin message and parameters field */
MQeStoreAndForwardQueueAdminMsg msg = new MQeStoreAndForwardQueueAdminMsg ();

MQeFields parms = new MQeFields();

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

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

/* Add any characteristics of queue here that you want to inquire.*/
parms.putAsciiArray(MQeStoreAndForwardQueueAdminMsg.Queue_QMgrNameList,new String[0]);

/* Set the admin action to inquire */
msg.inquire(parms);
```

Figure 45.

Store and forward queue attributes

Store and Forward queues have a number of attributes extra to those of remote queues – these are listed below. Information about these attributes is passed either via API parameters or configuration structures/MQeFields objects.

In Java, the queue manager name list identifies the field in the message representing a set of target queue managers.This does not occur in the native codebase.

Java

The parameters in Java are passed in using MQeFields objects. The values are passed using field elements of specific types. The field names are as follows:

Table 18.

Element type	Field label	Textual value of field label
public static final java.lang.String	Queue_QMgrNameList	″qqmnl″

Chapter 11. Connection definition

Connection definitions provide WebSphere MQ Everyplace with information on how to locate and communicate with remote queue managers. The name of a connection definition is that of the remote queue manager to which it describes a route, thus there may only be one direct connection definition for a remote queue manager. As connection definitions define the WebSphere MQ Everyplace network they are held in permanent storage in the registry and therefore persist across instances of the queue manager.

The route created using a connection definition uses an internal object called a channel as the transport mechanism to send data between two queue managers. Channels may not be accessed directly by a user but configuration decisions made for a queue manager affects the behavior of a channel. Refer to Communication Channel Security Considerations in Chapter 15, “Security”, on page 207 for more information.

At the lowest level of the communications layers is the communications adapter. The reason they are mentioned here is that it is imperative the connection definition defines the same communications adapter class as the adapter class being used by the listener on the listening queue manager. If the communications adapters are not exactly the same a successful connection will not be made.

Direct connection definition

A direct connection definition supplies information to allow the local queue manager to create a channel to a remote queue manager in the WebSphere MQ Everyplace network. The information is the actual network information for the remote queue manager and does not involve any routing via other queue managers.

There are two variants of a direct connection, these are:

Alias connection definition
: An alias connection definition provides just one piece of information, the name of an actual connection definition or another alias. One may think of these aliases as queue manager aliases, they allow an administrator to set up a connection definition to a particular queue manager which may then be referred to by another name.

MQ connection definition
: This is a specialized connection that identifies a remote queue manager as a WebSphere MQ queue manager as opposed to a WebSphere MQ Everyplace queue manager. For further information on the Bridge functionality of WebSphere MQ Everyplace refer to Chapter 13, “Administering bridge resources”, on page 129.

Indirect connection definition

You can also have an indirect connection definition:

Via connection definition

A via connection definition supplies information to allow the local queue manager to create a channel to a remote queue manager using a route via an intermediate queue manager. The intermediate queue manager(s) should be configured so they have connection definitions to either the next queue manager in the route or the final destination queue manager. It is the responsibility of the administrator to ensure that all necessary connection definitions are configured on the route.

General

In order to create, alter, or delete a connection definition it is necessary to use a specialized message, an administration message. For an overview of administration messages refer to Chapter 2, “Administration using administration messages”, on page 5. For the connection definition to create a successful connection to a remote queue manager it is necessary for the correct communications adapter, the correct network address of the listening queue manager and the correct listening location to be specified. If any of this information is incorrect it is not possible to make a connection to the remote queue manager.

Connection definition administration in Java

This section shows you how to create, alter, delete connection definitions in Java.

Creating a connection definition

In order to create a connection definition an administration message must be created and put to the administration queue. A reply must be received to indicate successful creation of a connection definition before any attempt is made to use the connection, indeterminate behavior may result if an attempt is made to use a connection before such as reply has been received.

In order to show how one might create a connection definition we shall use the `examples.config.CreateConnectionDefinition` example. A connection definition administration message has a number of methods to help create the message correctly. First of all we need to create a MQeConnectionAdminMsg:

```
MQeConnectionAdminMsg connectionMessage = new MQeConnectionAdminMsg();
```

Once we have created the connection administration message we need to set the name of the resource we wish to work on:

```
connectionMessage.setName("RemoteQM");
```

We now need to set the information in the administration message that will set the action to create and will provide the information for the route to our remote queue manager:

```
connectionMessage.create(
                "com.ibm.mqe.adapters.MQeTcpipHistoryAdapter:127.0.0.1:8082",
                null,
                null,
                "Default Channel",
                "Example connection");
```

There are a number of things to note about the information passed to the create method.

The first parameter is a colon delimited string and has a profound affect on what type of connection definition will be created. The string used in the above example will create a connection to a queue manager called RemoteQM using the communications adapter MQeTcpipHistoryAdapter running on the local machine listening at port 8082. If we had merely specified a queue manager name, for instance ″ServerQM″ then a via connection definition would have been created and we would have to either already have a connection definition for ServerQM or create one before we attempted to use the via connection definition.

The second parameter is really only useful for HTTP adapters that may run a servlet on the server. This is where you would define your servlet name which would then be passed within the HTTP header.

The third parameter allows the persistent option to be set or unset, although in reality this should be done with great care as the default values for persistence are set within the communications adapters so they are consistent with the protocol being used. For instance the MQeTcpipLengthAdapter and MQeTcpipHistoryAdapter both use persistence, that is the socket is kept open, the MQeTcpipHttpAdapter on the other hand uses a new socket for each conversation.

The fourth parameter defines the channel, this should always be set to ″Default Channel″.

The fifth parameter provides descriptive text for the connection definition.

We now need to add information to the administration message that will determine which queue manager receives the administration message.

```
connectionMessage.setTargetQMgr("LocalQM");
```

Specify that you want to receive a reply, if using the `Msg_Style_Datagram`, indicate that no reply was required. The reply indicates success or failure of the administrative action.

```
connectionMessage.putInt(MQe.Msg_Style, MQe.Msg_Style_Request);
```

The queue and queue manager that will receive the reply, this may not necessarily be the queue manager that created and sent the administration message. Using the default administration reply queue allows you to use the definition of the String provided in the MQe class. Also, the reply must arrive on the local queue.

```
connectionMessage.putAscii(MQe.Msg_ReplyToQ, MQe.Admin_Reply_Queue_Name);
connectionMessage.putAscii(MQe.MSG_ReplyToQMgr, "LocalQM");
```

A unique identifier must be added to the message before putting it onto the administration queue. This allows you to identify the appropriate reply message. Use the system time in order to do this.

```
String match = "Msg" + System.currentTimeMillis();
connectionMessage.putArrayOfByte(MQe.Msg_CorrelID, match.getByte());
```

You can now put our administration message to the default administration queue, the fourth parameter allows for an MQeAttribute to be specified with the fifth parameter allowing for an identifier that allows you to undo the put. As neither is required, specify null and zero respectively.

```
queueManager.putMessage("LocalQM", MQe.Admin_Queue_Name,
         connectionMessage, null, 0);
```

Before we can safely use the connection definition we need to ensure it has been correctly created and must therefore wait for a reply. We specified the reply should be sent to the queue manager LocalQM on the default administration reply queue. We create a filter using the correlation id so we get the correct reply:

```
MQeFields filter = new MQeFields();
filter.putArrayOfByte(MQe.Msg_CorrelID, match.getBytes());
```

Now using the filter we have created we wait for a reply message on the default administration reply queue. The return from the waitForMessage method gives a MQeMsgObject, so we cast that to an MQeAdminMsg. The fourth parameter which we have set to null may be used for an MQeAttribute, this is set to null as we have not used security during this example, the zero passed in parameter five is for a confirm ID that may be used in an undo operation, again we have not used this. The last parameter defines how long to wait in milliseconds, we are waiting for three seconds.

```
// all on one line
MQeAdminMsg response = (MQeAdminMsg)
     queueManager.waitForMessage(queueManagerName,
                    MQe.Admin_Reply_Queue_Name,
                    filter,
                    null,
                    0,
                    3000);
```

Once we have received the reply we check to make sure we have a successful return code, there is additional checking done within the example, for the purposes of this manual we just look at the successful return. As can be seen there is a useful method on the administration message which will return a return code to us for easy checking.

```
switch (response.getRC()) {
                case MQeAdminMsg.RC_Success :
                    System.out.println("connection created");
                    break;
⋮
```

We have now successfully created a connection definition to a remote queue manager.

Altering and deleting connection definitions

Connection definitions define the network for WebSphere MQ Everyplace and therefore great care should be taken when altering or deleting them. It is strongly recommended that when altering or deleting a connection definition one should ensure there is no activity on the network that may be using that connection definition.

As with creating a connection definition, in order to alter or delete a connection definition an administration message must be used. The approach is the same as for creating a connection definition, with a different action being used for the administration message. For instance in order to update a connection definition the following method should be used:

```
updateMessage.update(
"com.ibm.mqe.adapters.MQeTcpipHttpAdapter:127.0.0.1:8083",
null, null, "DefaultChannel", "Altered Example Connection");
```

In order to delete a connection definition all that is required is the resource name and the relevant action being set, so the following method is used:

```
deleteMessage.setAction(MQeAdminMsg.Action_Delete);
```

Connection definition administration in C

There is an important difference between administration available in C to that in Java. The Java product relies solely on the administration message, C provides an administration API for the user to locally administer WebSphere MQ Everyplace. More information may be found abut the administration API in Chapter 4, “Administration using the administrator API”, on page 55, this chapter assumes you have already read the chapter on administration and know how to create an administrator handle and exception block used in the calls to the administration API. This example is in the transport.c in the broker.dll for C

Before we look at the individual functions providing the API to administer the connection definition, it will be worthwhile looking at the structure containing the information about the connection definition that is passed into all the functions requiring information, that is all except the function to delete the connection definition. The MQeConnectionDefinitionParms structure is as follows:

```
MQEVERSION         version;
MQEINT32           opFlags;
MQeStringHndl      hDescription;
MQeStringHndl      hAdapterClass;
MQeStringHndl *    phAdapterParms;
MQEINT32           destParmLen;
MQeStringHndl      hAdapterCommand;
MQeStringHndl      hChannelClass;
MQeStringHndl      hViaQMName;
```

Version

This is a field for internal use only and should not be set by the user.

opFlags

On input to a function this field provides bit flags indicating the areas of the

resource that are to be administered. On output from a function if the action has been successful the flags will indicate the operations performed, if the action has failed the flags will indicate the failed component.

hDescription
The description for this connection definition.

hAdapterClass
The communications adapter class that will be used by this connection definition, currently there is just one communications adapter for C. In the MQe_Adapter_Constants.h header file there is a constant to define the class – MQE_HTTP_ADAPTER.

phAdapterParams
An array containing the network information required to connect to the remote queue manager. In an IP network this will contain the network address and IP port. The first element in the array is assumed to be the IP address, the second element is assumed to be the port number.

destParmLen
The length of the phAdapterParams array.

hAdapterCommand
This field may contain a servlet name to be included in an HTTP header.

hChannelClass
The class of channel to use, this should be set to MQE_CHANNEL_CLASS, defined in MQe_Connection_Constants.h

hViaQMName
If this connection definition defines a via connection then all other parameters should be null with this parameter containing the name of the via queue manager name.

A constant in MQe_Connection_Constant.h - CONNDEF_INIT_VAL will set the values of this structure to initial values which can then be altered as required.

Create a connection definition

In order to create a connection definition will need to call the function:

```
mqeAdministrator_Connection_create(MQeAdministratorHndl, hAdmin,
                                   MQeExceptBlock* pExceptBlock,
                                   MQeStringHndl hConnectionName,
                                   MQeConnectionDefinitionParms* pParams);
```

The third parameter will define the name of the connection definition. As stated, this must be the name of the remote queue manager to which this connection definition holds the route.

The fourth parameter is a structure holding information that is required to setup the connection definition information. Either the hViaQMName field should be set or the hAdapterClass, phAdapterParams, destParmLen, hAdapterCommand and

hChannelClass in order to create a connection definition. For instance, to create a connection definition, first create and set up an MQeConnectionDefinition parameter structure:

```
/* Create the structure and set it to the initial values */
MQeConnectionDefinitionParms parms = CONNDEF_INIT_VAL;
```

Create an MQeString to hold the name of the remote queue manager, this becomes the name of the connection definition:

```
rc = OSAMQESTRING_NEW(&error, "ServerQM", SB_STR, &hQueueMgrName);
```

Set the adapter and channel class names, these must be set to these names as these are the only classes currently supported:

```
parms.hAdapterClass = MQE_HTTP_ADAPTER;
parms.hChannelClass = MQE_CHANNEL_CLASS;
```

In order to set up an array we need to allocate some memory then setup the network information. This example shows using the loopback address with the listener expected to be on port 8080:

```
   OSAMEMORY_ALLOC(&error, (MQEVOID**) &parms.phAdapterParms,
            (sizeof(MQEHANDLE) * 2), "comms test");
   rc = OSAMQESTRING_NEW( &error, "127.0.0.1", SB_STR,
         &parms.phAdapterParms[0]);
   rc = OSAMQESTRING_NEW( &error, "8080", SB_STR,
         &parms.phAdapterParms[1]);
```

We now set the number of element in the array:

```
parms.destParmLen = 2;
```

And last of all set the flags to tell the receiving administration function what information it should look for in the structure:

```
parms.opFlags = CONNDEF_ADAPTER_CLASS_OP | CONNDEF_ADAPTER_PARMS_OP |
                CONNDEF_CHANNEL_CLASS_OP;
```

Now, having set everything up we can call the administration function in order to create our connection definition. Note, it is wise to check the return code in order to determine whether the call has been successful

```
rc = mqeAdministrator_Connection_create( hAdministrator, &error,
                h hQueueMgrName, &parms);
if (MQERETURN_OK == rc) {
          fprintf(pOutput,"connection definition to ServerQM
          at 127.0.0.1:8081 successfully added\n");
      }
```

The above creates a direct connection definition, if we want to create a via connection definition we would need to set the parameter structure to the default values and the name of the remote queue manager as usual:

```
MQeConnectionDefinitionParms parms = CONNDEF_INIT_VAL;
rc = OSAMQESTRING_NEW(&error, "ServerQM", SB_STR, &hQueueMgrName);
```

```
We now need to set the name of the queue manager that will then route
the messages on to the remote queue manager.

rc = OSAMQESTRING_NEW(&error,
        "RoutingQM",
                        SB_STR,
        &parms.hViaQMName);
```

Now all that is left to do is correctly set the flags that tells the administration function what to look for in the structure:

```
parms.opFlags  = CONNDEF_VIAQM_OP;
```

We then call the function as with the direct connection definition:

```
rc = mqeAdministrator_Connection_create(  hAdministrator,
               &error,
                                          hQueueMgrName,
               &parms);
```

Delete a connection definition

A connection may be deleted as follows. If the connection doesn't exist then the return code of MQERETURN_COMMS_MANAGER_WARNING will be given with the reason code of MQEREASON_CONDEF_DOES_NOT_EXIST.

```
rc = mqeAdministrator_Connection_delete(hAdministrator,
          &error, hQueueMgrName);
```

Update a connection definition

As has been previously stated it is strong recommended you ensure a connection is not being used when a connection definition is updated. The flags are used to determine which parts of the information in the connection definition are to be updated. So, even if a value is provided in the structure, if the correct flag is not set that value will not be used:

```
MQeConnectionDefinitionParms parms = CONNDEF_INIT_VAL;
```

We will create a new description:

```
rc = OSAMQESTRING_NEW(&error, "replacement description", SB_STR,
        &parms.hDescription);
```

If we set the opFlags field as follows the description will not be updated, instead the administration function will attempt to update the value for the name of the via queue manager:

```
parms.opFlags = CONNDEF_VIAQM_OP;
```

We need to set the opFlags field as follows in order to obtain the desired behavior:

```
Parms.opFlags = CONNDEF_DESC_OP;
```

The function to update the connection definition is then called as follows:

```
rc = mqeAdministration_Connection_update(hAdministrator , &error,
               hQueueMgrName, &parms);
```

General comment

As can be seen from the example there is much repetitive code involved in creating then checking the reply for an administration message. It is therefore probably desirable to put this code into a common class that may be used by all classes creating and checking the replies of administration messages.

The full code for updating a connection definition and for deleting a connection definition may be found in the examples.

Chapter 12. Listener

In order for a queue manager to receive requests from other queue managers it is necessary for an MQeListener to be instantiated and running. At present this functionality is only available in Java.

A listener uses a communications adapter to listen at a named location, in an IP network this is a named port. For a client to make a successful connection, the network address of the listening queue manager, the named location, and the communications adapter class must be made known to the client. An error in any one of these in the connection definition on the client will result in an error when they try to connect.

Creating a listener

In order to create a listener is it necessary to use an administration message. The following is based upon the example example.config.ConfigListener, the administration message is instantiated as follows:

```
MQeCommunicationsListenerAdminMsg createMessage =
                new MQeCommunicationsListenerAdminMsg();
```

We now need to provide a name for the listener:

```
createMessage.setName("Listener1");
```

The name of the queue manager to which the administration message is intended is also required:

```
createMessage.setTargetQMgr(queueManagerName);
```

The next thing we need to do is set the action for the administration message as well as providing the information the listener requires in order to function.

```
createMessage.create(com.ibm.mqe.adapters.MQeTcpipHistoryAdapter,
        8087, 36000000, 10);
```

The first parameter provides the name of the communications adapter we wish to use, in this instance we have stipulated the MQeTcpipHistoryAdapter, an alias may be used instead. The type of communications adapter being used by the listener needs to be made known to clients wishing to connect to the queue manager using the listener.

The second parameter defines the named location the listener uses, in this instance an IP port number of 8087, again the clients will need to be aware of this in order to contact this listener.

The third parameter specifies the channel timeout value. This value is used to determine when an incoming channel should be closed. WebSphere MQ Everyplace polls the channels, if a channel has been idle for longer than the timeout value it will be closed.

The last parameter determines the maximum number of channels the listener will have running at any one time. If a client tries to connect once this value has been reached the connection is refused.

Having set the correct action and provided the relevant information we can set the message type, in this instance we are using a request message style which indicates we would like a reply to indicate success or failure. However, it might make no difference if a description is altered successfully or not. In this case, use a message style of datagram which indicates no reply is required.

```
createMessage.putInt(MQe.Msg_Style, MQe.Msg_Style_Request);
```

When requesting a reply, provide the queue and owning queue manager name to which the reply must be sent. This example uses the default administration reply queue.

```
createMessage.putAscii(MQe.Msg_ReplyToQ, MQe.Admin_Reply_Queue_Name);
createMessage.putAscii(MQe.Msg_ReplyToQMgr, queueManagerName);
```

To get the correct reply message that corresponds to our administration message, use a correlation ID. This is copied from the administration message into the reply so we can get the correct message. To generate an id that is relatively safe as being unique, use the system time:

```
String match = "Msg" + System.currentTimeMillis();
createMessage.putArrayOfByte(MQe.Msg_CorrelID, match.getBytes());
```

We are now in a position to put the administration message to the administration queue of the target queue manager. The last two parameters provide the ability to use an attribute and an id to allow the undo method to be called, neither of which we shall worry about at this juncture.

```
queueManager.putMessage(queueManagerName, MQe.Admin_Queue_Name,
         createMessage, null, 0);
```

Having put the message to the queue we shall now wait for a reply. As can be seen we use the correlation identifier we used to put the message in order to get the reply and there is a useful method that provides us with the reason code to indicate success or failure.

```
MQeFields filter = new MQeFields();
filter.putArrayOfByte(MQe.Msg_CorrelID, match.getBytes());

// now wait for a reply
MQeAdminMsg response =
     (MQeAdminMsg) queueManager.waitForMessage(
        queueManagerName,
        MQe.Admin_Reply_Queue_Name,
        filter,
       null,
       0,
      3000);
 // the administration message has a method that
```

```
 //will get out the return code
switch (response.getRC()) {
     case MQeAdminMsg.RC_Success :
     break;
```

Having successfully created our listener we need to start it, the listener is only automatically started on the next restart of the queue manager. Again an administration message is required to start or stop a listener, we can use the approach taken above, using the following methods in the MQeCommunicationsListenerAdminMsg class. To start the listener:

```
MQeCommunicationsListenerAdminMsg startMessage =
                new MQeCommunicationsListenerAdminMsg();

⋮

 startMessage.start();
```

To stop the listener:

```
MQeCommunicationsListenerAdminMsg startMessage =
                new MQeCommunicationsListenerAdminMsg();

⋮

startMessage.stop();
```

In order to delete a listener we need to set the action of the administration message to delete as follows:

```
deleteMessage.setAction(MQeAdminMsg.Action_Delete);
```

If you try to delete a listener that is running you will receive an exception, so make sure your listener has successfully stopped before trying to delete it.

Chapter 13. Administering bridge resources

This chapter describes how WebSphere MQ Everyplace interacts with other messaging software, under the following headings:

- The WebSphere MQ bridge
- What makes a queue manager bridge-enabled
- Finding out if a queue manager is bridge-enabled
- Classes required to make a queue manager bridge-enabled
- Configuring the WebSphere MQ bridge

The WebSphere MQ bridge

To exchange messages with a WebSphere MQ queue manager, a solution needs to use a piece of the WebSphere MQ Everyplace toolkit called the *bridge*. The WebSphere MQ bridge consists of a number of classes, which must be available on the CLASSPATH, for the Java Virtual Machine to use. These are described in more detail later in this chapter. The bridge is a server-side component, in that it is deployed at the server-end of client-server network topologies. WebSphere MQ bridge queue managers with a device or client role need not have a bridge themselves if they can connect to a *bridge-enabled* queue manager. In such cases, message traffic passed from these *leaf node* queue managers can be routed via the bridge-enabled WebSphere MQ bridge queue manager, which in turn can use bridge functionality to convey the message to a WebSphere MQ queue manager.

Normally, the bridge-enabled WebSphere MQ Everyplace queue manager is deployed within a DMZ or behind the firewall, where the network connections between it and the WebSphere MQ queue managers it talks to are either on the same machine, or on a machine which is contactable with a reliable high-bandwidth LAN network. A bridge-enabled WebSphere MQ Everyplace queue manager is often referred to as a *gateway* queue manager, because it provides a gateway between the WebSphere MQ Everyplace and WebSphere MQ messaging networks.

The bridge function is available only in the Java part of the WebSphere MQ Everyplace toolkit, and is usable only by WebSphere MQ Everyplace queue managers running within a Java Virtual Machine. The bridge resources can be manipulated from a native platform with the aid of WebSphere MQ Everyplace administration messages. A specialized set of administration messages is provided in the WebSphere MQ Everyplace product for this purpose, and is described later in this chapter.

Messages from a WebSphere MQ application destined for WebSphere MQ Everyplace are addressed to the WebSphere MQ Everyplace queue manager and queue as normal. Standard WebSphere MQ routing, using remote queue and remote queue manager definitions, is used to route messages to the WebSphere MQ Everyplace queue managers. WebSphere MQ channels are not defined for transmission queues. Instead, the WebSphere MQ Everyplace gateway pulls the messages off these queues and ensures their delivery to the WebSphere MQ Everyplace destination. This is explained in more detail below.

The WebSphere MQ bridge handles the transfer of messages between the two systems, including translation between different message formats. Configuring the WebSphere MQ bridge provides a detailed description of this interface.

What makes a queue manager bridge-enabled

Some WebSphere MQ Everyplace queue managers are capable of exchanging messages with WebSphere MQ, and some are not. Those which can are said to be bridge-enabled or bridge-capable. Put simply, a bridge-enabled queue manager is one which runs in an environment capable of supporting theWebSphere MQ Java classes, and when the WebSphere MQ bridge software is available for the JVM to load.

When a WebSphere MQ Everyplace queue manager is activated, it attempts to load the WebSphere MQ bridge software component. If the WebSphere MQ Everyplace classes and dependent software are all loadable, then the queue manager can later report that it is bridge-capable. If required Java classes are not loadable, then error information is traced at that point, but the queue manager will continue to activate, resulting in a queue manager which reports that it is not bridge-capable.

Finding out if a queue manager is bridge-enabled

If you apply an inquireAll operation to a queue manager, a *bridge-capable* property is returned. This field is boolean. A true value indicates that the classes required to support the bridge function are present on the class path. A false value indicates that required classes are missing from the class path.

If the queue manager is reporting that it is bridge-capable, bridge resources can be configured and manipulated on that queue manager. If the queue manager reports that it is not bridge-capable, any attempt to administer bridge resources will fail. Such situations are often indicative that the required WebSphere MQ Java classes, or parts of the WebSphere MQ bridge software are not available on the classpath.

Changing the classpath to reference the WebSphere MQ Java and WebSphere MQ bridge classes, and restarting the JVM in which the WebSphere MQ Everyplace queue manager is running should result in the queue manager reporting that it is bridge-capable. The code in `examples.mbridge.administration.commandline.IsQueueManagerBridgeCapable` provides an example of how to code this query.

Classes required to make a queue manager bridge-enabled

To use the WebSphere MQ bridge you must have:

- WebSphere MQ Classes for Java version 5.1 or later, installed on your WebSphere MQ Everyplace system, and available on the classpath for JVMs to use. WebSphere MQ Classes for Java is available for free download from the Web as SupportPac™ MA88. This can be downloaded for free from:
  http://www.ibm.com/software/mqseries/txppacs.
  The WebSphere MQ classes for Java are also shipped with WebSphere MQ software, though may not be installed depending on the options selected when

WebSphere MQ was installed. An example script below demonstrates what might be needed to set the correct environment on a Windows system. This example was taken from the Java\Demo\Windows folder. A similar bsh UNIX example can be found in Java\Demo\Unix directory.

```
@Rem  Set up the name of the MQ Series directory.
@Rem  This should be modified to suit your installation.
set MQDIR=C:\Program Files\IBM\MQSeries

@Rem   If you wish to use the WebSphere MQ bridge then the CLASSPATH also
@Rem   needs to know how to get to the MQSeries Java Client.
if Exist "%MQDIR%\java\lib"^
       set CLASSPATH=%CLASSPATH%;%MQDIR%\java\lib;
if Exist "%MQDIR%\java\lib\com.ibm.mq.jar"^
         set CLASSPATH=%CLASSPATH%; %MQDIR%\java\lib\com.ibm.mq.jar
if Exist "%MQDIR%\java\lib\com.ibm.mqbind.jar"^
       set CLASSPATH=%CLASSPATH%;%MQDIR%\java\lib\com.ibm.mqbind.jar
if Exist "%MQDIR%\java\lib\com.ibm.mq.iiop.jar"^
       set CLASSPATH=%CLASSPATH%;%MQDIR%\java\lib\com.ibm.mq.iiop.jar
if Exist "%MQDIR%\java\lib\jta.jar"^
       set CLASSPATH=%CLASSPATH%;%MQDIR%\java\lib\jta.jar
if Exist "%MQDIR%\java\lib\jndi.jar"^
       set CLASSPATH=%CLASSPATH%;%MQDIR%\java\lib\jndi.jar
if Exist "%MQDIR%\java\lib\jms.jar"^
              set CLASSPATH=%CLASSPATH%;%MQDIR%\java\lib\jms.jar
if Exist "%MQDIR%\java\lib\com.ibm.mqjms.jar"^
       set CLASSPATH=%CLASSPATH%;%MQDIR%\java\lib\com.ibm.mqjms.jar
if Exist "%MQDIR%\java\lib\connector.jar"^
         set CLASSPATH=%CLASSPATH%;%MQDIR%\java\lib\connector.jar
if Exist "%MQDIR%\java\lib\fscontext.jar"^
           set CLASSPATH=%CLASSPATH%;%MQDIR%\java\lib\fscontext.jar
if Exist "%MQDIR%\java\lib\ldap.jar"^
              set CLASSPATH=%CLASSPATH%;%MQDIR%\java\lib\ldap.jar

@Rem  The MQSeries Bridge also requires access to the MQSeries
@Rem   Executables so that native DLLs can be found.
if Exist "%MQDIR%\java\lib" set PATH=%PATH%;%MQDIR%\java\lib
if Exist "%MQDIR%\bin"      set PATH=%PATH%;%MQDIR%\bin;
```

- WebSphere MQ Everyplace classes, of which an example of superset classes can be found in the Java/Jars/MQeGateway.jar file. Deploying this file and adding it to your classpath will provide the queue manager with all the required classes necessary to use bridge function. For example,

  ```
  set CLASSPATH=%CLASSPATH%;%MQeDIR%\Java\Jars\MQeGateway.jar
  ```

Configuring the WebSphere MQ bridge

The configuration of the WebSphere MQ bridge requires you to perform some actions on the WebSphere MQ queue manager, and some on the WebSphere MQ Everyplace queue manager. The bridge can be divided into two pieces:

- Configuration of resources required to route a message from WebSphere MQ Everyplace to WebSphere MQ

- Configuration of resources required to route a message from WebSphere MQ to WebSphere MQ Everyplace

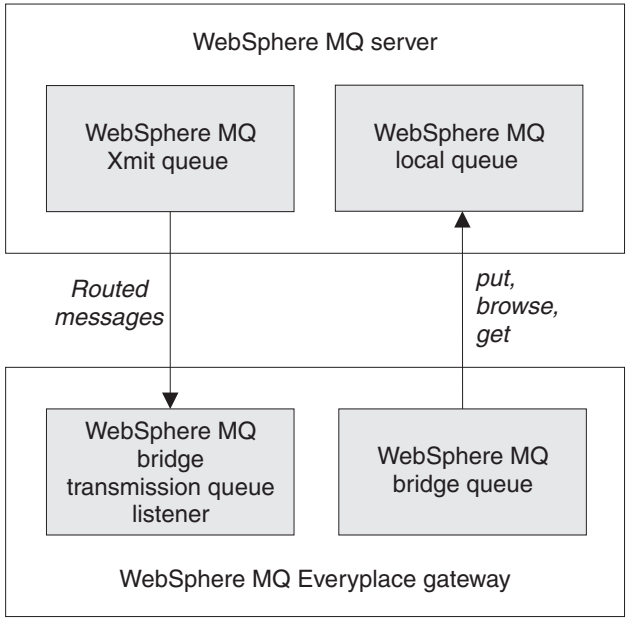

Figure 46.

Configuration of both types of routes are discussed in the following sections.

The bridge objects are defined in a hierarchy as shown in Figure 47 on page 134

The following rules govern the relationship between the various objects:

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

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

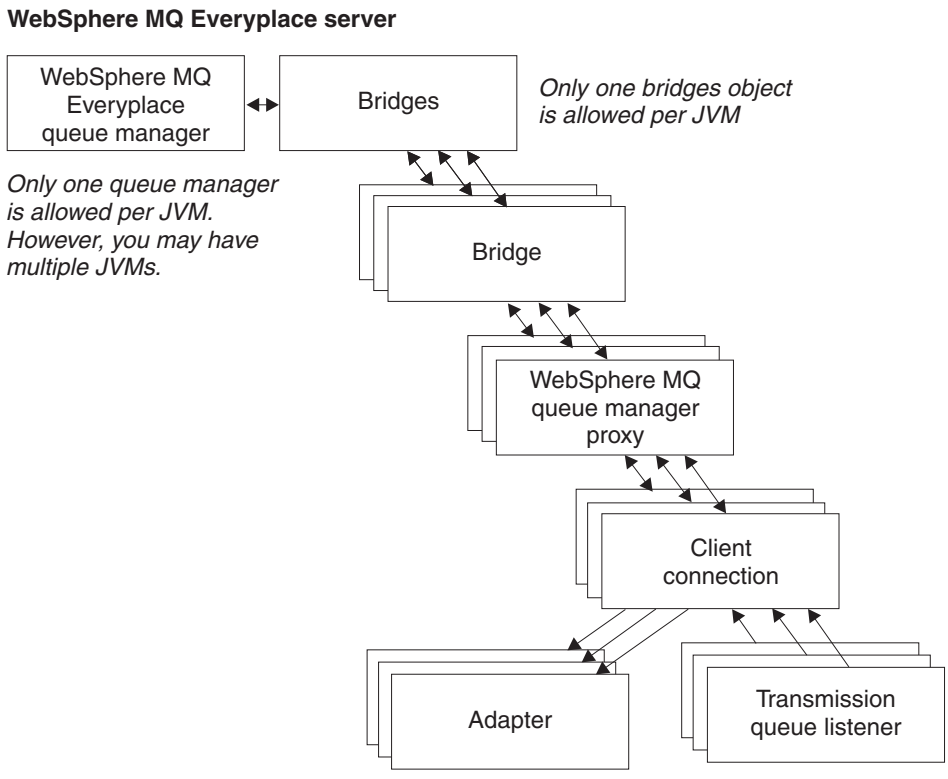

Figure 47. Bridge object hierarchy

The bridge configuration option allows a WebSphere MQ Everyplace queue managar to communicate with WebSphere MQ host and distributed queue managers through client channels. The bridge component manages a pool of client channels that can be attached to one or more host or distributed queue managers. You can configure multiple bridge-enabled WebSphere MQ Everyplace queue managers in a single network.

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

When moving messages in the other direction, from WebSphere MQ Everyplace to WebSphere MQ, the gateway queue manager configures one or more *bridge queues*. Each bridge queue can connect to any queue manager directly and send its messages to the target queue. In this way a gateway can dispatch WebSphere MQ Everyplace

messages routed through a single WebSphere MQ Everyplace queue manager to any WebSphere MQ queue manager, either directly or indirectly.

The bridges resource

The bridges resource is responsible for maintaining a list of bridge resources. It provides a single-resource which can be started and stopped, where starting and stopping a bridges resource can start and stop all the resources beneath it in the resource hierarchy. It is ″owned″ by the WebSphere MQ Everyplace queue manager. If the WebSphere MQ Everyplace queue manager is bridge-enabled, then a bridges resource is automatically created, and present. This resource has no persistent information associated with it. It has the following properties:

Table 19. Bridges properties

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

The bridges, and the other bridge resources can be started and stopped independently of the WebSphere MQ Everyplace queue manager. If such a bridge resource is started (or stopped) the action also applies to all of its children, that is all bridges, queue manager proxies, client connections, and transmission queue listeners.

More detail of these properties can be found in the WebSphere MQ Everyplace Java Programming Reference in the administration class `com.ibm.mqe.mqbridge.MQeMQBridgesAdminMsg`. The bridges resource supports the Inquire and InquireAll, start, and stop operations. Create, delete, and update are not appropriate actions to use with this resource. Examples of how to inquire, start, and stop a bridges resource can be found in the java class `examples.mqbridge.administration.programming.AdminHelperBridges`

The bridge resource

The bridge resource is responsible for holding a number of persistent property values, and a list of WebSphere MQ queue manager proxy resources. If started or stopped, it can act as a single point of control to start and stop all the resources beneath it in the bridge hierarchy. Each bridge object supports the full range of create, inquire, inquire-all, update, start, stop, and delete operations. Examples of these operations can be found in the java class `examples.mqbridge.administration.programming.AdminHelperBridge`. The bridge resource has the following properties:

Table 20. Bridge properties

Property	Explanation
Class	Bridge class
Default transformer	The default class, rule class, to be used to transform a message from WebSphere MQ Everyplace to WebSphere MQ, or vice versa, if no other transformer class has been associated with the destination queue

Table 20. Bridge properties (continued)

Property	Explanation
Heartbeat interval	The basic timing unit to be used for performing actions against bridges
Name	Name of the bridge
Run state	Status: running or stopped
Startup rule class	Rule class used when the bridge is started
WebSphere MQ Queue Manager Proxy Children	List of all Queue Manager Proxies that are owned by this bridge

More detail of each property can be found in the WebSphere MQ Everyplace Java Programming Reference, in the administration class `com.ibm.mqe.mqbridge.MQeMQBridgeAdminMsg`.

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

The WebSphere MQ queue manager proxy

The WebSphere MQ queue manager proxy holds the properties specific to a single WebSphere MQ queue manager. The proxy properties are shown in the following table:

Table 21. WebSphere MQ queue manager proxy properties

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

More detail of each property can be found in the WebSphere MQ Everyplace Java Programming Reference, in the administration class `com.ibm.mqe.mqbridge.MQeMQQMgrProxyAdminMsg`.

Each proxy object supports the full range of create,inquire, inquire-all, update, start, stop, delete operations. Examples of these operations can be found in the java class `examples.mqbridge.administration.programming.AdminHelperMQQMgrProxy`.

The client connection resource

The client connection definition holds the detailed information required to make a connection to a WebSphere MQ queue manager. The connection properties are shown in the following table:

Table 22. Client connection service properties

Property	Explanation
Adapter class	Class to be used as the gateway adapter
CCSID*	The integer WebSphere MQ CCSID value to be used
Class	Bridge client connection service class
Max connection idle time	The maximum time a connection is allowed to be idle before being terminated
WebSphere MQ password*	Password for use by the Java client
WebSphere MQ port*	IP port number used to create connections to the WebSphere MQ queue manager via the Java client classes. If not specified then the WebSphere MQ queue manager is assumed to be on the same machine as the bridge and the Java bindings are used
WebSphere MQ receive exit class*	Used to match the receive exit used at the other end of the client channel; the exit has an associated string to allow data to be passed to the exit code
WebSphere MQ security exit class*	Used to match the security exit used at the other end of the client channel; the exit has an associated string to allow data to be passed to the exit code
WebSphere MQ send exit class*	Used to match the send exit used at the other end of the client channel; the exit has an associated string to allow data to be passed to the exit code
WebSphere MQ user ID*	user ID for use by the Java client
Client connection service name	Name of the server connection channel on the WebSphere MQ machine
Name of owning queue manager proxy	The name of the owning queue manager proxy
Startup rule class	Rule class used when the bridge client connection service is started
Sync queue name	The name of the WebSphere MQ queue that is used by the bridge for synchronization purposes
Sync queue purger rules class	The rules class to be used when a message is found on the synchronous queue
Run state	Status: running or stopped
Name of owning Bridge	The name of the bridge that owns this client connection
MQ XmitQ Listener Children	List of all the listeners that use this client connection

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

The maximum connection idle time is used to control the pool of Java client connections maintained by the bridge client connection service to its WebSphere MQ system. When a WebSphere MQ connection becomes idle, through lack of use, a timer is started and the idle connection is discarded if the timer expires before the connection is reused. Creation of WebSphere MQ connections is an expensive operation and this process ensures that they are efficiently reused without consuming excessive resources. A value of zero indicates that a connection pool should not be used.

More detail of each property can be found in the WebSphere MQ Everyplace Java Programming Reference, in the administration class `com.ibm.mqe.mqbridge.MQeClientConnectionAdminMsg`.

Each client connection object supports the full range of create,inquire, inquire-all, update, start, stop, delete operations. Examples of these operations can be found in the java class `examples.mqbridge.administration.programming.AdminHelperMQClientConnection`.

The transmit queue listener resource

The listener moves messages from WebSphere MQ to WebSphere MQ Everyplace.

Table 23. Listener properties

Property	Explanation
Class	Listener class
Dead letter queue name	Queue used to hold messages from WebSphere MQ to WebSphere MQ Everyplace that cannot be delivered
Listener state store adapter	Class name of the adapter used to store state information
Listener name	Name of the WebSphere MQ XMIT queue supplying messages
Owning client connection service name	Client connection service name
Run state	Status: running or stopped
Startup rule class	Rule class used when the listener is started
Transformer class	Rule class used to determine the conversion of a WebSphere MQ message to WebSphere MQ Everyplace
Undelivered message rule class	Rule class used to determine action when messages from WebSphere MQ to WebSphere MQ Everyplace cannot be delivered

Table 23. Listener properties (continued)

Property	Explanation
Seconds wait for message	An advanced option that can be used to control listener performance in exceptional circumstances

More detail of each property can be found in the WebSphere MQ Everyplace Java Programming Reference, in the administration class `com.ibm.mqe.mqbridge.MQeListenerAdminMsg`.

Each transmit queue listener object supports the full range of create,inquire, inquire-all, update, start, stop, delete operations. Examples of these operations can be found in the java class

```
// type all on one line, no spacing
examples.mqbridge.administration.programming.
   AdminHelperMQTransmitQueueListener
```

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

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

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

The bridge queue

A WebSphere MQ bridge queue definition can contain the following attributes.

Table 24. WebSphere MQ bridge queue properties

Property	Explanation
Alias names	Alternative names for the queue
Authenticator	Must be null
Class	Object class
Client connection	Name of the client connection service to be used
Compressor	Must be null
Cryptor	Must be null
Expiry	Passed to transformer
Maximum message size	Passed to the rules class
Mode	Must be synchronous

Table 24. WebSphere MQ bridge queue properties (continued)

Property	Explanation
MQ queue manager proxy	Name of the WebSphere MQ queue manager to which the message should first be sent
WebSphere MQ bridge	Name of the bridge to convey the message to WebSphere MQ
Name	Name by which the remote WebSphere MQ queue is known to WebSphere MQ Everyplace
Owning queue manager	Queue manager owning the definition
Priority	Priority to be used for messages, unless overridden by a message value
Remote WebSphere MQ queue name	Name of the remote WebSphere MQ queue
Rule	Rule class used for queue operations
Queue manager target	WebSphere MQ queue manager owning the queue
Transformer	Name of the transformer class that converts the message from WebSphere MQ Everyplace format to WebSphere MQ format
Type	WebSphere MQ bridge queue

More detail of each property can be found in the WebSphere MQ Everyplace Java Programming Reference, in the administration class `com.ibm.mqe.mqbridge.MQeMQBridgeQueueAdminMsg`.

Example code which manipulates a bridge queue can be found in the java class `examples.mqbridge.administration.programmingAdminHelperBridgeQueue`.

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

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

Applications make use of WebSphere MQ bridge queues like any other WebSphere MQ Everyplace remote queue, using the `putMessage`, `browseMessages`, and `getMessage` methods of the MQeQueueManager class. The queue name parameter in these calls is the name of the WebSphere MQ bridge queue, and the queue manager name parameter is the name of the WebSphere MQ queue manager. However, in order for this queue manager name to be accepted by the local WebSphere MQ Everyplace

server, a connection definition with this WebSphere MQ queue manager name must exist with null for all the parameters, including the channel name.

Note: there are some restrictions on the use of getMessage and browseMessages with WebSphere MQ bridge queues. It is not possible to get or browse messages from WebSphere MQ bridge queues that point to WebSphere MQ remote queue definitions. Nor is it possible to use nonzero Confirm IDs on WebSphere MQ bridge queue gets. This means that the getMessage operation on WebSphere MQ bridge queues does not provide assured delivery. If you need a get operation to be assured, you should use transmission-queue listeners to transfer messages from WebSphere MQ.

Administration of the WebSphere MQ bridge is handled in the same way as the administration of a normal WebSphere MQ Everyplace queue manager, through the use of administration messages. New classes of messages are defined as appropriate to the queue.

Naming recommendations for interoperability with a WebSphere MQ network

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

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

  We strongly recommend that you do not use this character in the name of any WebSphere MQ queue or queue manager.
- WebSphere MQ queue and queue manager names have a limit of 48 characters but WebSphere MQ Everyplace names have no length restrictions.

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

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

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

Configuring a basic installation

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

1. Make sure you have a WebSphere MQ system installed and that you understand local routing conventions, and how to configure the system.
2. Install WebSphere MQ Everyplace on a system (It can be the same system as your WebSphere MQ system is located on if you wish)
3. If WebSphere MQ Classes for Java is not already installed, download it from the Web and install it.
4. Set up your WebSphere MQ Everyplace system and verify that it is working properly before you try to connect it to WebSphere MQ.
5. Update the MQe_java\Classes\JavaEnv.bat file so that it points to the Java classes that are part of the WebSphere MQ Classes for Java, and to the classpath for your JRE (Java Runtime Environment). Ensure that the SupportPac MA88 .jar files are in the classpath, and that the java\lib and \bin directories are in your path. This is set by the *MQE_VM_OPTIONS_LOCN* which should be set to point to the vm_options.txt file during installation.
6. Plan the routing you intend to implement. You need to decide which WebSphere MQ queue managers are going to talk to which WebSphere MQ Everyplace queue managers.
7. Decide on a naming convention for WebSphere MQ Everyplace objects and WebSphere MQ objects and document it for future use.
8. Modify your WebSphere MQ Everyplace system to define a WebSphere MQ bridge on your chosen WebSphere MQ Everyplace server. See the WebSphere MQ Everyplace Java Programming Reference for information on using examples.mqbridge.awt.AwtMQBridgeServer to define a bridge.
9. Connect the chosen WebSphere MQ queue manager to the bridge on the WebSphere MQ Everyplace server as follows:
   - On the WebSphere MQ queue manager:

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

     a. Define the server connections
     b. Define a sync queue for WebSphere MQ Everyplace to use to provide assured delivery to the WebSphere MQ system. You need one of these for each server connection that the WebSphere MQ Everyplace system can use.
   - On the WebSphere MQ Everyplace server:
     a. Define a WebSphere MQ queue manager proxy object which holds information about the WebSphere MQ queue manager. This involves the following steps:
        1) Collect the Hostname of the WebSphere MQ queue manager.
        2) Put the name in the WebSphere MQ queue manager proxy object.
     b. Define a Client Connection object that holds information about how to use the WebSphere MQ Classes for Java to connect to the server connection on the WebSphere MQ system. This involves the following steps:
        1) Collect the Port number, and all other server connection parameters.

2) Use these values to define the client connection object so that they match the definition on the WebSphere MQ queue manager.

10. Modify your configuration on both WebSphere MQ Everyplace and WebSphere MQ to allow messages to pass from WebSphere MQ to WebSphere MQ Everyplace.
    a. Decide on the number of routes from WebSphere MQ to your WebSphere MQ Everyplace network. The number of routes you need depends on the amount of message traffic (load) you use across each route. If your message load is high you may wish to split your traffic into multiple routes.
    b. Define your routes as follows:
        1) For each route define a transmission queue on your WebSphere MQ system. DO NOT define a connection for these transmission queues.
        2) For each route create a matching transmission queue listener on your WebSphere MQ Everyplace system.
        3) Define a number of remote queue definitions, (such as remote queue manager aliases and queue aliases) to allow WebSphere MQ messages to be routed onto the various WebSphere MQ Everyplace-bound transmission queues that you defined in step b. 1.
11. Modify your configuration on WebSphere MQ Everyplace to allow messages to pass from WebSphere MQ Everyplace to WebSphere MQ:
    a. Publish details about all the queue managers on your WebSphere MQ network you want to send messages to from the WebSphere MQ Everyplace network. Each WebSphere MQ queue manager requires a connections definition on your WebSphere MQ Everyplace server. All fields except the Queue manager name should be null, to indicate that the normal WebSphere MQ Everyplace communications connections should not be used to talk to this queue manager.
    b. Publish details about all the queues on your WebSphere MQ network you want to send messages to from the WebSphere MQ Everyplace network. Each WebSphere MQ queue requires a WebSphere MQ bridge queue definition on your WebSphere MQ Everyplace server. This is the WebSphere MQ Everyplace equivalent of a `DEFINE QREMOTE` in WebSphere MQ.
        - The queue name is the name of the WebSphere MQ queue to which the bridge should send any messages arriving on this WebSphere MQ bridge queue.
        - The queue manager name is the name of the WebSphere MQ queue manager on which the queue is located.
        - The bridge name indicates which bridge should be used to send messages to the WebSphere MQ network.
        - The WebSphere MQ queue manager proxy name is the name of the WebSphere MQ queue manager proxy object, in the WebSphere MQ Everyplace configuration, that can connect to a WebSphere MQ queue manager.
        - The WebSphere MQ queue manager should have a route defined to allow messages to be posted to the `Queue Name` on `Queue Manager Name` to deliver the message to its final destination.

12. Start your WebSphere MQ and WebSphere MQ Everyplace systems to allow messages to flow. The WebSphere MQ system client channel listener must be started. All the objects you have defined on the WebSphere MQ Everyplace must be started. These objects can be started in any of the following ways:
    - Explicitly using the Administration GUI described in WebSphere MQ Everyplace Configuration Guide.
    - Configuring the rules class, as described in WebSphere MQ Everyplace System Programming Guide, to indicate the startup state (running or stopped), and restarting the WebSphere MQ Everyplace server
    - A mixture of the two previous methods

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

Configuring a bridge using WebSphere MQ Everyplace administration messages and WebSphere MQ PCF messages

PCF messages are administration messages used by WebSphere MQ queue managers. A supportpac ″MS0B: MQSeries Java classes for PCF' supportpac″ contains java code, which supplies PCF message support. This code is available as a free download from the WebSphere MQ download site at http://www.ibm.com/software/ts/mqseries/txppacs.

If you download and install it, and put the `com.ibm.mq.pcf.jar` file on your ClassPath environment variable, you have access to java classes, which can dynamically manipulate WebSphere MQ resources. When PCF messages are combined with WebSphere MQ Everyplace administration messages, complete programmatic configuration of bridge resources, and corresponding resources on a WebSphere MQ Everyplace queue manager are possible. Example code contained in the `examples.mqbridge.administration.programming.AdminHelperMQ` class, used in conjunction with the `examples.mqbridge.administration.programming.MQAgent` demonstrate how to do this. This example code has been added to the `examples.awt.AwtMQeServer program`, such that using the view **->** ″Connect local MQ default queue manager″ menu item will:

- Ensure that a bridge object exists, creating one as required.
- Query properties from the default WebSphere MQ queue manager.
- Attempt to connect that queue manager to the currently running WebSphere MQ Everyplace queue manager.
- Ensure a proxy object representing the default WebSphere MQ queue manager exists, creating one if necessary.
- Ensure a WebSphere MQ Everyplace client connection exists, and that a corresponding WebSphere MQ server connection channel exists also, creating these resources if necessary.
- Ensure a 'sync queue' exists on the WebSphere MQ queue manager.
- Ensure a transmit queue on WebSphere MQ exists, and create if necessary.
- Ensure a matching WebSphere MQ transmit queue listener exists in the configuration of the current WebSphere MQ Everyplace queue manager, creating one if necessary.
- Ensure all the bridge resources are started.
- Ensure a test queue on the WebSphere MQ queue manager exists, creating one if necessary.
- Ensure a matching WebSphere MQ Everyplace bridge queue exists, which refers to that test queue.
- Send a test MQeMQMsgObject to the test queue to make sure the configuration is working.
- Get the test MQeMQMsgObject from the test queue to make sure the configuration is working.

Configuration example

This section describes an example configuration of 4 systems.

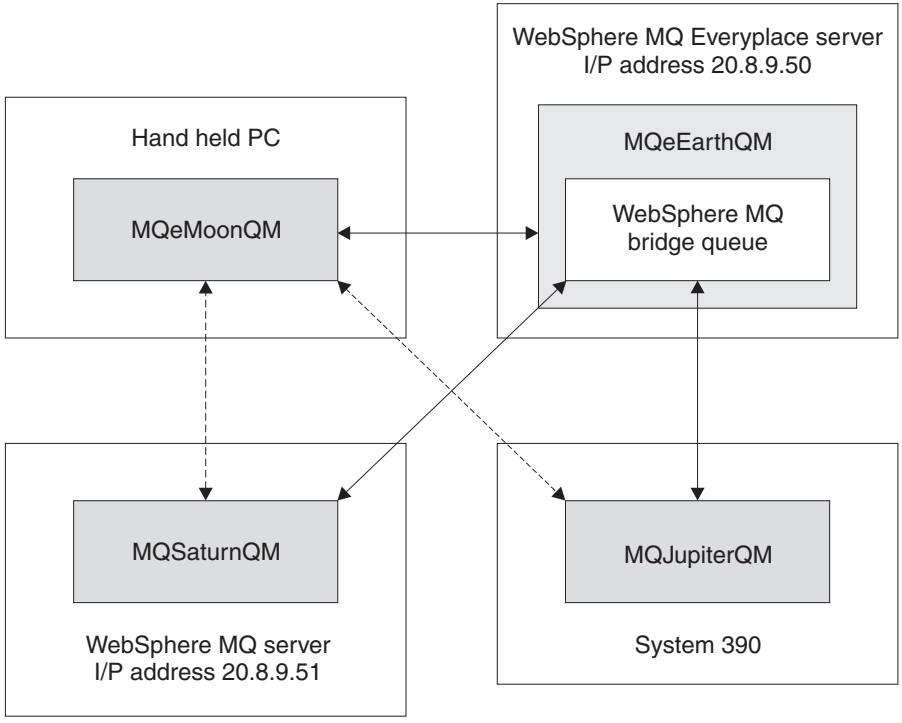

Figure 48. Configuration example

The four systems are:

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

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

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

MQJupiterQM
This is a WebSphere MQ queue manager, installed on a System/390® platform.

Requirement

The requirement for this example is that all machines are able to post a message to a queue on any of the other machines.

It is assumed that all machines are permanently connected to the network, except the MQeMoonQM machine, which is only occasionally connected.

Initial setup

For this example, it is assumed that there are local queues, to which messages can be put, on all the queue managers. These queues are called:

- `MQeMoonQ` on the `MQeMoonQM`
- `MQeEarthQ` on the `MQeEarthQM`
- `MQSaturnQ` on the `MQSaturnQM`
- `MQJupiterQ` on the `MQJupiterQM`

Enabling MQeMoonQM to put and get messages to and from the MQeEarthQM queue manager

On MQeMoonQM:

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

   Target queue manager name: `MQeEarthQM`
   Adapter: `FastNetwork:20.8.9.50`

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

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

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

   Applications on `MQeMoonQM` now have access to the `MQeMoonQ` (a local queue) in a synchronous manner, and the `MQeEarthQ` in an asynchronous manner.

Enabling the MQeEarthQM to send messages to the MQeMoonQM queue manager

Since the `MQeMoonQM` is not attached to the network for most of the time, use a store-and-forward queue on the `MQeEarthQM` to collect messages destined for the `MQeMoonQM` queue manager.

On MQeEarthQM:

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

Queue name: TO.HANDHELDS
Queue Manager Name: MQeEarthQM

2. Add a **queue manager** to the **store-and-forward queue** using the following parameters:

   Queue Name: TO.HANDHELDS
   Queue manager: MQeMoonQM

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

On MQeMoonQM:

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

   Queue Name: TO.HANDHELDS
   Queue manager name: MQeEarthQM

Any messages arriving at MQeEarthQM that are destined for MQeMoonQM are stored temporarily in the store-and-forward queue TO.HANDHELDS. When MQeMoonQM next connects to the network, it's home-server queue TO.HANDHELDS gets any messages currently on the store-and-forward queue, and delivers them to the MQeMoonQM queue manager, for storage on local queues.

Applications on MQeEarthQM can now send messages to MQeMoonQ in an asynchronous manner.

Enabling MQeEarthQM to send a message to MQSaturnQ

On MQeEarthQM:

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

   Bridge name: MQeEarthQMBridge

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

   Bridge Name: MQeEarthQMBridge
   MQ QMgr Proxy Name: MQSaturnQM
   Hostname: 20.8.9.51

3. Define a **client connection** with the following parameters:

   Bridge Name: MQeEarthQMBridge
   MQ QMgr Proxy Name: MQSaturnQM
   ClientConnectionName: MQeEarth.CHANNEL
   SyncQName: MQeEarth.SYNC.QUEUE

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

ConnectionName: `MQSaturnQM`
Channel: `null`
Adapter: `null`

5. Define an **WebSphere MQ bridge queue** with the following parameters:

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

On MQSaturnQM:

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

   Name: `MQeEarth.CHANNEL`

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

   Name: `MQeEarth.SYNC.QUEUE`

   The sync queue is needed for assured delivery.

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

Enabling MQeEarthQM to send a message to MQJupiterQ

On MQeEarthQM:

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

   ConnectionName: `MQeJupiterQM`
   Channel: `null`
   Adapter: `null`

2. Define an **WebSphere MQ bridge queue** with the following parameters:

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

On MQSaturnQM:

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

   Queue Name: `MQJupiterQ`
   Transmission Queue: `MQJupiterQM.XMITQ`

On both MQSaturnQM and MQJupiterQM:

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

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

Enabling MQeMoonQM to send a message to MQJupiterQ and MQSaturnQ

On MQeMoonQM:

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

   Target Queue manager name: MQSaturnQM
   Adapter: MQeEarthQM

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

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

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

   Queue name: MQJupiterQ
   Queue manager name: MQJupiterQM
   Access mode: Asynchronous

Applications connected to MQeMoonQM can now issue messages to MQeMoonQ, MQeEarthQ, MQSaturnQ, and MQJupiterQ, even when the handheld PC is disconnected from the network.

Enabling MQSaturnQM to send messages to the MQeEarthQ

On MQSaturnQM:

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

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

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

On MQeEarthQM:

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

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

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

Enabling MQSaturnQM to send messages to the MQeMoonQ

On MQSaturnQM:

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

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

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

The store-and-forward queue TO.HANDHELDS collects the message, and when the MQeMoonQM next connects to the network, the home-server queue retrieves the message from the store-and-forward queue, and delivers the message to the MQeMoonQ.

Enabling the MQJupiterQM to send messages to the MQeMoonQ

On MQJupiterQM:

Set up **remote queue manager aliases** for the MQeEarthQM and MQeMoonQM to route messages to MQSaturnQM using normal WebSphere MQ routing techniques.

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

Administration of the WebSphere MQ bridge

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

The example administration GUI application

An example administration GUI is provided with the WebSphere MQ bridge. It is a subclass of the examples.administration.console.Admin example described in “Example administration console” on page 21.

The subclass is called examples.mqbridge.administration.console.AdminGateway.

WebSphere MQ bridge function cannot execute on a client queue manager, so using this class in conjunction with a client queue manager does not allow the administration of bridge objects on that client queue manager, but it does enable administration of a remote WebSphere MQ bridge-enabled server queue manager.

You can administer all the resources required to configure a bridge so that it can communicate with WebSphere MQ, without programming in Java, or using the java examples supplied with WebSphere MQ Everyplace. For example, using the examples.awt.AwtMQeServer, open a queue manager and select View **– >** Admin menu item to bring up the administration GUI.

You can also use the examples.mqbridge.administration.command line tools in conjucntion with the the examples.administration.command line package, to configure the WebSphere MQ bridge. The WebSphere MQ Everyplace Java Programming Reference describes how to use these tools. Use batch files to combine a sequence of these example configuration tools, as described in “Example of use of command-line tools” on page 32.

WebSphere MQ bridge administration actions

Run state: Each administered object has a *run state*. This can be ’running’ or ’stopped’ indicating whether the administered object is active or not.

When an administered object is ’stopped’, it cannot be used, but its configuration parameters can be queried or updated.

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

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

The actions supported by the WebSphere MQ bridge administration objects are described in the following sections.

Start action: An administrator can send a **start** action to any of the administered objects.

The *affect children* boolean flag affects the results of this action. The **start** action starts the administered object and all its children (and children’s children) if the *affect children* boolean field is in the message and is set to `true`. If the flag is not in the message or is set to `false`, only the administered object receiving the start action changes its run-state. For example, sending **start** to a bridge object with *affect children* as `true` causes all proxy, client connection, and listeners that are ancestors, to start. If *affect children* is not specified, only the bridge is started. An object cannot

be started unless its parent object has already been started. Sending a start event to an administered object attempts to start all the objects higher in the hierarchy that are not already running.

Stop action: An administered object can be stopped by sending it a **stop** action. The receiving administered object always makes sure all the objects below it in the hierarchy are stopped before it is stopped itself.

Inquire action: The **inquire** action queries values from an administered object.

If the administered object is running, the values returned on the inquire are those that are currently in use. The values returned from a stopped object reflect any recent changes to values made by an **update** action. Thus, a sequence of **start**, **update**, **inquire**, returns the values configured *before* the update, while **start**, **update**, **stop**, **inquire**, returns the values configured *after* the update.

You may find it less confusing if you stop any administered object before updating variable values.

Update action: The **update** action changes one or more values for characteristics for an administered object. The values set by an **update** action do not become current until the administered object is next stopped. (See “Inquire action”.)

Delete action: The **delete** action permanently removes all current and persistent information about the administered object. The *affect children* boolean flag affects the outcome of this action. If the *affect children* flag is present and set to `true` the administered object receiving this action issues a **stop** action, and then a **delete** action to all the objects below it in the hierarchy, removing a whole piece of the hierarchy with one action. If the flag is not present, or it is set to `false`, the administered object deletes only itself, but this action cannot take place unless all the objects in the hierarchy below the current one have already been deleted.

Create action: The **create** action creates an administered object. The run state of the administered object created is initially set to `stopped`.

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

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

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

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

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

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

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

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

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

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

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

The bridge client-connection object maintains a pool of connections, that are awaiting use. If there is no bridge activity, the pool retains WebSphere MQ client channel connections until the connection idle time exceeds the idle time-out period (as specified on the client connection object configuration), at which point the channels in the pool are closed.

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

Administered objects and their characteristics

This section describes the characteristics of the different types of administered objects associated with the WebSphere MQ Everyplace WebSphere MQ bridge. Characteristics are object attributes that can be queried using an **inquireAll()** administration message. The results can be read and used by the application, or they can be sent in an update or create administration message to set the values of the characteristics. Some characteristics can also be set using the create and update administration messages. Each characteristic has a unique label associated with it and this label is used to set and get the characteristic value.

The following lists show the attributes that apply to each administered object. The label constants are defined in the header file published/MQe_MQBridge_Constants.h. If you include published/MQe_API.h in you installation, this file is included automaticallyclass com.ibm.mqe.mqbridge.MQeCharacteristicLabels.

Characteristics of bridges objects
Refer to the WebSphere MQ Everyplace Java Programming Reference for information on the `com.ibm.mqe.mqbridge.MQeMQBridgesAdminMsg`.

Characteristics of bridge objects
Refer to the WebSphere MQ Everyplace Java Programming Reference for information on the `com.ibm.mqe.mqbridge.MQeMQBridgeAdminMsg`.

Characteristics of WebSphere MQ queue manager proxy objects
Refer to the WebSphere MQ Everyplace Java Programming Reference for information on the `com.ibm.mqe.mqbridge.MQeMQQMgrProxyAdminMsg`.

Characteristics of client connection objects
Refer to the WebSphere MQ Everyplace Java Programming Reference for information on the `com.ibm.mqe.mqbridge.MQeClientConnectionAdminMsg`.

Characteristics of WebSphere MQ transmission queue listener objects
Refer to the WebSphere MQ Everyplace Java Programming Reference for information on the `com.ibm.mqe.mqbridge.MQeListenerAdminMsg`.

Handling undeliverable messages

The WebSphere MQ bridge's transmission queue listener acts in a similar way to a WebSphere MQ channel, pulling messages from a WebSphere MQ transmission queue, and delivering them to the WebSphere MQ Everyplace network. It follows the WebSphere MQ Everyplace convention in that if a message cannot be delivered, an *undelivered message rule* is consulted to determine how the transmission queue listener should react. If the rule indicates the report options in the message header, and these indicate that the message should be put onto a dead-letter queue, the message is placed on the WebSphere MQ queue, on the *sending* queue manager.

National Language Support

This section describes how the WebSphere MQ bridge handles messages flowing between MQSeries systems that use different national languages. The diagram in Figure 49 is used to describe the flow of a message from an WebSphere MQ Everyplace client application to a WebSphere MQ application.

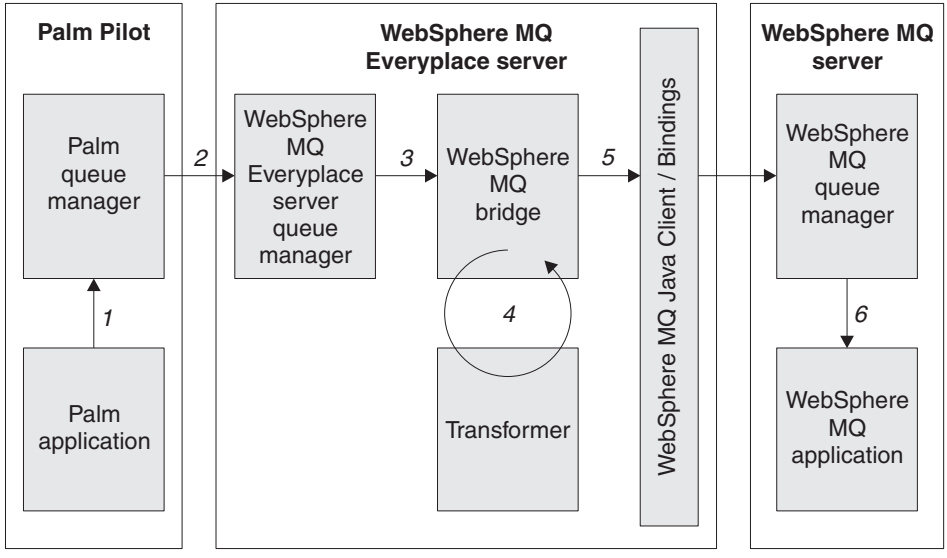

Figure 49. Message flow from WebSphere MQ Everyplace to WebSphere MQ

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

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

      A byte field
      This field should never be translated

      An ascii field
      This string has a very limited range of valid characters, conforming to the ASCII standard. The only valid characters are those that are invariant over all ASCII codepages.

   b. The message is put to the Palm queue manager. No translation is done during this put.
2. **Client queue manager puts to the server queue manager**
   The message is not translated at all through this step.
3. **WebSphere MQ Everyplace server puts the message onto the WebSphere MQ bridge queue**

The message is not translated at all through this step.

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

   Note: The examples in this section are in Java because transformers can only be written in Java. Refer to the WebSphere MQ Everyplace Application Programming Guide for more information.

   The transformer creates a WebSphere MQ message as follows:

   - The Unicode field in the WebSphere MQ Everyplace message is retrieved using:
     ```
     String value = MQemsg.GetUnicode(fieldname)
     ```
   - The retrieved value is copied to the WebSphere MQ message using **MQmsg.writeChars(value)**
   - The byte field in the WebSphere MQ Everyplace message is retrieved using:
     ```
     Byte value = MQemsg.getByte(fieldName)
     ```
   - The retrieved value is copied to the WebSphere MQ message using **MQmsg.writeByte(value)**
   - The ascii field in the WebSphere MQ Everyplace message is retrieved using either **MQmsg.writeChars(value)** to create a unicode value, or **MQmsg.writeString(value)** to create a code-set-dependent value, in the WebSphere MQ message.

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

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

   Unicode values in the WebSphere MQ message are translated from big-endian to little-endian, and vice versa, as required. Byte values in the WebSphere MQ message are translated from big-endian to little-endian, and vice versa, as required. The field that was created using **writeString()** is translated as the message is put to WebSphere MQ, using conversion routines inside the WebSphere MQ Classes for Java. ASCII data should remain ASCII data regardless of the character set conversions performed. The translations done during this step depend on the code page of the message, the CCSID of the sending WebSphere MQ Classes for Java client connection, and the CCSID of the receiving WebSphere MQ server connection.

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

   If the message contains a unicode string, the application must deal with that string as a unicode string, or else convert it into some other format (UTF8, for example). If the message contains a byte string, the application may use the bytes as it is (raw data). If the message contains a string, it is read from the message, and may be converted to a different data format as required by the application. This conversion is dependent on the codeset value in the *characterSet* header field. Java classes provide this automatically.

Conclusion

If you have an WebSphere MQ Everyplace application, and wish to convey character-related data from WebSphere MQ Everyplace to WebSphere MQ, your choice of method is determined largely by the data you wish to convey:

- **If your data contains characters in the variant ranges of the ASCII character codepages**, the character for a codepoint changes as you change between the various ASCII codepages, then use either **putUnicode**, which is never subject to translation between codepages, or **putArrayOfByte**, in which case you have to handle the translation between the sender's codepage and the receiver's codepage.

  Note: ***DO NOT USE*** **putAscii()** as the characters in the variant parts of the ASCII codepages are subject to translation.

- **If your data contains only characters in the invariant ranges of the ASCII character codepages**, then you can use **putUnicode** (which is never subject to translation between codepages) or **putAscii**, which is never subject to translation between codepages, as all your data lies within the invariant range of the ASCII codepages.

Chapter 14. Message resolution

This chapter explains, in detail, the concept of messages routes and how to use them with WebSphere MQ Everyplace.

Assumptions

It is assumed the following is understood:

- Basic Understanding of what messaging is
- Basic understanding of connection definitions and listeners
- Basic understanding of the queue types
- Basic understanding of the WebSphere MQ bridge

If you are not familiar with these concepts, refer to the WebSphere MQ Everyplace Introduction and WebSphere MQ Everyplace Application Programming Guide for more information.

Topics not covered

The following topics are covered in later chapters or other manuals:

- Configuration of connection definitions, listeners, queues, and the WebSphere MQ bridge. In this book, refer to Chapter 11, “Connection definition”, on page 115, Chapter 12, “Listener”, on page 125, Chapter 7, “Administering local queues”, on page 79, and Chapter 13, “Administering bridge resources”, on page 129.
- Synchronous versus Asynchronous Delivery. Refer to Chapter 5, Message Delivery, of the WebSphere MQ Everyplace Application Programming Guide.
- Assured Delivery. Refer to Chapter 5, Message Delivery, of the WebSphere MQ Everyplace Application Programming Guide.
- Security. Refer to Chapter 1, Security, of the WebSphere MQ Everyplace System Programming Guide.
- Rules. Refer to Chapter 3, Rules, of the WebSphere MQ Everyplace System Programming Guide.

Terminology

Queue Resolution: The process by which a queue manager chooses which queue to place a message on.

- Queue Resolution: performed by a queue manager when a message is put to it.
- Connection resolution: performed by a remote queue reference when routing a message (or request) to the real destination queue.

What you will know at the end

At the end of this chapter you will understand the following.

- How WebSphere MQ Everyplace resolves message routing
  - The steps and the rules implemented by WebSphere MQ Everyplace
  - Concepts and use of message routes
- Simple and more complex WebSphere MQ Everyplace network topologies
  - How listeners and connection definitions interact
  - Role of local queues in messaging topologies
  - Role of remote queues in messaging topologies
  - Role of store and forward queues in messaging topologies
  - Role of home server queues in messaging topologies
  - Role of queue aliases in messaging topologies
  - Role of queue manager aliases in messaging topologies
  - Role of WebSphere MQ bridge components in messaging topologies involving WebSphere MQ

Warning

Several features of WebSphere MQ Everyplace allow the routing of messages to be altered dynamically. Changing the WebSphere MQ Everyplace network topology in this fashion is not always a wise thing to do. Care must be taken to ensure that there are no 'in doubt' messages that would be affected by the change. If a message is put with a non-zero confirm id, and then the WebSphere MQ Everyplace network topology is changed to alter the routing of the subsequent confirmGetMessage call, then the unconfirmed message will not be found. WebSphere MQ Everyplace protocol treats a failure to confirm a put as an indication that the put message has been confirmed already, and therefore assumes success. This could leave an unconfirmed message on a queue, which represents a loss of a message, and therefore breaks the assured delivery promise.

Since WebSphere MQ Everyplace uses the same two step process to assure delivery of asynchronously sent messages, regardless of whether a zero or non-zero confirmId is used, changing the network topology can break the assured delivery of asynchronous message sends.

WebSphere MQ Everyplace Message Resolution

The route that a message takes through a WebSphere MQ Everyplace network can depend upon many resources (queues, connection definitions, listeners and so on). These need to be correctly set up, often in pairs whose settings need to be complementary. Failure to set up the correct resources, or setting certain of their values incorrectly can result in failure to deliver messages. Since the task of setting up a network that correctly routes messages can initially appear complex, the current chapter describes the theory underlying message resolution.

The document begins by introducing the notation used in diagramming the WebSphere MQ Everyplace network topology, and then shows message resolutions of increasingly complex nature. Along the way certain terms are surreptitiously defined to describe the process of message resolution. At the end of the document the terms are consolidated into a complete description of message resolution that encapsulates all the complexity.

A common source of confusion when discussing WebSphere MQ Everyplace is the differentiation between a local queue that exists on a remote machine (or queue manager), and a local definition of that queue on the remote machine. Both of these entities are commonly referred to as 'remote queue's. In order to disambiguate these, the term 'remote queue reference' will always be used to describe a local definition of a queue that resides on another (remote) machine (or queue manager).

Notation

This document uses a consistent notation for diagramming the resources. The diagramming technique allows the areas of specific interest to be shown prominently, while the less relevant parts of a system can be hidden. This is easier to show with a diagram. Figure 50 shows a host and the WebSphere MQ Everyplace resources on it in the familiar tree notation.

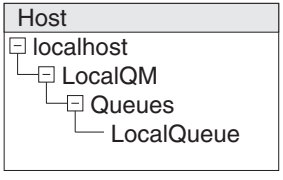

Figure 50. A host and the WebSphere MQ Everyplace resources on it.

Figure 51 shows the same resources in the 'dispersed' form.

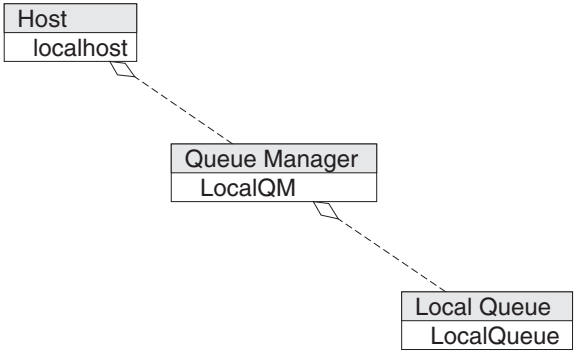

Figure 51. A host and the WebSphere MQ Everyplace resources on it: 'dispersed' form.

The line with a diamond shape shows that the queue manager is the child of the host. This preserves the parent/child relationship from the tree, that would otherwise be lost by separating the elements.

Local Queue Resolution

Local message putting is the bedrock upon which WebSphere MQ Everyplace stands. Messages, if they are to be useful, must always end up on a local queue. Message route resolution is the mechanism by which a message travels through a WebSphere MQ Everyplace network to its ultimate destination.

Figure 52 shows a simple local message put.

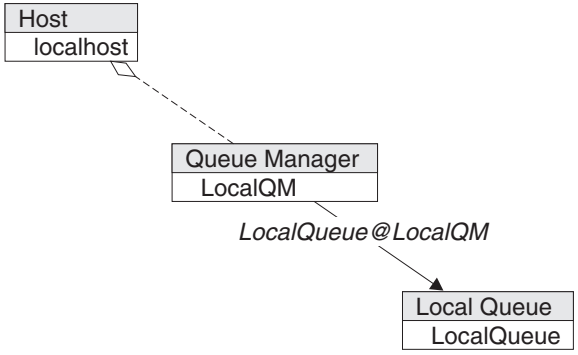

Figure 52. A simple local message put.

The message route is shown for a message put to (QueueManager)LocalQM destined for the (Queue)LocalQueue@LocalQM. This is clearly a put to a local queue, as the queue's 'queue manager name' is the same as the name of the queue manager to which the message is put.

The message route is shown with an arrow labelled with the message route name. The arrow indicates the direction in which the message flows. The text on the label indicates the currently used target name (this can change during message resolution). LocalQM looks for a queue to accept a message for LocalQueue@LocalQM. The process of determining which queue to place a message on is called Queue Resolution. LocalQM finds an exact match for the destination, the local queue. It then puts the message onto the local queue. The message will then reside on the local queue until it is retrieved via the getMessage() API call.

Local Queue Alias

Local queues can have aliases. If we add a queue alias to the local queue we provide it with another name by which it will be known. So the local queue LocalQueue@LocalQM could be given an alias of 'LocalQueueAlias', see Figure 53 on page 163.

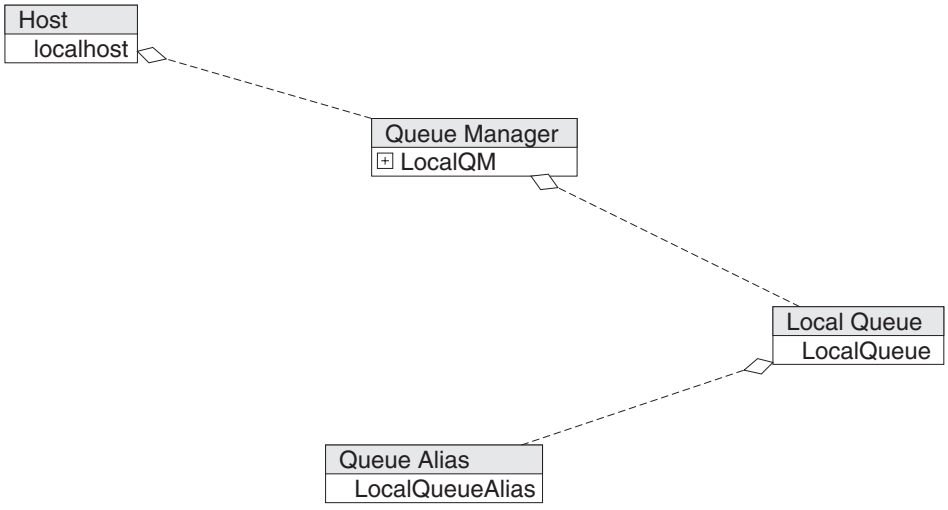

Figure 53. LocalQueue@LocalQM with an alias of 'QueueAlias'.

Messages addressed to LocalQueueAlias@LocalQM would be directed by the queue manager to LocalQueue@LocalQM. We could envisage this as the message being placed on the matching alias, almost as if the alias were a queue, and then the alias moves the message to the correct destination, see Figure 54.

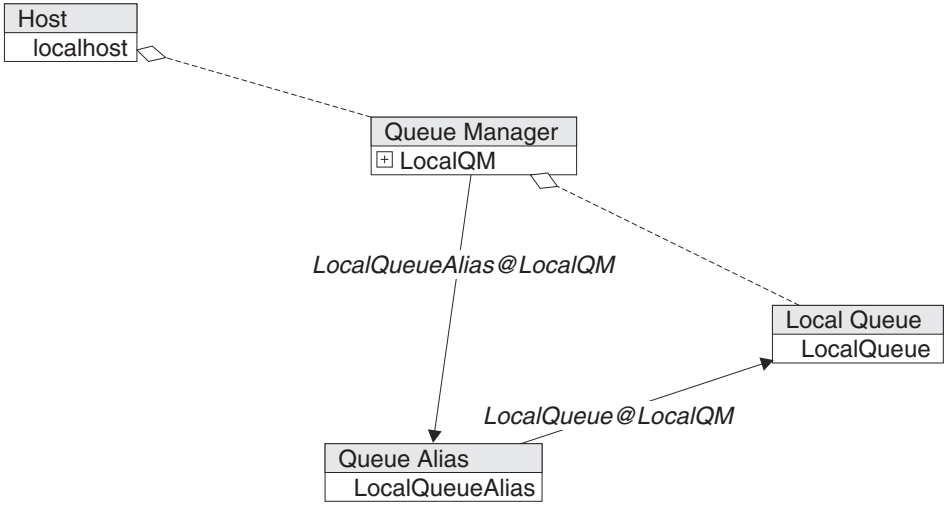

Figure 54. A message being placed on a matching alias.

The redirection of the message by the alias is accompanied by a change in the 'destination queue name' from LocalQueueAlias@LocalQM to LocalQueue@LocalQM. The fact that the message was originally put to the alias is completely lost. This can be seen by the labelling of the message route from the alias to the queue. In this particular

case the change of 'put name' is of little or no importance, but when we come to discuss some more complex message resolutions it plays a larger role.

It is important to note that the resolution of the queue alias is performed just before the message is routed to the queue. The resolution is as late as it could possibly be, and is sometimes termed 'late resolution'.

Queue Manager Alias

Queue aliases allowed us to refer to queues by more than one name. Queue Manager Aliases allow us to refer to queue managers by more than one name. We can define a Queue Manager Alias 'AliasQM' referring to the local queue manager as in Figure 55.

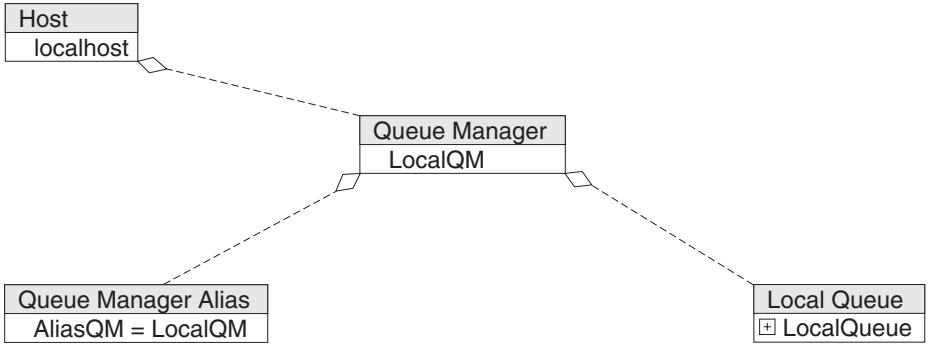

Figure 55. Defining a queue manager alias.

Messages addressed to 'AliasQM' will be routed to 'LocalQM', see Figure 56.

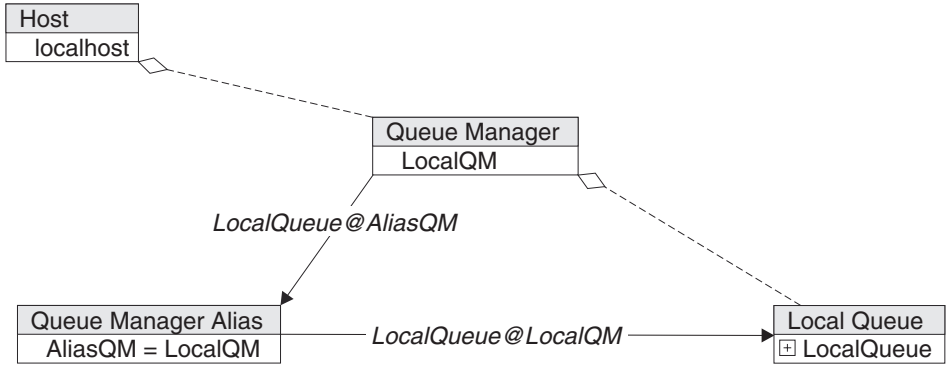

Figure 56. Addressing messages to a queue manager alias.

The redirection of the message by the alias is accompanied by a change in the 'destination queue name' from LocalQueue@AliasQM to LocalQueue@LocalQM. The fact that the message was originally put to the alias is completely lost. This can be seen

by the labelling of the message route from the alias to the queue. Queue Manager Aliases are resolved very early during message resolution; in fact this is the first step that is performed.

Queue Manager Aliases are not much use in this scenario, but become very effective as part of more complex topologies. To complete the picture we can resolve both the Queue Manager Alias and the Queue Alias, see Figure 57.

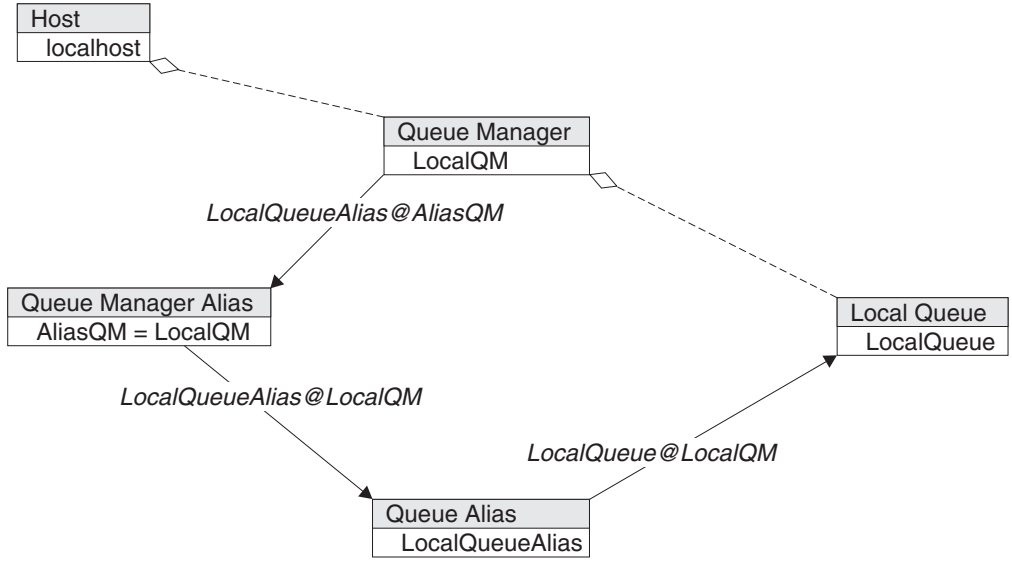

Figure 57. Resolving the queue manager alias and the queue alias.

Here we put a message to LocalQueueAlias@AliasQM, and it is resolved first via the Queue Manager Alias, and then through the Queue Alias.

Resolution of queueManager aliases happens as soon as the request reaches a queue manager. The effect is to substitute the aliased string for the aliasing string. So for the first example above, as soon as the putMessage(″AliasQM″,....) call crosses the API, it is converted to a putMessage(″LocalQM″,....) call. This resolution is also performed when a message is put to a remote queue manager. On a remote queue manager the queue aliases on that queue manager are used, not those on the originating queue manager.

An alias can point to another alias. However, circular definitions have indeterminate results. An alias can also be made of the local queue manager name. This may not seem immediately useful, but it has a well defined purpose - it allows a queue manager to behave as if it were another queue manager. This pretence means that we can remove a queue manager entirely from the network, and by creating suitable queue manager aliases elsewhere we can allocate its workload to another queue manager. This feature is useful when modifying WebSphere MQ Everyplace network topologies,

as servers, under the control of system administrators, can be moved, removed or renamed without breaking the connectivity of clients, which may not be so readily accessible.

Remote Queue Resolution

Remote queue resolution involves connection definitions and network resolution. We require a setup where there are two queue managers, one of which is the local queue manager that we use to put the message, and the other is the queue manager to which we want the message to go. Furthermore, we require that the remote queue manager has a listener, and that the local queue manager has a connection definition describing the listener, see Figure 58.

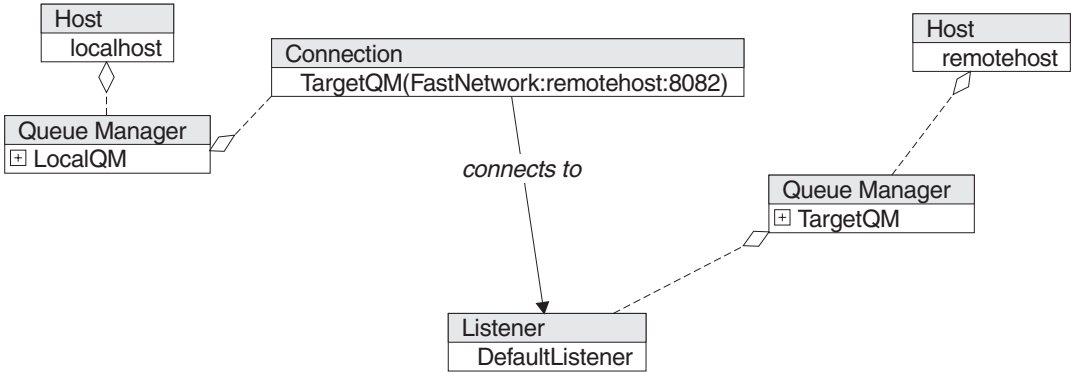

Figure 58. Local and remote queue managers with a definition and listener pair.

The connection definition/listener pair allows WebSphere MQ Everyplace to establish the network communications necessary to flow the message. The connection definition contains information about communicating with a single queue manager. The connection definition is named for the queue manager to which it defines a route. So in this example the connection definition is called TargetQM, and contains the information necessary to establish connection with (QueueManager)TargetQM. This information includes the address of the machine upon which the queue manager resides (remote host in this example), the port upon which the queue manager is listening (8081 in this example), and the protocol to use when conversing with the queue manager (FastNetwork in this example).

We need a remote queue reference on LocalQM representing the destination queue TargetQueue which resides on TargetQM. There are therefore two entities called TargetQueue@TargetQM. One is the 'real' queue, that is a local queue, and one is a reference to the real queue, a remote queue reference. Refer to Figure 59 on page 167.

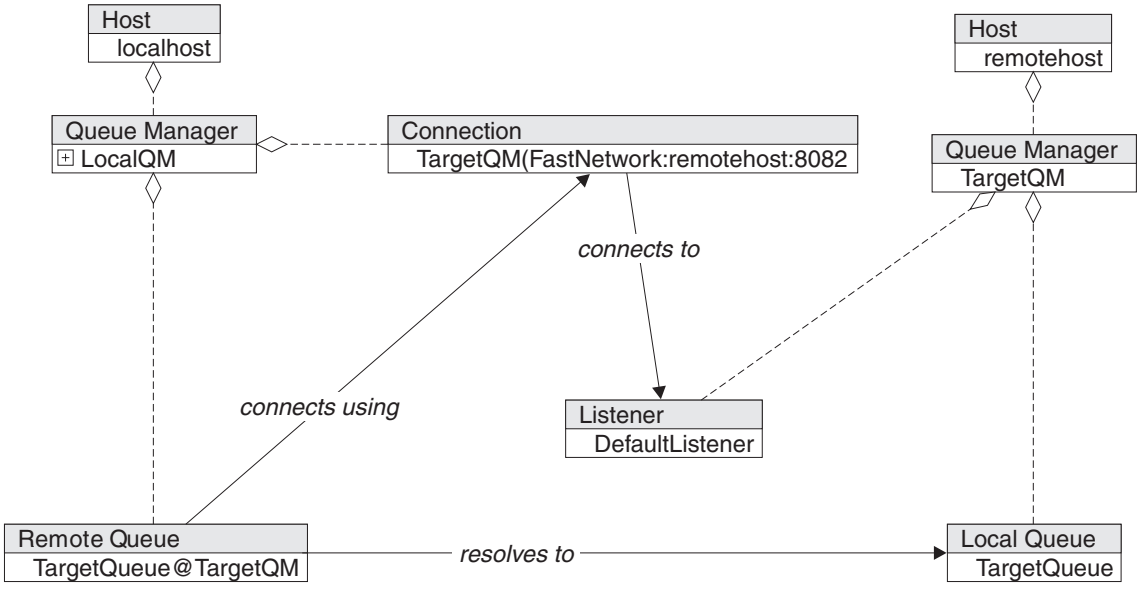

Figure 59. A remote queue reference.

The message resolution for a put on LocalQM to TargetQueue@TargetQM works as follows, see Figure 60.

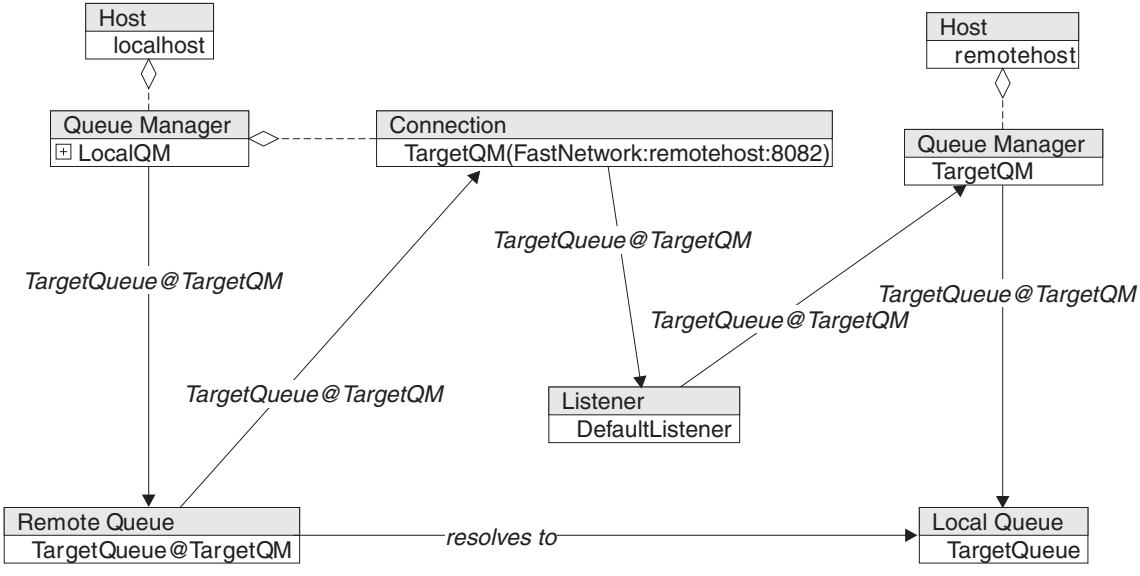

Figure 60. Message resolution for a put.

The message route is as follows:

- The message is put on LocalQM addressed to TargetQueue@TargetQM.
- LocalQM performs queue resolution and finds the remote queue reference as an exact match. LocalQM places the message onto the remote queue reference.
- The remote queue reference then performs connection resolution. It looks for a connection that will allow it to pass the message to the queue manager owning the final queue. The remote queue reference finds the connection definition called TargetQM and passes the message to it.
- The connection definition now moves the message to its partner listener, which puts the message to the remote queue manager.
- The remote queue manager performs queue resolution just as if the message had been put locally, finds TargetQueue@TargetQM, and puts the message on it.

Although the connection definition and listener are vital to the message resolution they do not affect the routing in this example, and so we can omit them for the sake of clarity. See Figure 61.

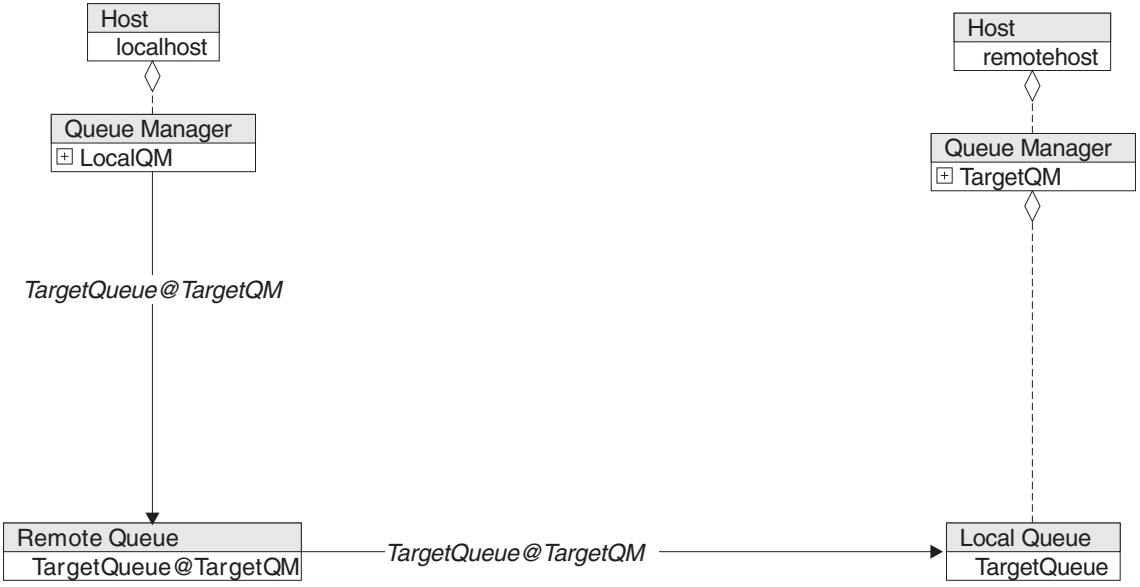

Figure 61. Message resolution for a put

In later examples the connection definitions play a more important role and we will need to show them explicitly. For now we will assume the presence of the logical link formed by the listener and not show them in the diagrams. It is often much more convenient to use a simplified view of the message route. We can do this by thinking of the four elements that contribute to this message resolution as a single, composite, entity. This entity is a Message Route, see Figure 62 on page 169.

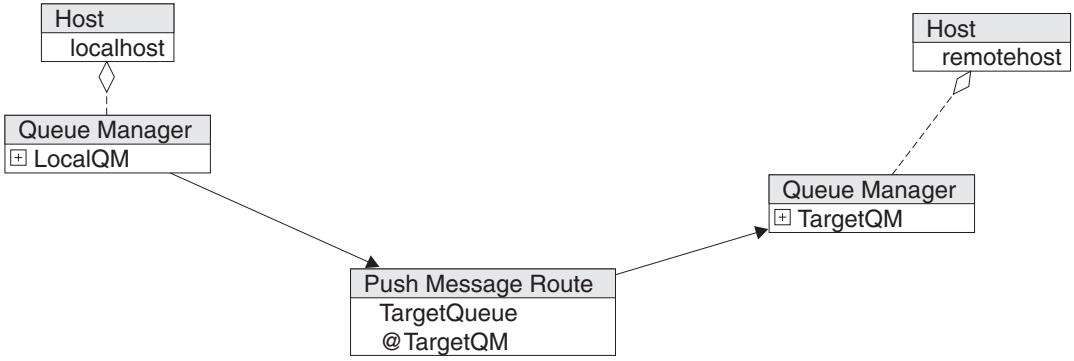

Figure 62. A message route entity.

Here we see the message route that indicates that all messages put to LocalQM and addressed to TargetQueue@TargetQM will be moved directly to the destination. A Message Route is valid only if all the necessary components (Connection Definition, Listener, Remote Queue Definition, and destination queue) are present and correctly configured.

The Message Route is defined as a Push Message Route because messages are pushed from the source queue to the destination queue, by LocalQM.

Aliases on Remote Queue

We can use aliases on the remote queue, as the last step is simply queue resolution performed on TargetQM. The Queue Alias on the target queue appears to the local system as if it were a queue. The remote queue definition on the local system is therefore named for the Queue Alias, rather than the target queue. The diagram makes this clear (note that we have hidden the connection definition and the listener), see Figure 63 on page 170.

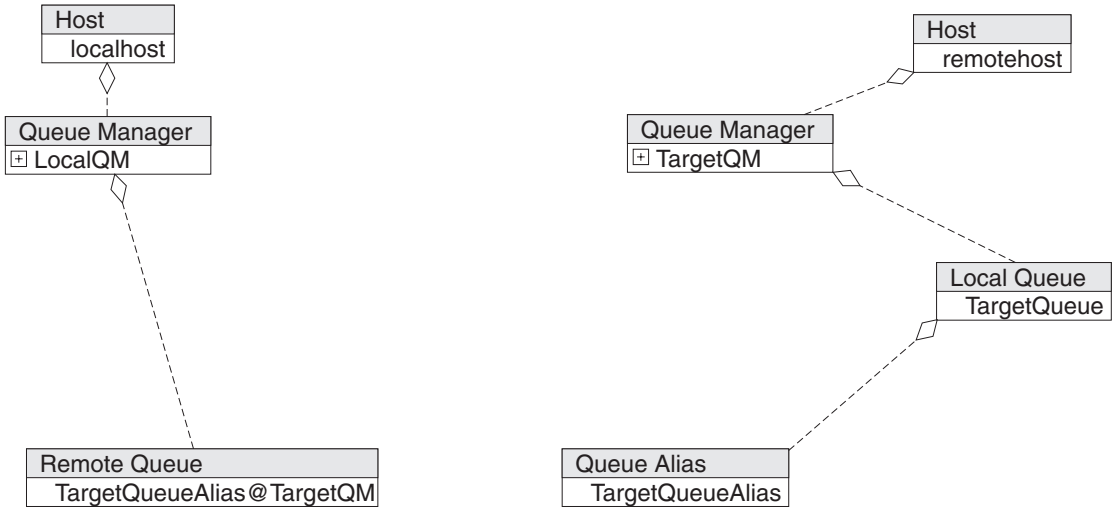

Figure 63. Using aliases on the remote queue.

Here we have defined a remote queue reference which actually refers to an alias for a queue on TargetQM. When we perform a put on LocalQM addressed to QueueAlias@TargetQM the resolution works as follows, see Figure 64.

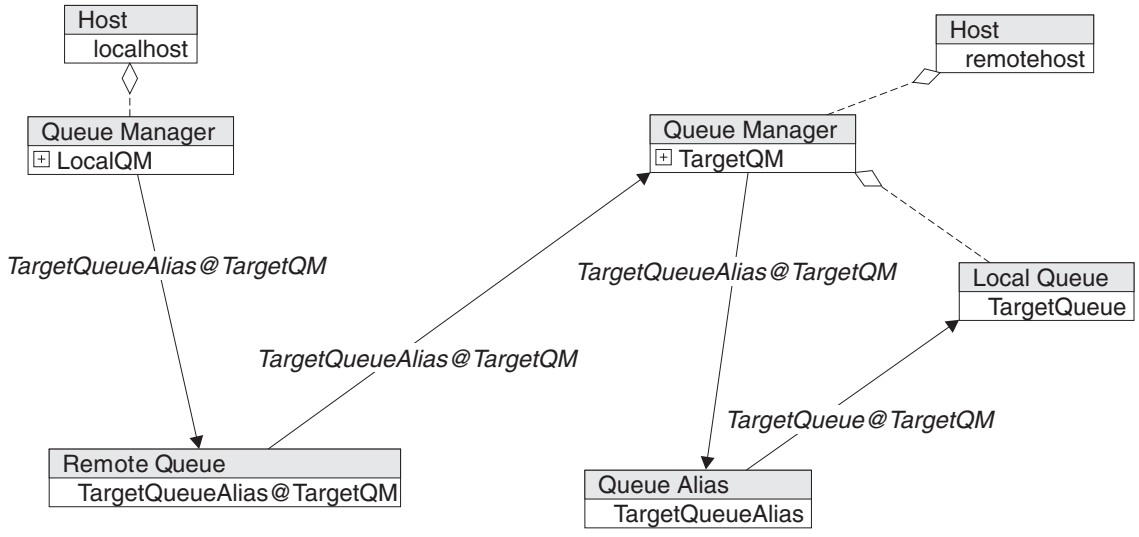

Figure 64. Message resolution for a put to a remote queue, using a Queue alias defined on TargetQM

- queue resolution on LocalQM finds the remote queue reference. The fact that this is a reference to a queue alias is completely immaterial to queue resolution.
- connection resolution works entirely as described above

- queue resolution on TargetQM now behaves exactly as local queue resolution of a queue alias described earlier.

Note that the destination name for the message remains QueueAlias@TargetQM until queue resolution onTargetQM. The Remote Queue Definition completes the requirements for another Message Route, see Figure 65.

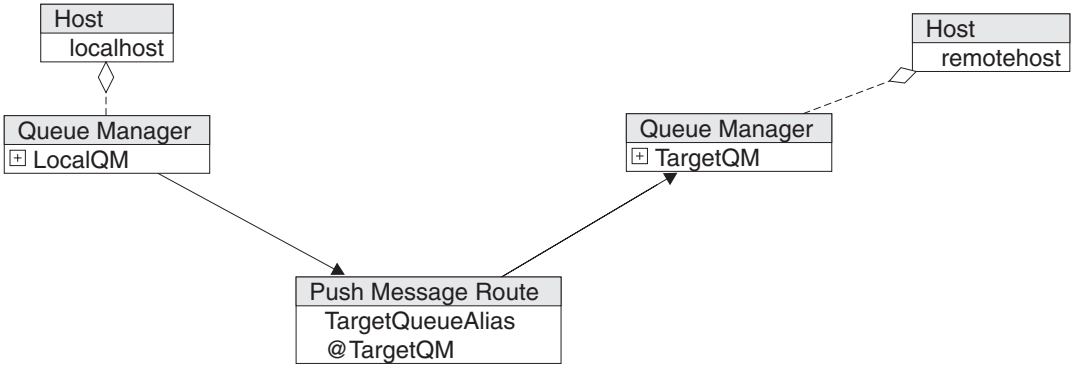

Figure 65. Message route entity of messages put to TargetQueueAlias on TargetQM

Parallel Routes

The use of aliases described in the previous section allows the creation of parallel routes between a source and a destination. This is sometimes desirable where we wish to send messages synchronously if possible, but asynchronously if the remote end is not currently connected. We can do this with the following setup, see Figure 66 on page 172.

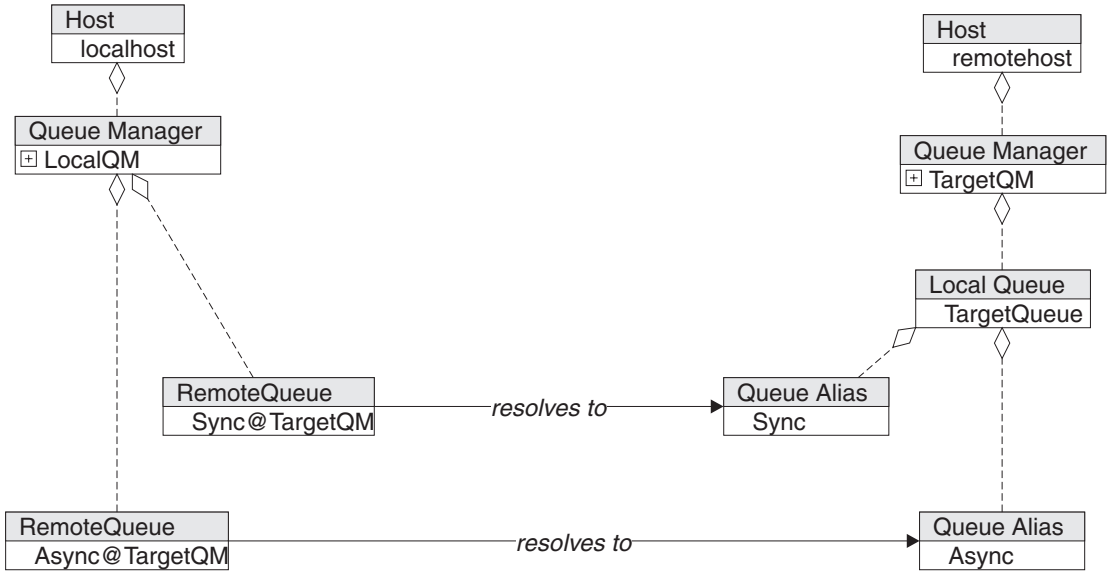

Figure 66. Creating parallel routes between source and destination.

Here we have defined two aliases on the target queue. One alias will be used to route synchronous traffic to the target queue, one will be used to route asynchronous traffic.

On LocalQM we have defined two remote queue definitions, one pointing at each alias. We can create an asynchronous Remote Queue Definition called Async@TargetQM, and a synchronous Remote Queue Definition called Sync@TargetQM. By choosing the name of the queue that we put to (Sync@TargetQM or Async@TargetQM) we can choose the route that the message follows, even though the destination is the same. First, the resolution of the synchronous route by putting a message to Sync@TargetQM, see Figure 67 on page 173.

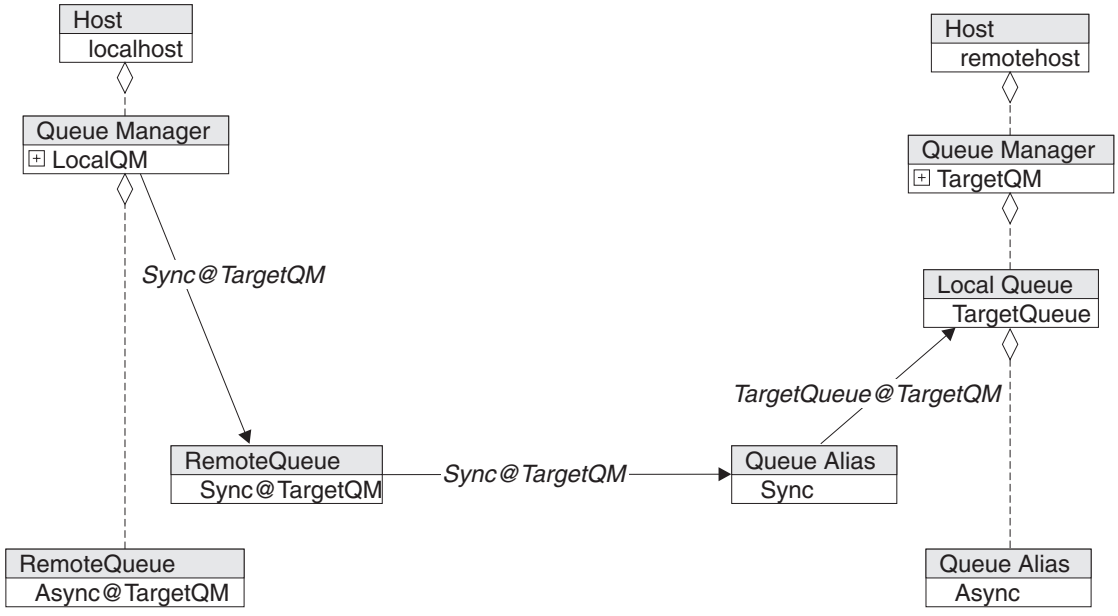

Figure 67. Resolving the synchronous route.

And secondly the asynchronous resolution using AsyncAlias@TargetQM, see Figure 68.

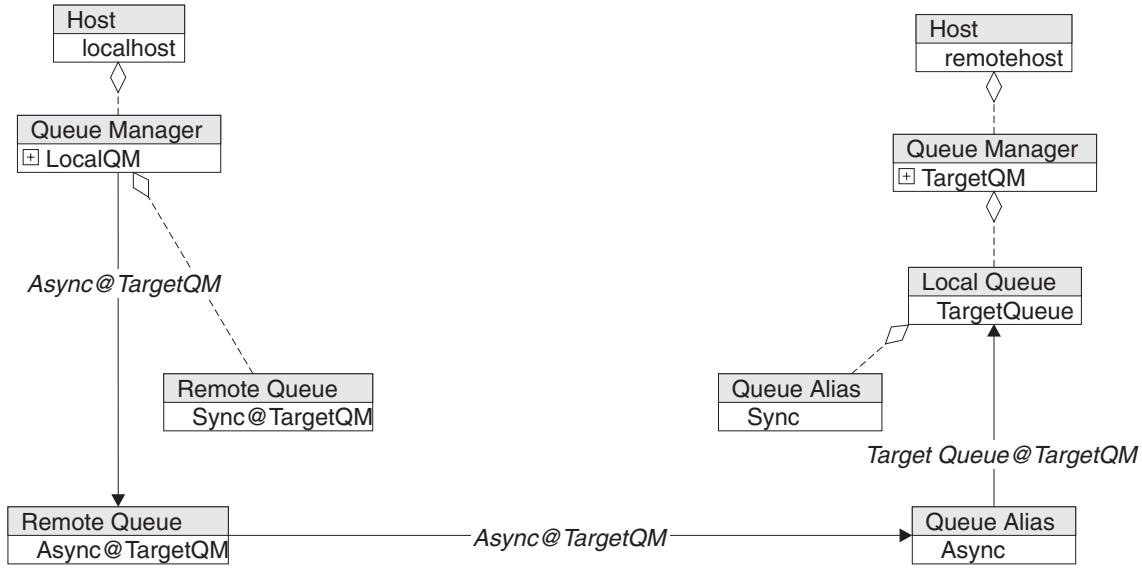

Figure 68. Resolving the asynchronous route.

We could choose to view this as a pair of Push Message Routes, see Figure 69.

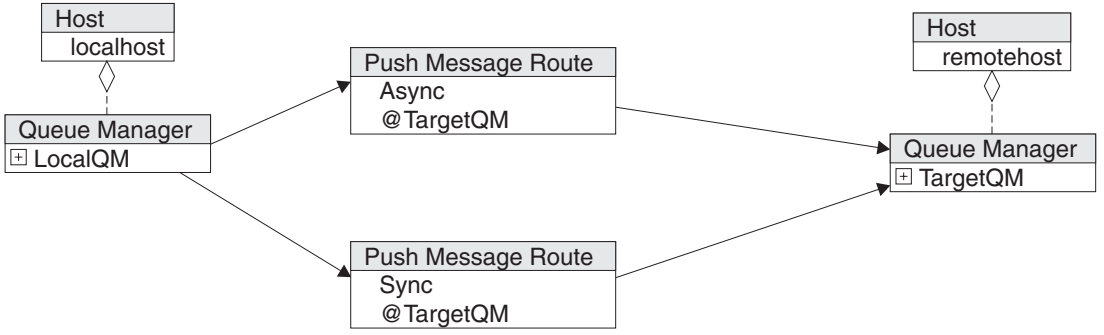

Figure 69. A pair of push message routes.

Chaining Remote Queue References

Remote queue references can be chained together to form a longer route. This requires the use of 'Via' connections, and so the technique is described later in this document.

Pushing Store And Forward Queues

WebSphere MQ Everyplace has a queue type that accepts messages on a queue manager basis rather than on a queue basis. These are called Store and Forward queues, S&F queues for brevity). S&F queues maintain a list of queue manager names, called 'Queue Manager Entries', or QME for brevity. The S&F queue will accept messages for any queue manager represented by a QME. This acceptance is independent of the destination queue name, and so allows one queue (the S&F queue) to route all messages for a given, or several given queue managers.

S&F queues can operate in two modes, pushing mode and pulling mode. In pushing mode the messages are moved to the next queue manager just as with remote queue references. In pulling mode the messages are removed from the S&F queue by the action of a Home Server Queue. This section deals only with the pushing of messages, pulling messages with a home server queue is described in another section. A typical pushing S&F queue system might look like Figure 70 on page 175.

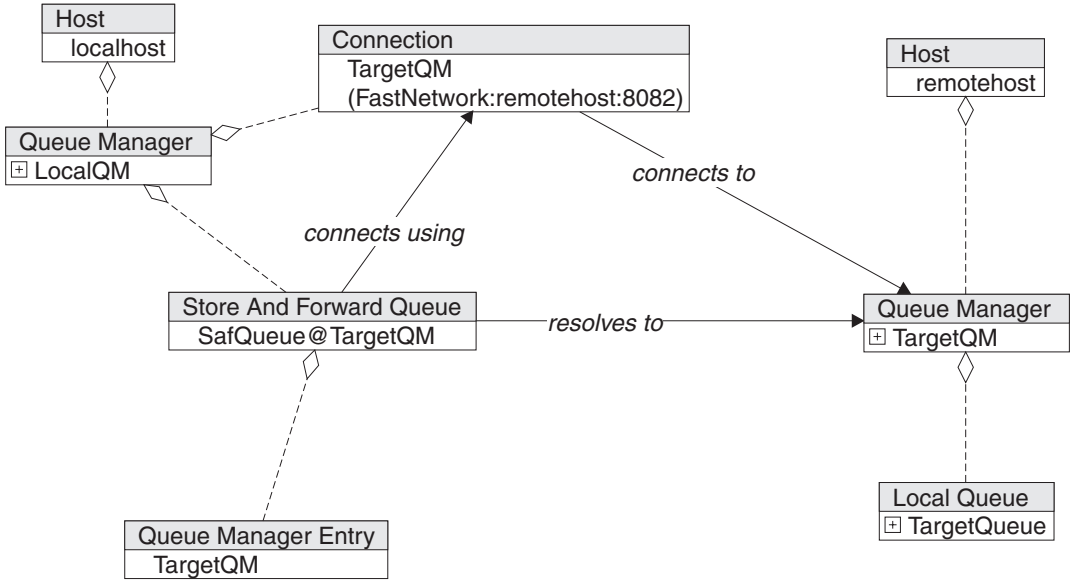

Figure 70. A typical pushing S&F queue system.

A S&F queue called SafQueue has a queue manager entry (QME) for TargetQM. This allows it to accept messages for any queue on TargetQM. In common with ordinary Remote Queues, a Store and Forward queue requires a connection definition/listener pair set up in order to push messages. Unlike a normal Remote Queue Definition, a Store and Forward Queue effectively pushes to a Queue Manager rather than to a queue. The message arrives at the Queue Manager, where queue resolution is performed. When a message is put to LocalQM addressed to TargetQ@TargetQM the resolution is as follows, see Figure 71 on page 176.

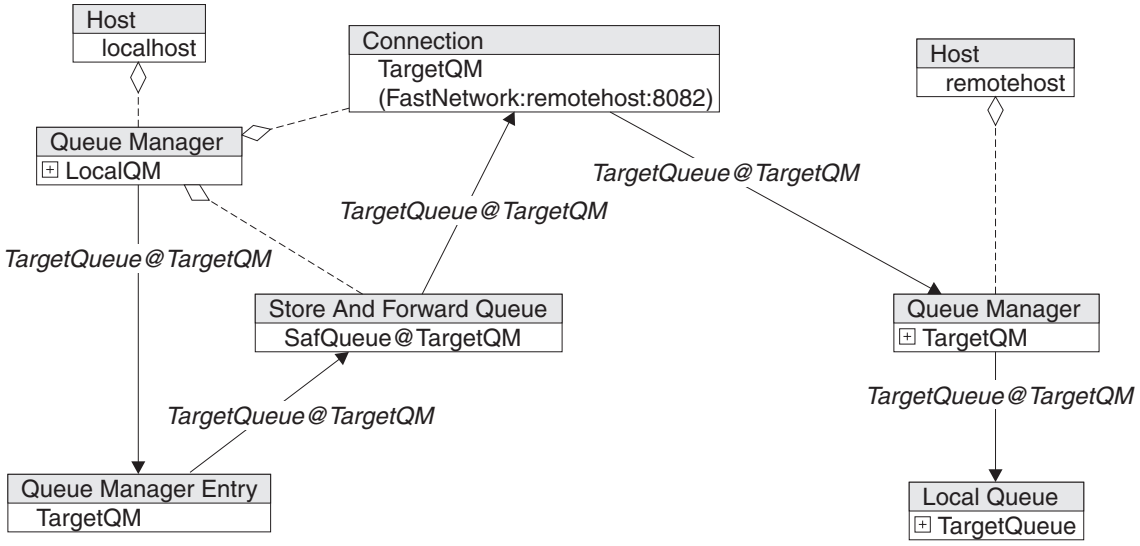

Figure 71. Routing of a message put to LocalQM and addressed to TargetQ@TargetQM

- LocalQM performs queue resolution which finds the queue manager entry TargetQM on SafQueue. LocalQM puts the message to the QME.
- Putting a message to the QME is equivalent to putting the message on the S&F queue owning the QME.
- The S&F queue performs connection resolution and finds the connection definition, and so uses it to push messages to RemoteQM just as the remote queue reference did in an earlier section.
- The queue manager then performs queue resolution and places the message on the target queue.

The Store and Forward queue forms part of a Multi Message Route. This abstract entity represents the potential for messages addressed to any queue on TargetQM, and so is called *@TargetQM, see Figure 72 on page 177.

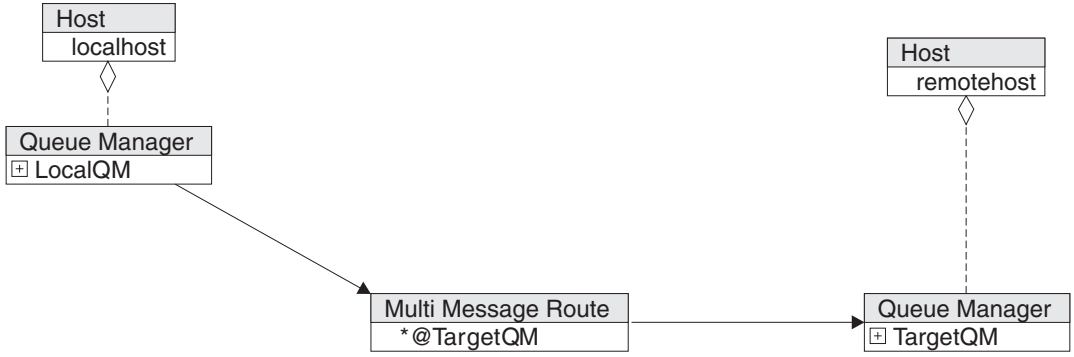

Figure 72. A multi message route.

Undelivered Messages: If there is no queue to which the message can be put, then it is not delivered. This prevents any further messages from being pushed from that Store and Forward queue to that Queue Manager.

Store and Forward Queues and Remote Queue References

Because S&F queues can accept messages for any queue on a given queue manager, they can appear to be in conflict with a remote queue reference. In such cases the remote queue reference takes precedence, because it is more specific. So if we look again at our S&F queue resolution, but add a remote queue reference we can see that the message route resolution changes immediately, the S&F queue becomes irrelevant. We can see this in Figure 73

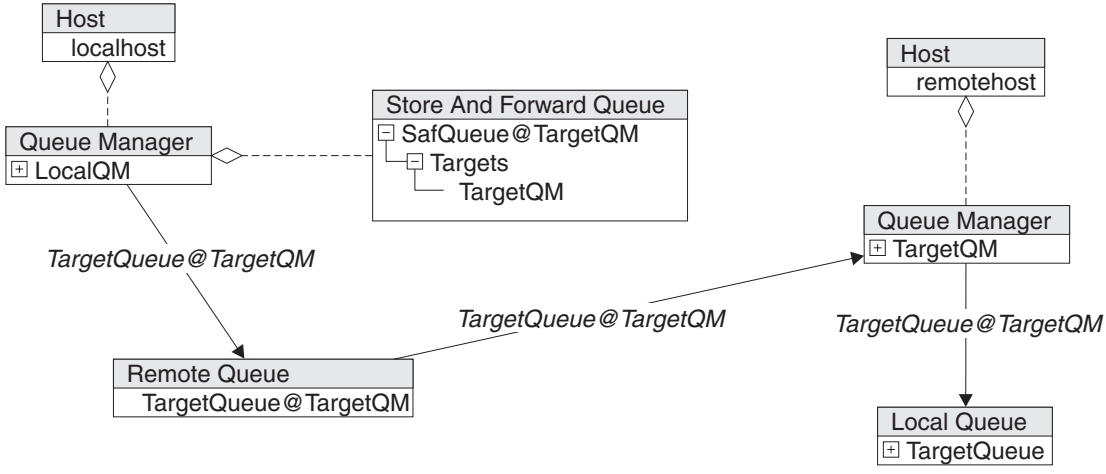

Figure 73. How routes using remote queue definitions take precedence over store-and-forward queue routes

that queue resolution finds the best (most exact) match for the message address.

So a message put to QueueAlias@TargetQM goes via the S&F queue (asynchronous transmission), but a put to TargetQueue@TargetQM goes synchronously via the remote queue reference.

Chaining Store and Forward Queues

Pushing store and forward queues can be chained together into a more complex route, see Figure 74.

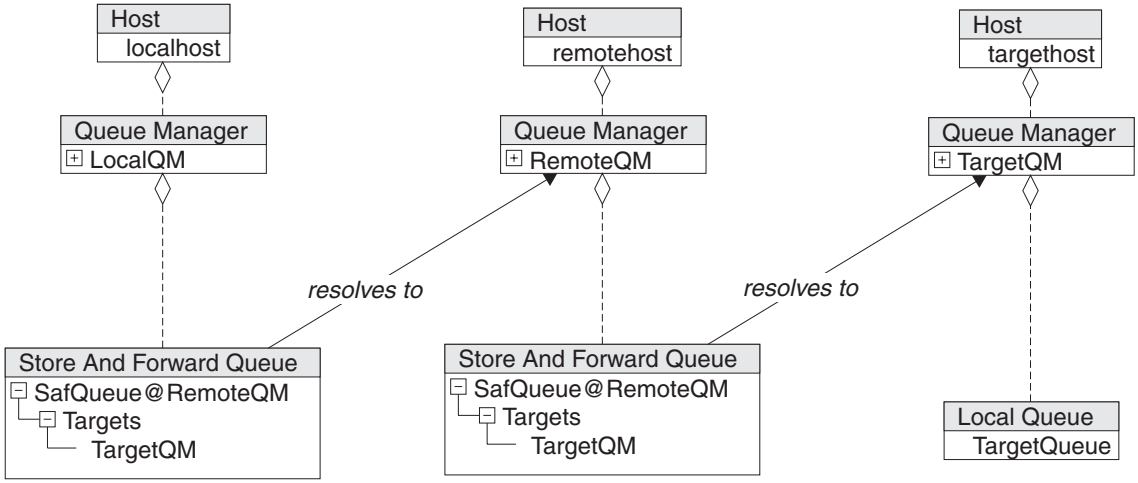

Figure 74. Pushing S&F queues chained together.

Note that the Store and Forward queue on LocalQM (SaFQueue@RemoteQM) has a Queue Manager Entry for TargetQM, but actually pushes to RemoteQM. LocalQM requires a connection definition to RemoteQM, but not to TargetQM. A message can then be transported via the intermediate S&F queue, see Figure 75 on page 179.

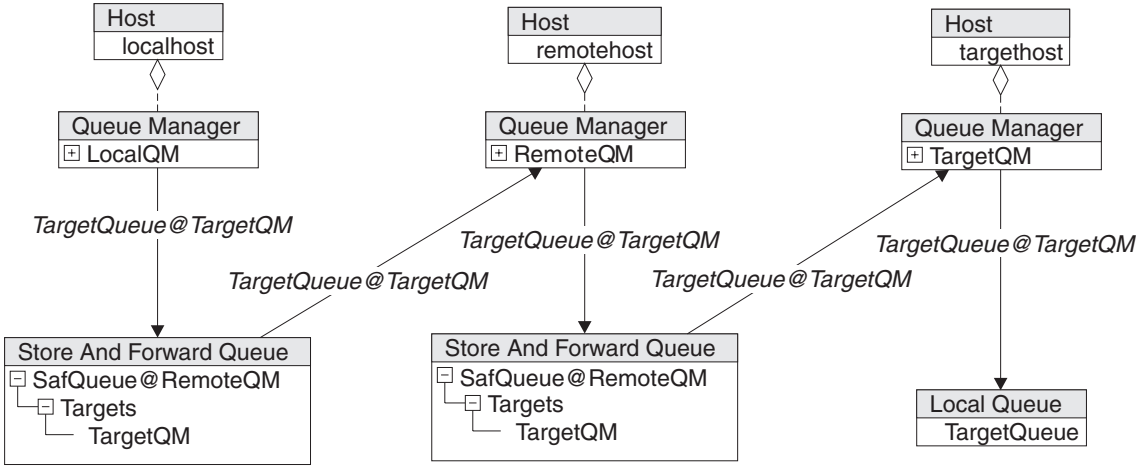

Figure 75. Transporting messages via an intermediate S&F queue.

This works because the combination of queue resolution and connection resolution on LocalQM results in the message being put to the S&F queue on RemoteQM, which can then move it to its destination. The chain of Store and Forward Queues could be arbitrarily long, with each queue manager in the chain needing to know only about the next queue manager in the chain. The Message Routes express this very succinctly, see Figure 76.

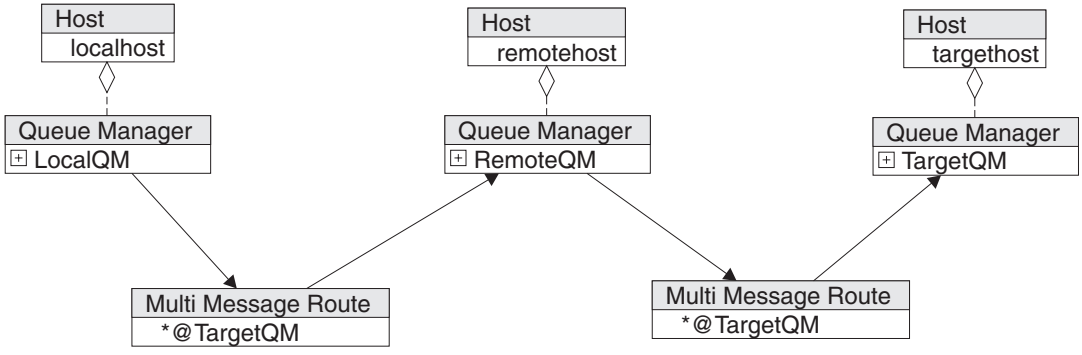

Figure 76. A chain of store and forward queues.

Home Server Queues

Home server queues pull messages from store and forward queues. The S&F queue may be a 'pushing' S&F queue (that is, has a valid connection definition). Home server queues will only pull messages across a single 'hop', and will only pull messages whose intended destination is the local queue manager - the queue manager upon which the home server queue resides. A typical Home Server Queue configuration is illustrated below, see Figure 77 on page 180.

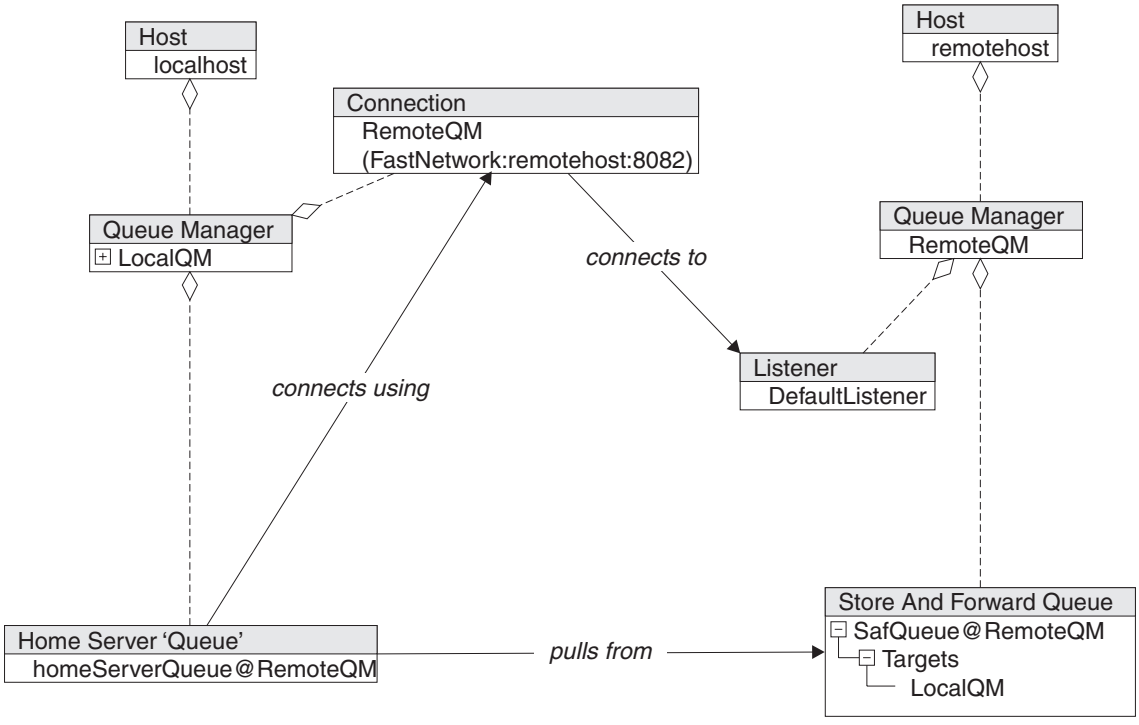

Figure 77. A home server queue configuration.

The diagram shows a simple HomeServerQueue setup. In this configuration the server queue manager has no connection definition to the client; instead it has a store queue (that is, a store and forward queue with no target queue manager) that collects all messages bound for the client. This message collection embraces all queue destinations on the client.

The client pulls the messages from the store queue using a home server queue pointing at the store queue on the client. The home server queue never stores messages itself, it collects them from the store queue and delivers them to their destinations on the client. The client makes the connection request to the server using its connection definition.

The home server queue 'homeServerQueue@RemoteQM' will attempt to pull messages from the queue manager 'RemoteQM'. It requires a connection definition to be able to do this. The home server queue will only be able to pull messages if there is a store and forward queue that is storing messages for LocalQM.

Messages that are pulled from RemoteQM are then 'pushed' to local queues on LocalQM. This is shown in the following diagram, where a Home Server Queue on LocalQM is pulling messages (for LocalQM) from RemoteQM. In this case a message for TargetQueue@LocalQM is shown being pulled, and the resolution at the queue

manager has been hidden for clarity. In reality, the Home Server Queue presents each pulled message to the local queue manager for resolution, see Figure 78.

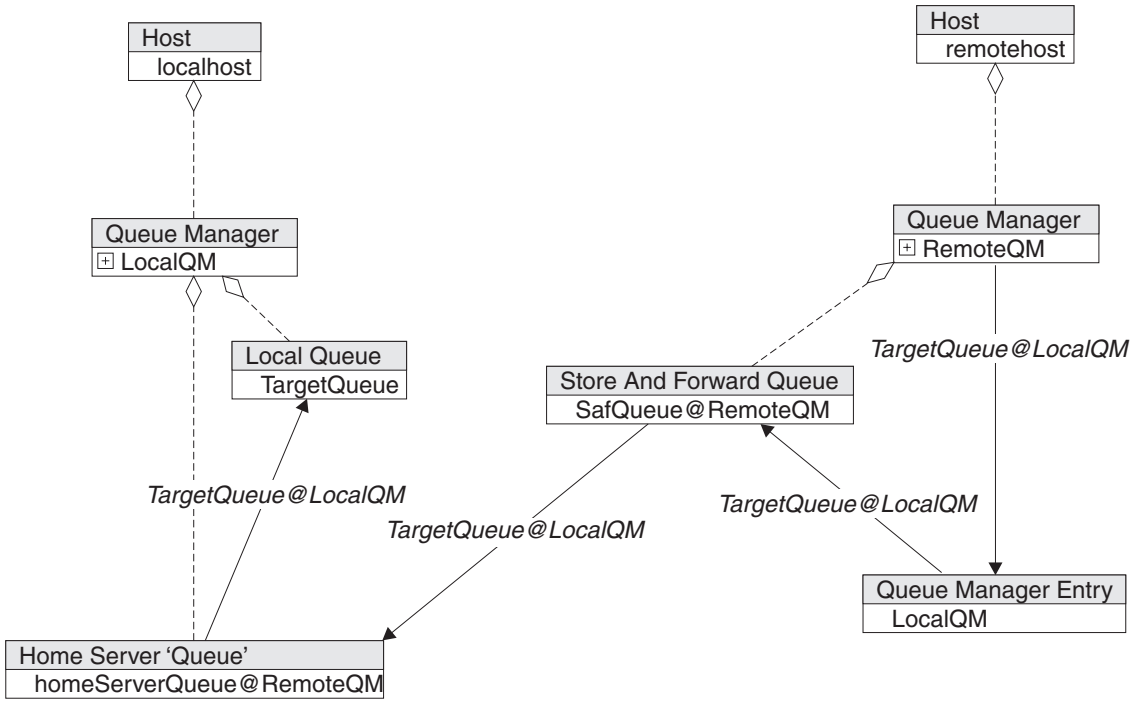

Figure 78. A home server queue pulling messages.

The pull message route can be viewed at a more abstract level, see Figure 79 on page 182.

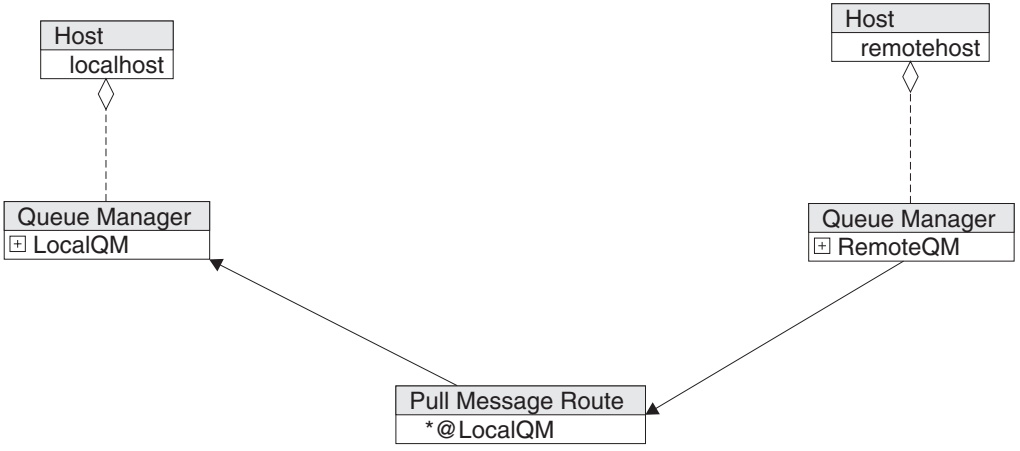

Figure 79. An abstract pull message route.

How are pulled message routes useful, and where would we use them? The most important feature of a pulled message route is that the flow of messages is under the control of the local queue manager. This makes it very useful to a client that spends much of its time disconnected. If we had to rely on the server pushing message, the server would need to continuously poll the client to check if it was available. This would not be a good solution for large numbers of clients, as much of the servers time would be spent polling for disconnected clients. Instead, with a Home Server queue, each client pulls messages when it is connected, and the server only has to deal with real requests from connected clients. One concrete example of this is the administration of queue managers that do not have listener capability. Administration messages for the client are placed upon a Store and Forward queue. The client can then use a Home Server queue to pull these when it is connected. Administration reply messages could then be pushed using normal push remote queue, see Figure 80 on page 183.

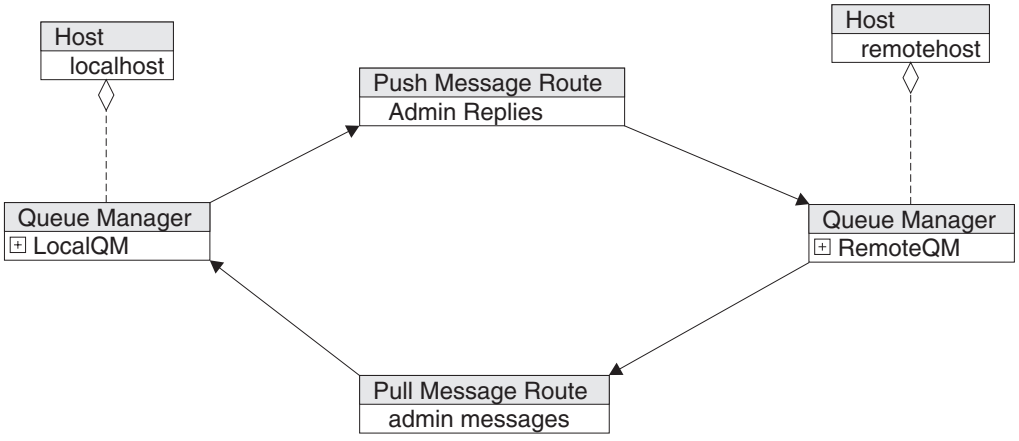

Figure 80. Administering queue managers that do not have listener capability.

Via Connections

Via connection allow messages to be routed via an intermediate queue manager. For example, we might wish messages from LocalQM to travel to TargetQM via RemoteQM. We can already do this with 'pushing' store and forward queues, but via connections provide an alternate mechanism, see Figure 81 on page 184.

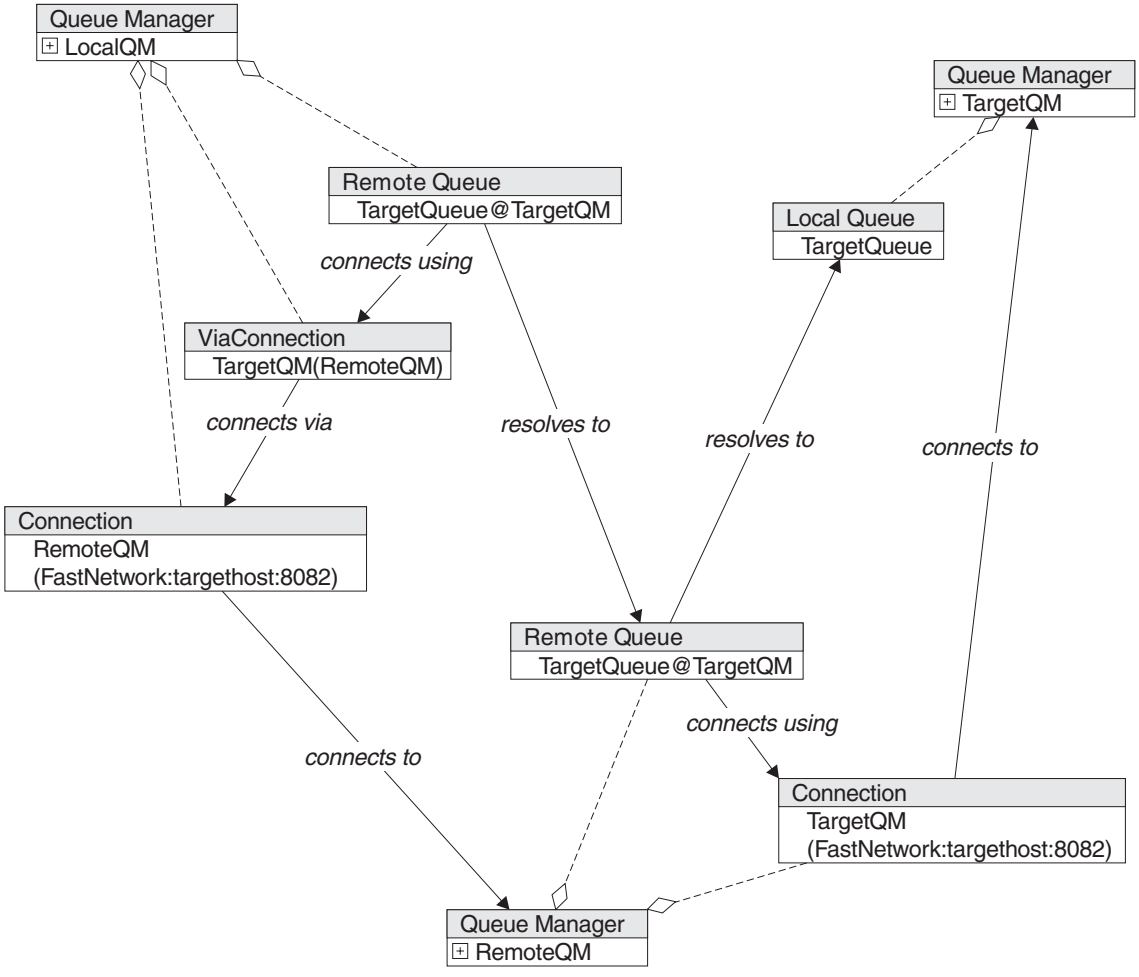

Figure 81. Via connections

The diagram illustrates the components being used. The connection definition called 'TargetQM' on LocalQM does not contain the address of TargetQM, but simply refers to the connection definition called 'RemoteQM'. This means that any messages destined for TargetQM will be sent to RemoteQM, and we assume that RemoteQM will be able to move the messages onward. In the diagram above, RemoteQM has the necessary connection to move the message to TargetQM.

The message flows as expected, see Figure 82 on page 185.

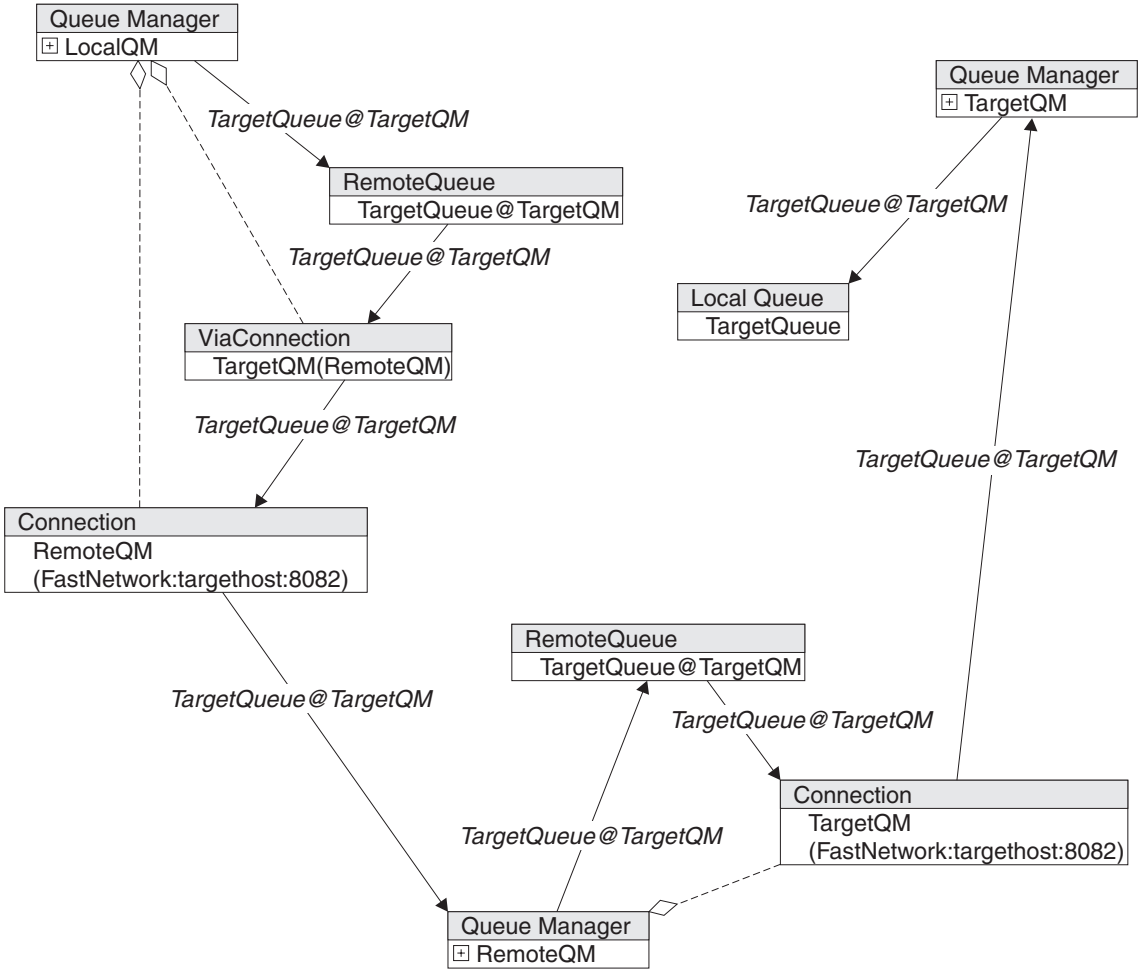

Figure 82. Message flow using a via connection

The Remote Queue on LocalQM uses Connection Resolution to find the Via Connection. This then passes the message on to the real connection which moves the message to RemoteQM. On RemoteQM queue resolution proceeds as for the simple case. We can show the topology most clearly using Message Routes, see Figure 83 on page 186.

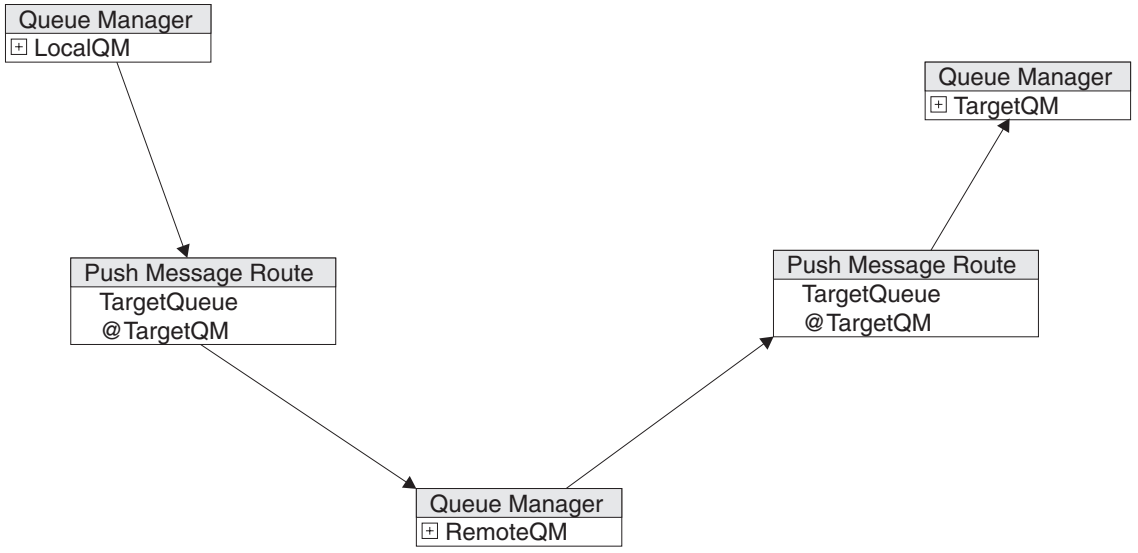

Figure 83. Via connections expressed using message route schema

This is known as 'chaining remote queues'. The central remote queue can be synchronous, asynchronous, or even a store and forward queue.

Rerouting with Queue Manager Aliases

Earlier in this document we described Queue Manager Aliases and said that they had a more important part to play in routing. To illustrate, we consider an example of a common situation often known as fail-over. In this we have a client communicating with a server, and we have a backup server that can be used if the main server fails, or is taken down for maintenance, see Figure 84 on page 187.

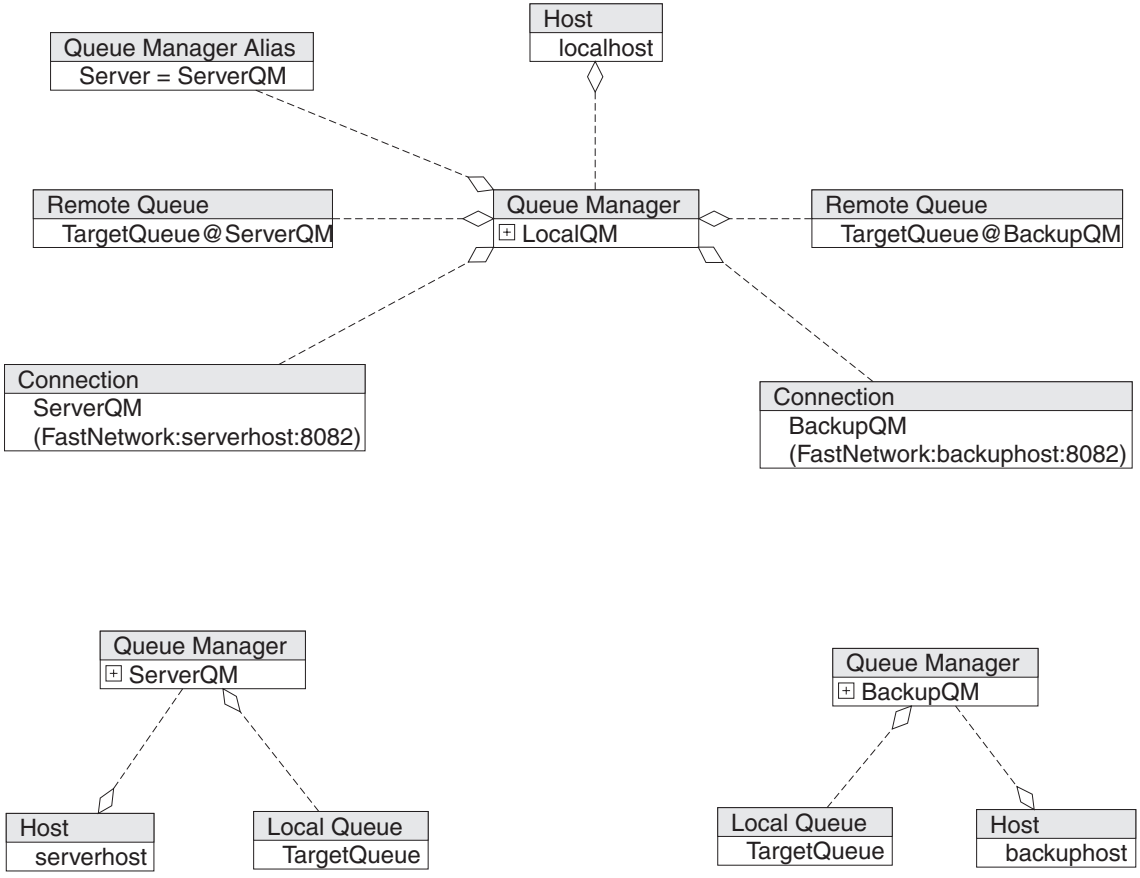

Figure 84. Queue manager aliases and fail-over.

Here we see the local client queue manager, with a connection to ServerQM and a remote queue definition for TargetQueue@ServerQM. The server (bottom left) has a local queue as the target for our example message, and this is mimicked by the backup server (bottom right). Additionally, on the client queue manager, there is a Queue Manager Alias mapping the name Server to ServerQM. This mapping is then used for messages put to the server. The message resolution is shown below for the normal operating configuration, where a message put to TargetQueue@Server is directed to TargetQueue@ServerQM, see Figure 85 on page 188.

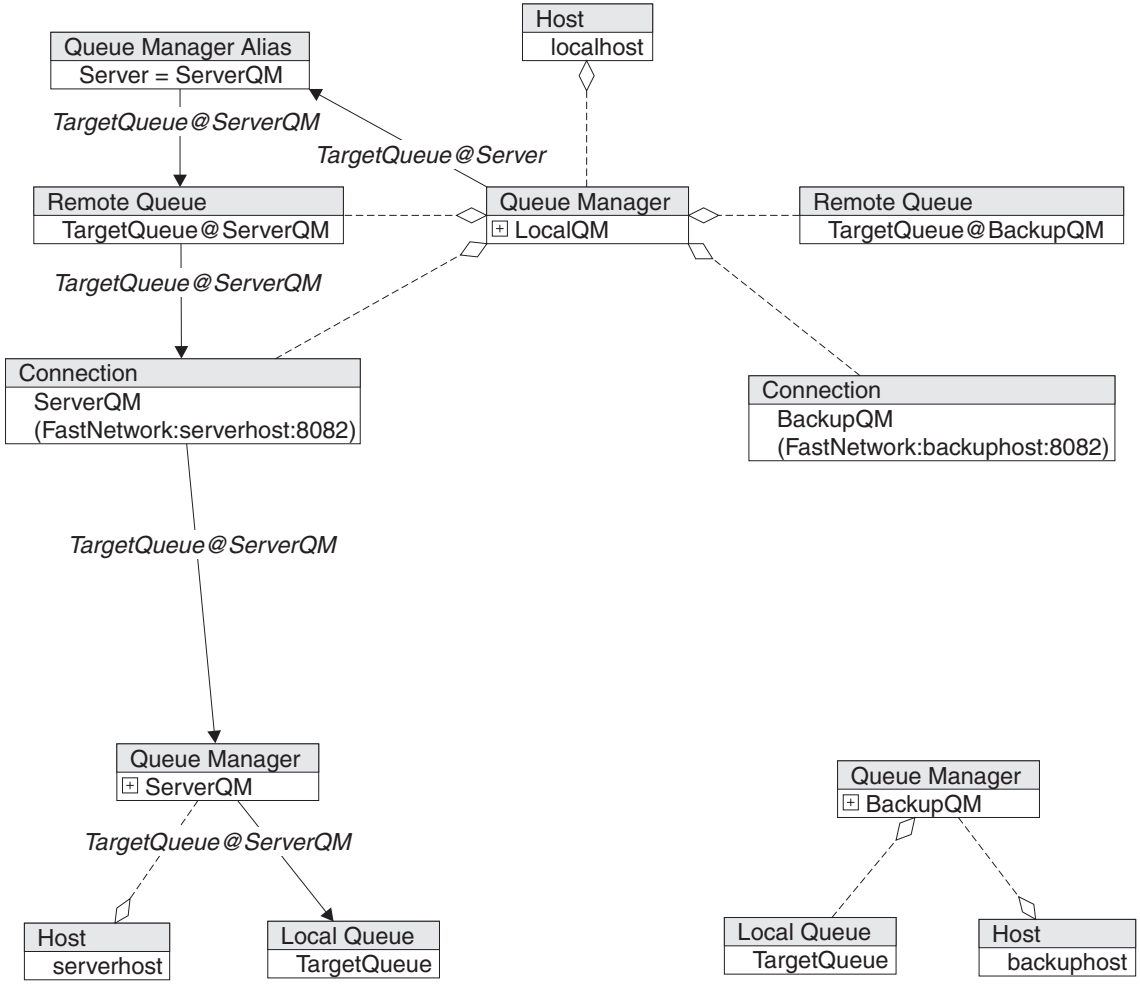

Figure 85. Routing traffic using a "server" alias

The alias maps messages for Server to ServerQM, and this selects the remote queue definition TargetQueue@ServerQM. If the network administrator needs to route traffic to the backup server, only the Queue Manager Alias needs to be changed (it is in fact deleted, and recreated with a different target name, in this case BackupQM, see Figure 86 on page 189).

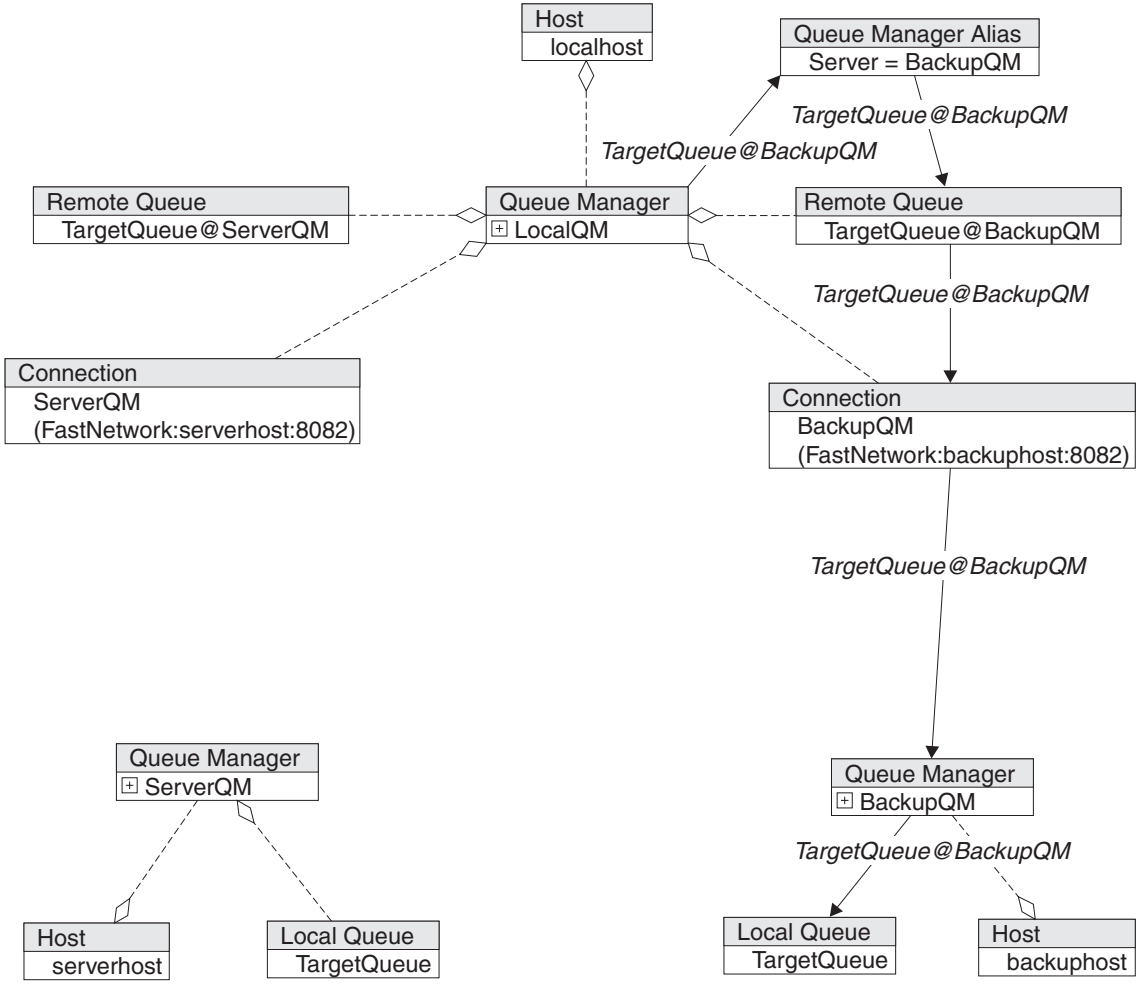

Figure 86. Routing traffic to the backup server, using a "server" alias

The change of alias reroutes the message to a different remote queue, and hence on to the backup queue manager and to TargetQueue@BackupQM. In essence what we have is a pair of message routes, one to each server, and we are using a Queue Manager Alias to choose between the message routes, see Figure 87 on page 190.

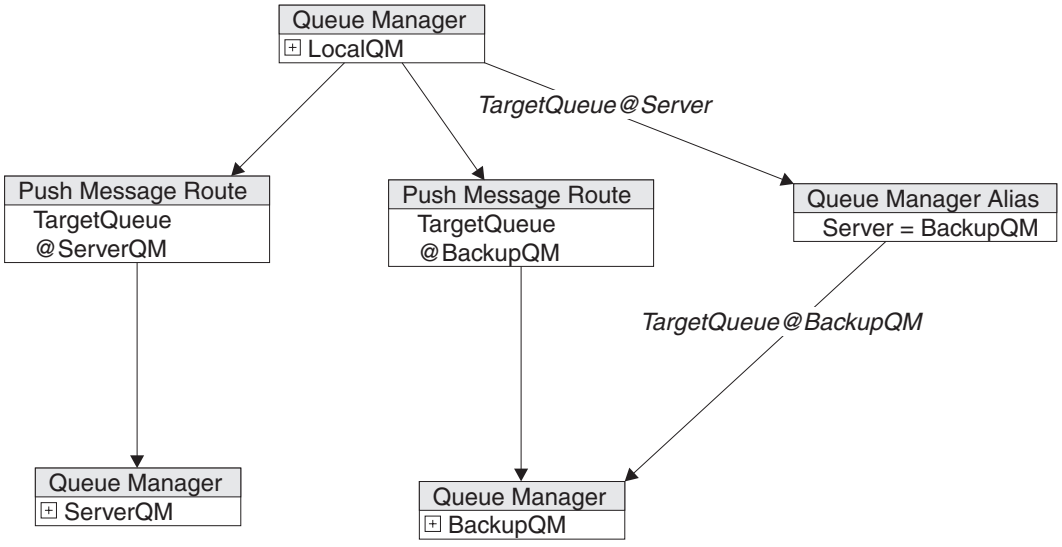

Figure 87. Choosing between message routes.

The example above required a change to every client on a system that requires rerouting to a backup server. If there are a large number of clients this may be impractical. In addition, each client requires two complete message route definitions (a remote queue and a connection definition for each). It would be far more elegant to avoid the need to change the client. We can do this by having a second server ready to listen on the same address and port as the first. When the administrator wishes to change over the first can be brought down, and the second can change over. In this circumstance it may be convenient to keep the names of the servers different. The backup server can be given a Queue Manager Alias mapping BackupQM to ServerQM. This will allow BackupQM to impersonate ServerQM.

Warning

Changing the WebSphere MQ Everyplace network topology in this fashion is not always a wise thing to do. Care must be taken to ensure that there are no 'in doubt' messages that would be affected by the change. If a message is put with a non-zero confirm id, and then the WebSphere MQ Everyplace network topology is changed to alter the routing of the subsequent confirmGetMessage call, then the unconfirmed message will not be found. WebSphere MQ Everyplace protocol treats a failure to confirm a put as an indication that the put message has been confirmed already, and therefore assumes success. This could leave an unconfirmed message on a queue, which represents a loss of a message, and therefore breaks the assured delivery promise.

Since WebSphere MQ Everyplace uses the same two step process to assure delivery of asynchronously sent messages, changing the network topology can break the assured delivery of asynchronous message sends.

WebSphere MQ Everyplace WebSphere MQ Bridge Message Resolution

A connection between WebSphere MQ Everyplace and WebSphere MQ queue managers involves a collection of objects. The following diagram shows only the entities that form the communications link between the two queue managers, see Figure 88.

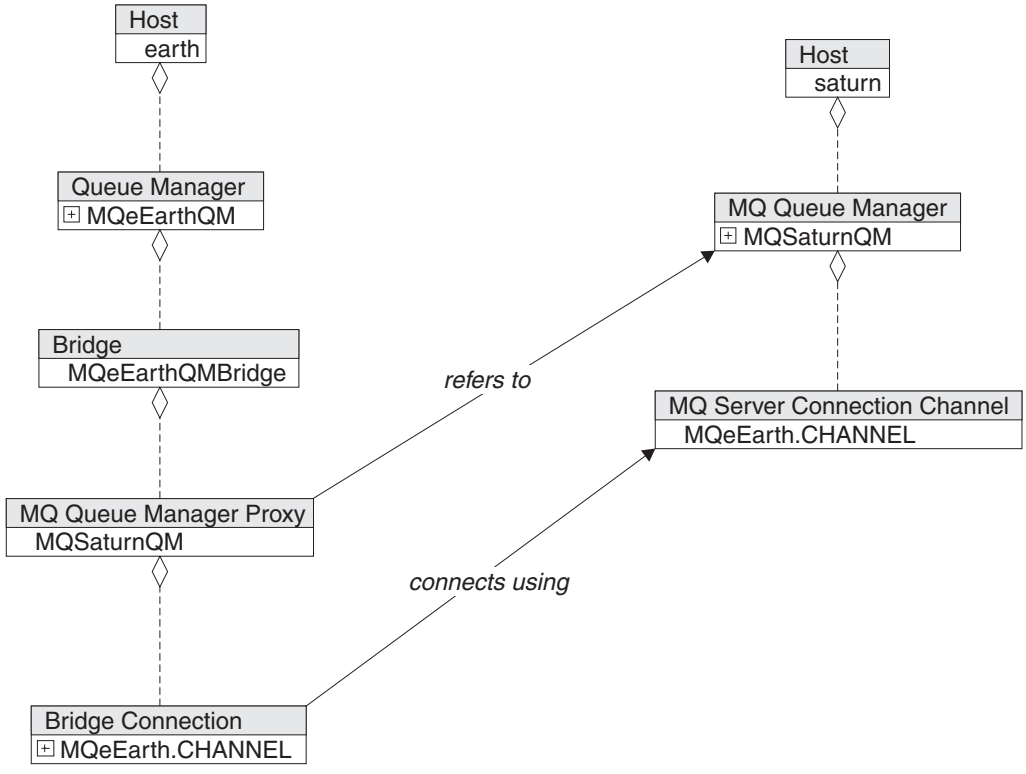

Figure 88. Connecting WebSphere MQ Everyplace and WebSphere MQ queue managers.

The important entities are:

- (Bridge)MQeEarthQMBridge - a bridge resource owned and controlled by the MQeEarthQM queue manager.
- (MQ Queue Manager Proxy)MQSaturnQM - describes MQSaturnQM and how to connect to it.
- (BridgeConnection)MQeEarth.CHANNEL - a communications path between MQeEarthQM and MQSaturnQM.
- (MQ Server Connection Channel) MQeEarth.CHANNEL - a standard WebSphere MQ server channel providing an entry point to MQSaturnQM for MQeEarthQM.

These entities are described in more details in other parts of the manual (reference??), and we will treat them as a transport layer for queue connectivity. These entities will be used in the following examples of bridge connectivity, but will not be shown in the diagrams.

Pulling Messages From WebSphere MQ

By setting up a Transmit queue on WebSphere MQ, and a bridge listener on a WebSphere MQ Everyplace queue manager we can enable the latter to pull messages from the former. Although in theory this is sufficient to pull messages from the transmission queue, we cannot place messages onto the transmission queue without creating extra queues on a WebSphere MQ queue manager.

Single pull route

To allow the messages to be correctly routed we create extra queues on a WebSphere MQ queue manager. The simplest form is to create a remote queue on WebSphere MQ to allow messages addressed to TargetQueue@MQeEarthQM to be accepted by the WebSphere MQ queue manager, see Figure 89 on page 193.

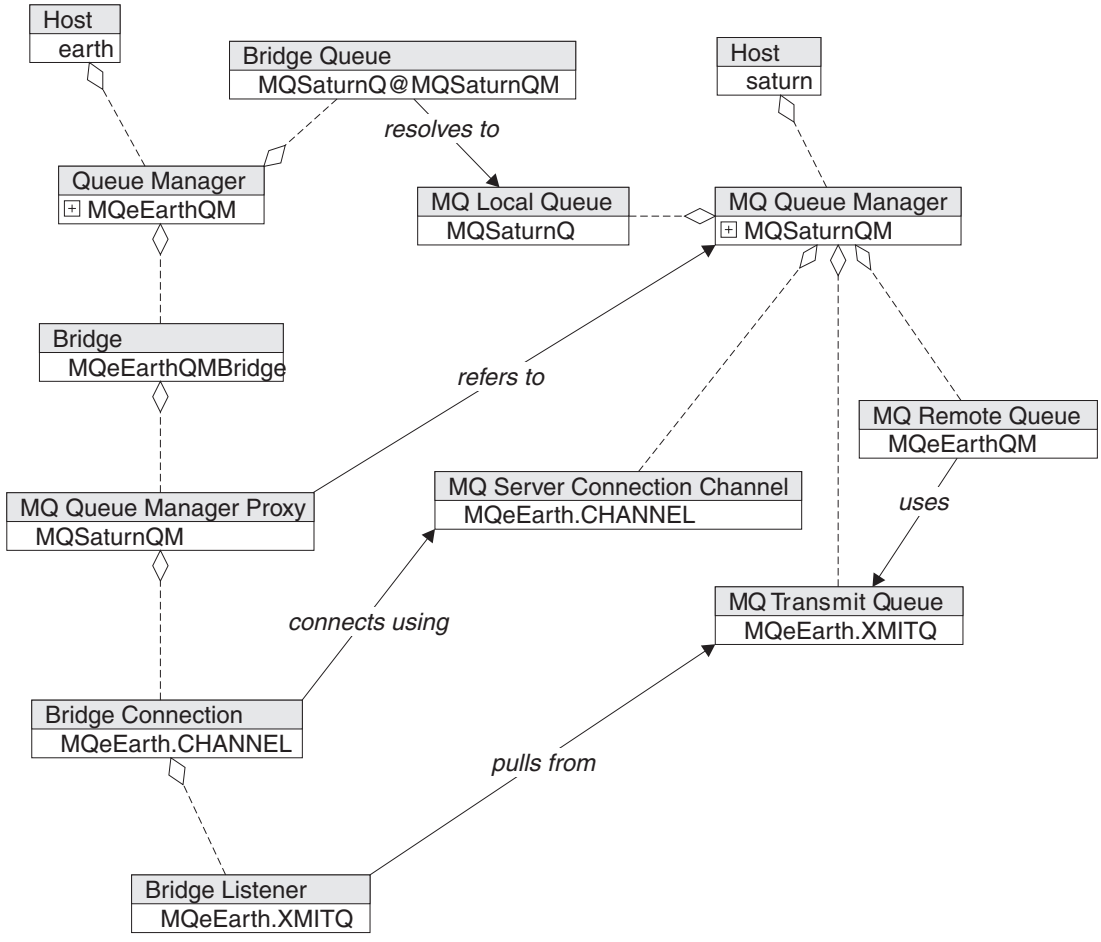

Figure 89. Creating a remote queue on WebSphere MQ.

Messages addressed to TargetQueue@MQeEarthQM are placed upon the WebSphere MQ Transmit queue. The bridge listener then pulls them from the transmit queue and presents them to the WebSphere MQ Everyplace queue manager. Message resolution then takes place, see Figure 90 on page 194.

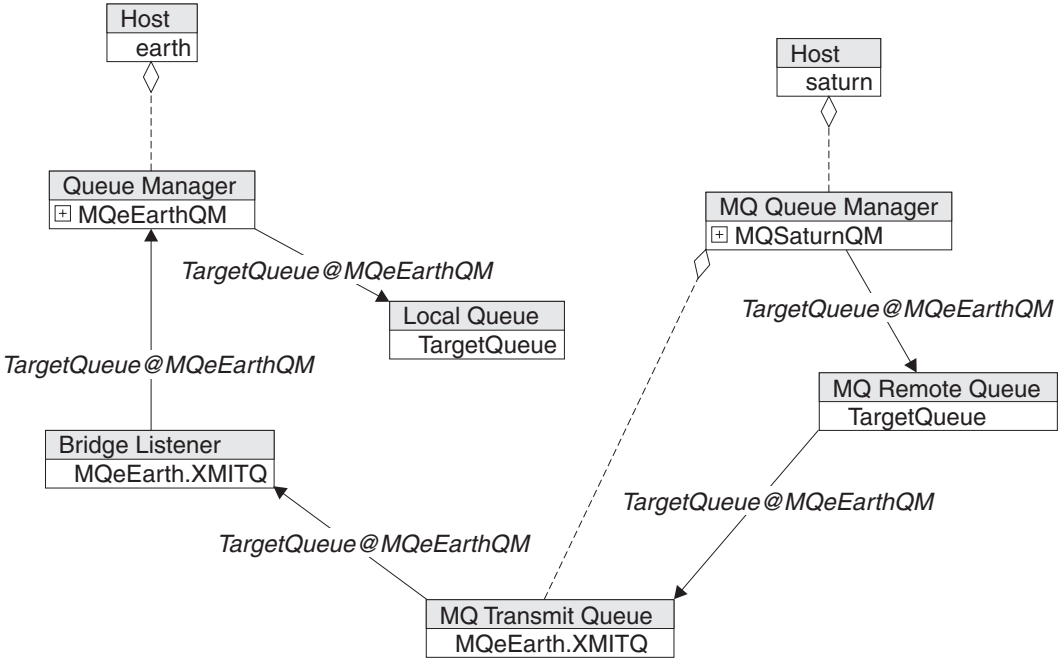

Figure 90. Bridge listener pulling from a WebSphere MQ Everyplace transmit queue

This is effectively a single pull message route, see Figure 91.

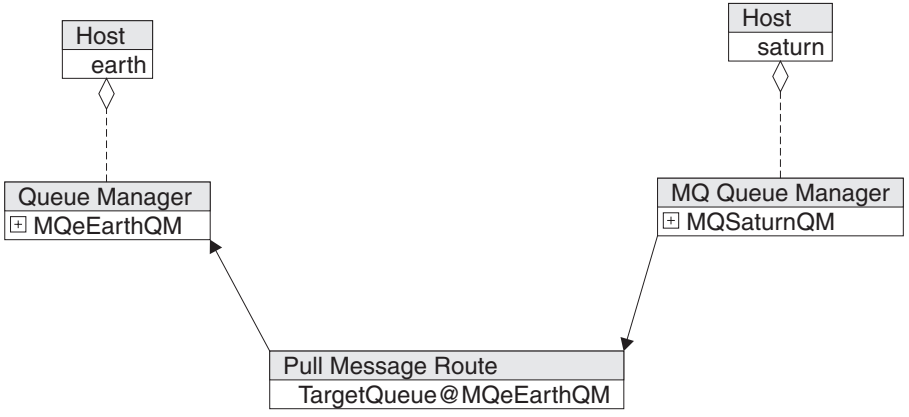

Figure 91. A single pull message route.

Multiple Pull Route

It is generally more efficient to use a multiple pull message route as this requires the same number of resource definitions, but will handle all the traffic for WebSphere MQ Everyplace queue manager. This is done using a Remote queue manager alias on

WebSphere MQ (effectively a remote queue where the target queue name is the same as the target queue manager name, see Figure 92).

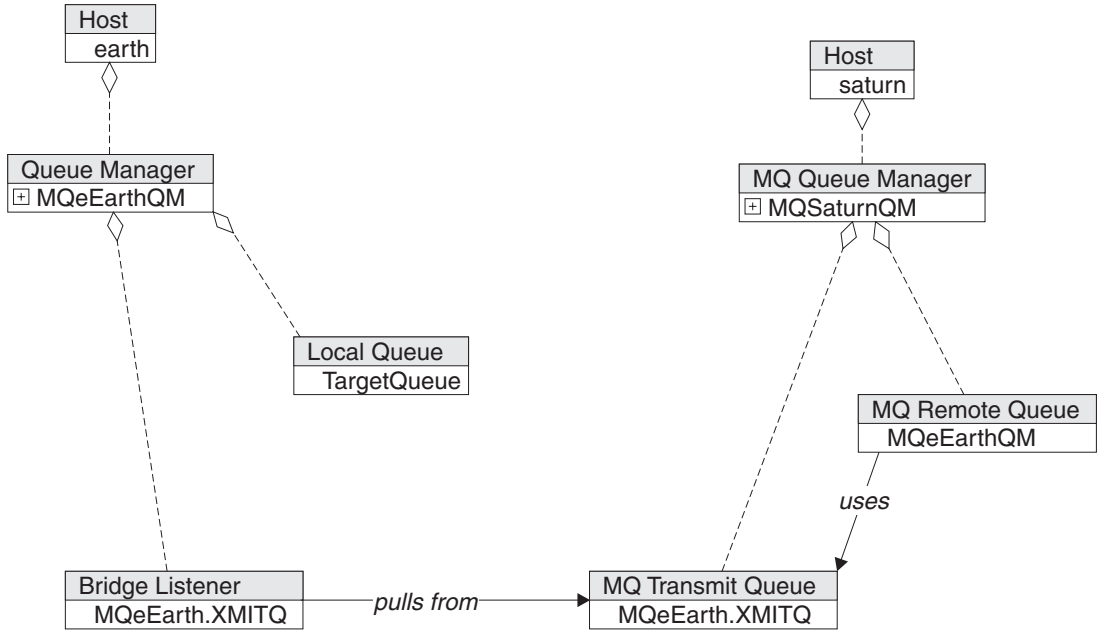

Figure 92. A multiple pull message route.

Message resolution works as previous, but now messages for any queue on MQeEarthQM will be move, making this a multiple pull message route, see Figure 93.

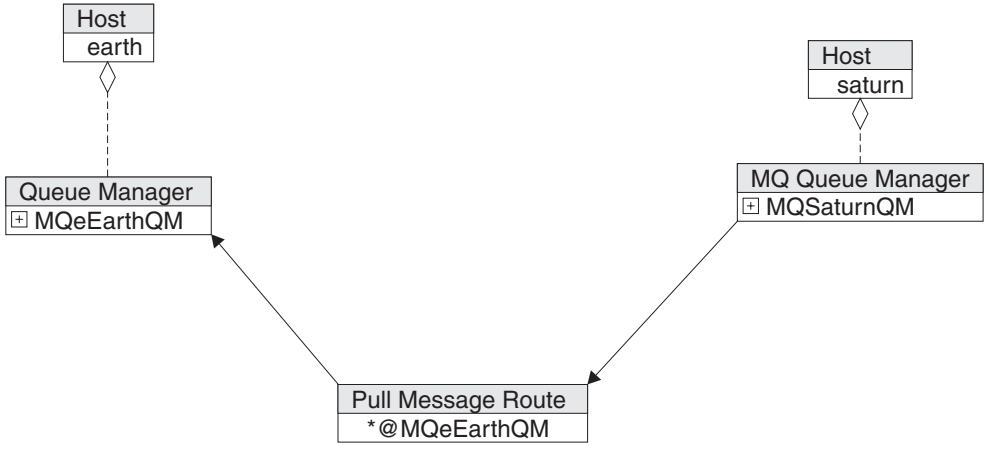

Figure 93. Multiple pull route, expressed using message route schema

Pushing messages to WebSphere MQ

Pushing messages to WebSphere MQ is more straightforward. Again we need to presume the presence of the common components described above, but now we need to create a Bridge Queue which is an WebSphere MQ Everyplace Remote queue that refers to a queue on a WebSphere MQ queue manager, see Figure 94.

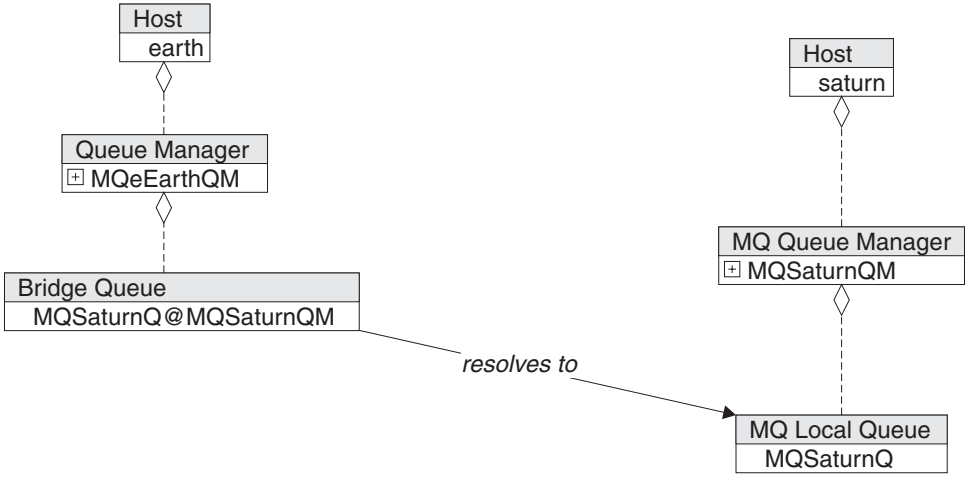

Figure 94. Pushing messages to WebSphere MQ.

Messages travel as expected across this remote queue definition, see Figure 95.

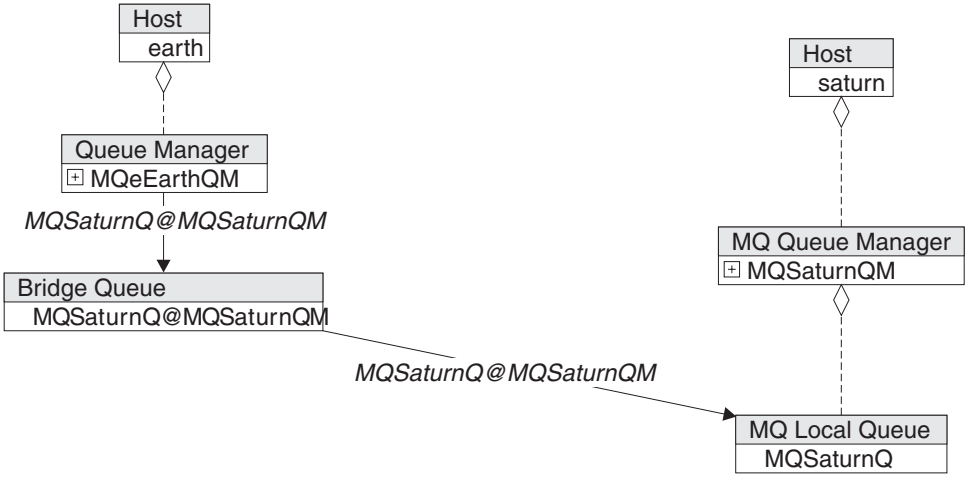

Figure 95. Messages travelling across a remote queue definition.

This is exactly the same as a simple push message route between two queue managers, see Figure 96 on page 197.

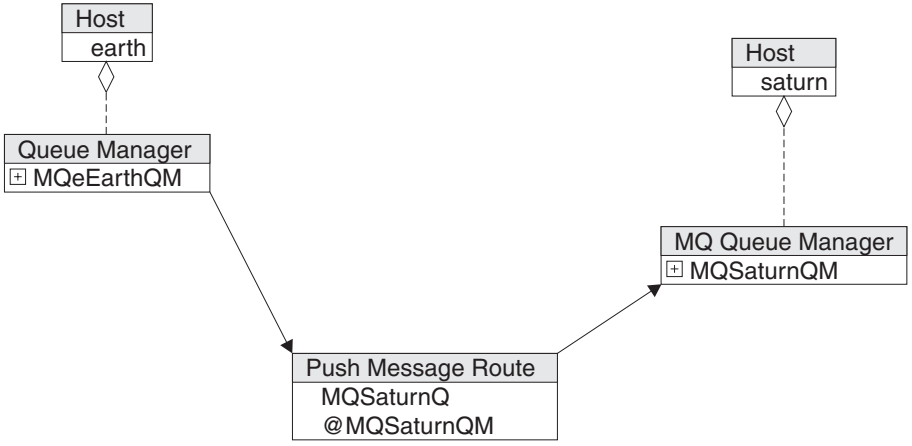

Figure 96. Simplified view of route pushing messages to WebSphere MQ

Connecting a client to WebSphere MQ via a bridge

A common topology is to allow messages to flow between WebSphere MQ and a client WebSphere MQ Everyplace queue manager. This cannot happen directly, but requires an intermediate bridge-enabled MQeQueue manager. The client can then be a small footprint device with no knowledge of WebSphere MQ. If we start from the configuration we have above, we can show the additions we will need to make to allow a client (MQeMoonQM, on a device called moon) to communicate with WebSphere MQ, see Figure 97 on page 198.

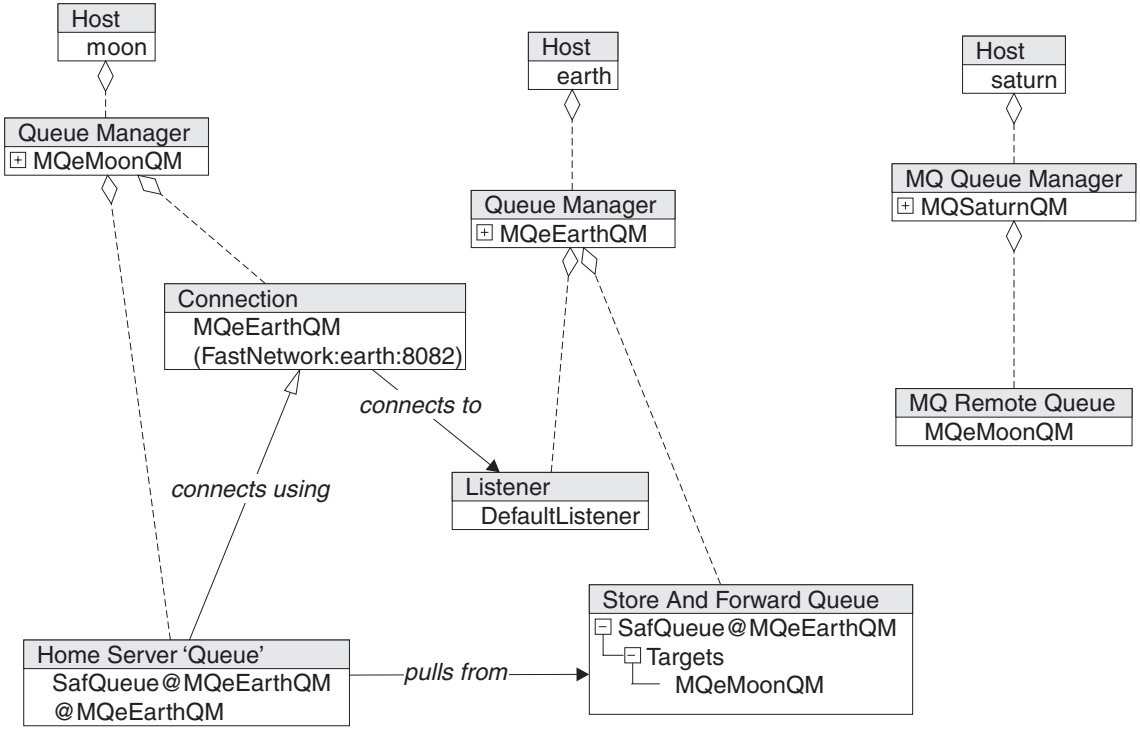

Figure 97. A client communicating with WebSphere MQ.

We have added the following:

- (Host)moon
- (QueueManager) MQeMoonQM on (Host)moon
- A connection definition from MQeMoonQM to a matching listener on MQeEarthQM to provide the connectivity between the two WebSphere MQ Everyplace queue managers.
- A store and forward queue on MQeEarthQM that will accept and hold messages for MQeMoonQM, and a home server queue on MQeMoonQM that will pull messages from the store and forward queue.
- A remote queue definition on the WebSphere MQ queue manager that will route messages for MQeMoonQM to the transmission queue MQeEarth.XMITQ. This will allow messages for MqeMoonQM to be placed on the transmission queue, from where they will be pulled to MQeEarthQM.

The topology is more readily seen as message routes, see Figure 98 on page 199.

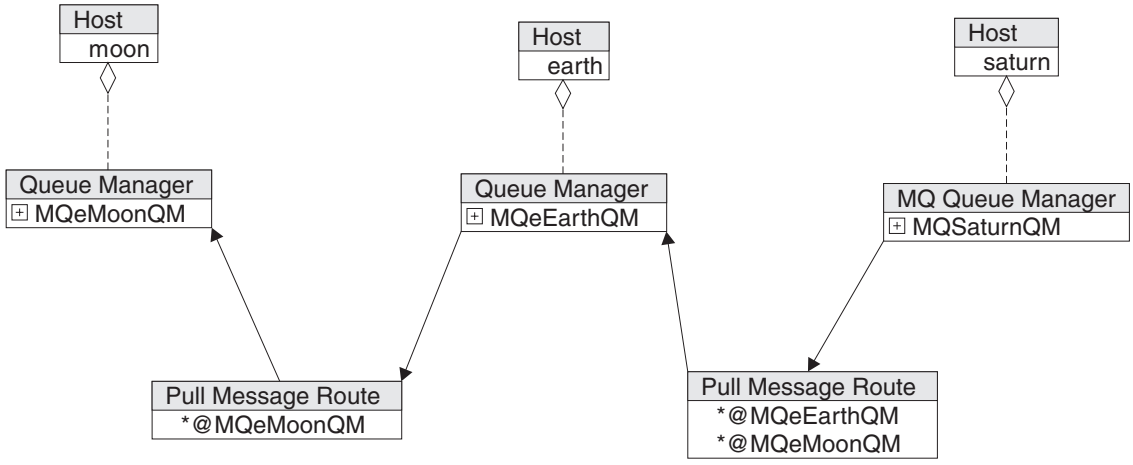

Figure 98. Simplified pull routes from WebSphere MQ through a WebSphere MQ Everyplace gateway to a WebSphere MQ Everyplace device style queue manager

Messages can be pushed to WebSphere MQ by using a via connection to chain remote queues, see Figure 99 on page 200.

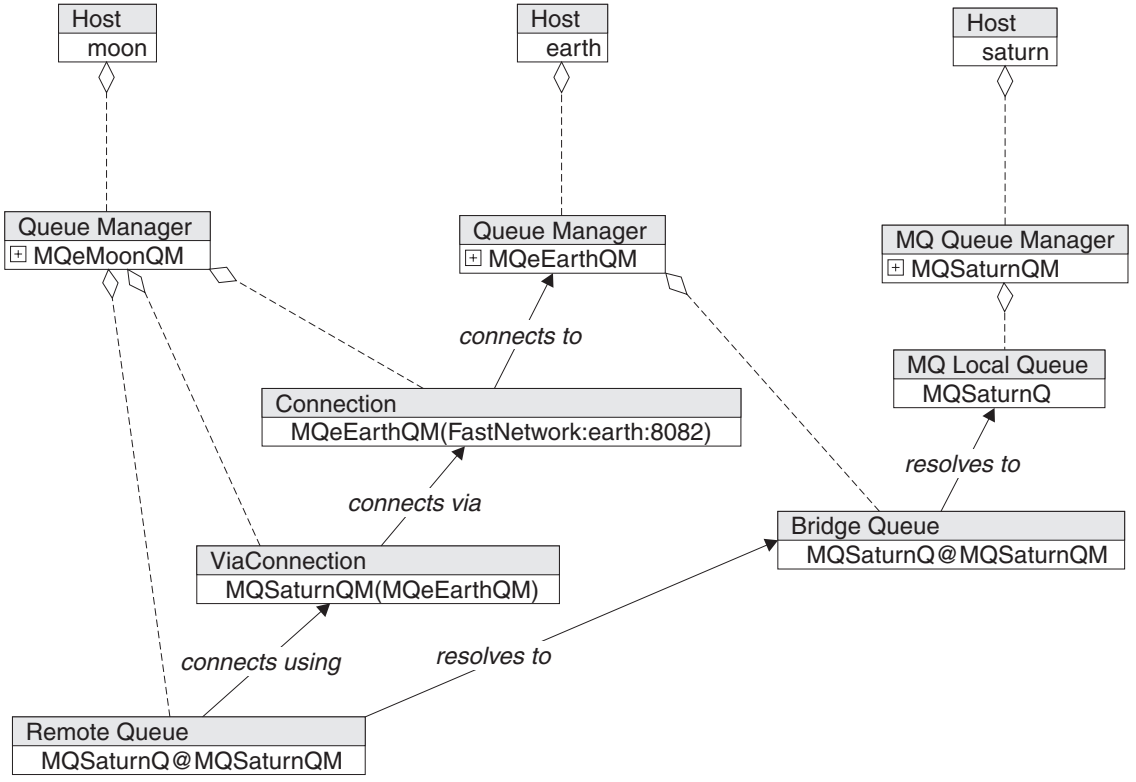

Figure 99. Pushing messages using a via connection.

Here we have added a via connection, to route messages destined for MQSaturnQM via MQeEarthQM, and we have added a remote queue definition for MQSaturnQ@MQSaturnQM. The messages can now flow from the client to WebSphere MQ, see Figure 100 on page 201.

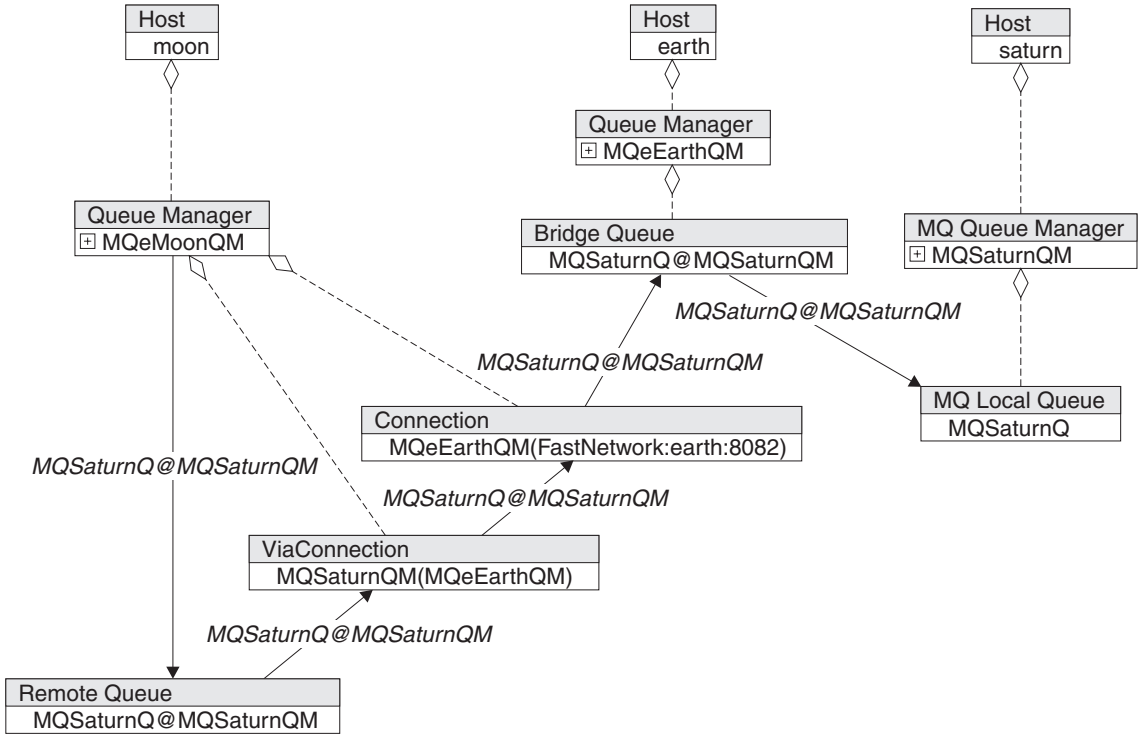

Figure 100. Pushing messages to WebSphere MQ

This topology is more easily understood as a collection of message routes, see Figure 101 on page 202.

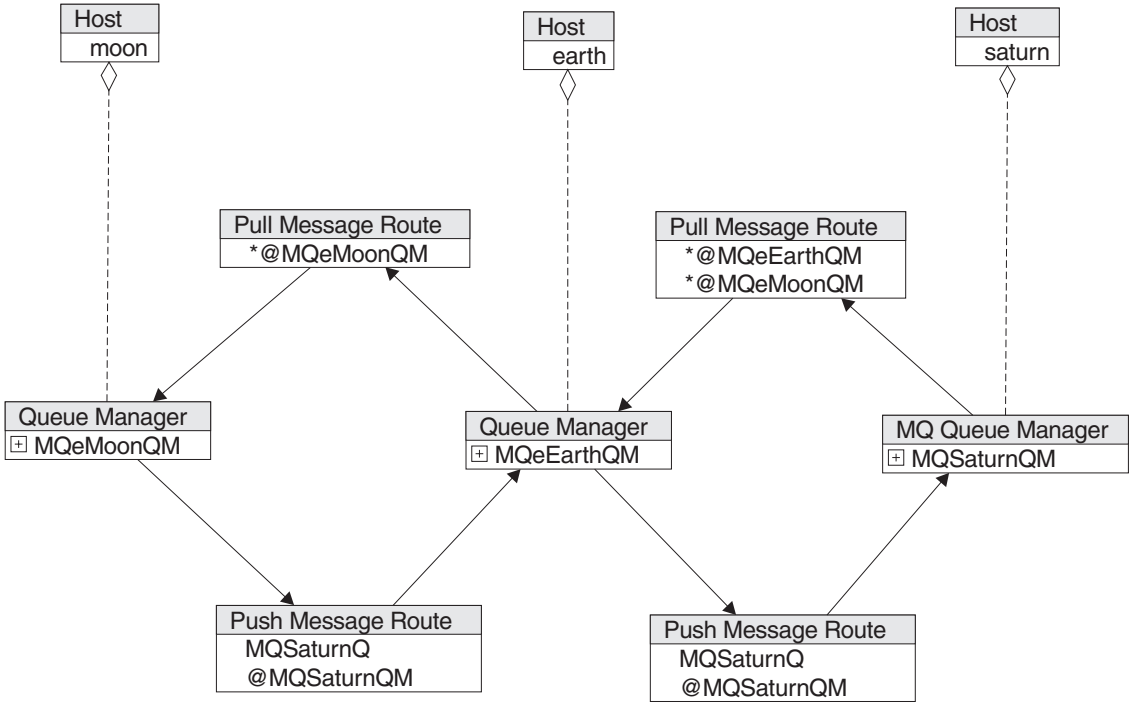

Figure 101. Simplified view showing routes which push messages from a device style WebSphere MQ Everyplace queue manager to a WebSphere MQ queue manager

Security considerations

Remote queue definitions define the security requirements that must be satisfied by channels moving messages to target queues. The queue manager attribute rule defines the rules for upgrading channels; consequently with a sufficiently flexible rule, multiple security requirements can be met by a single channel.

When a message must be stored on a queue, either en route or at the destination, then the queue attribute rule determines if the channel security meets the requirements of the queue. Note however that there are message transfers that do not involve a channel, for example, when a home server places a message it has received from a store queue on to its destination queue. In these cases there are no security requirements to be satisfied in the transfer, but the message will be stored in its destination queue in a manner controlled by that queue's security characteristics. Thus to continue this example in more depth, when the home server queue gets the message from the store queue a channel is involved (with characteristics determined by the home server queue and which must be acceptable to the store queue); however, when the home server queue passes the message to the destination queue, there are no channel characteristics to be compared with the destination queue's security characteristics.

In a single hop, message transfer, the security checking is between the source and target queue managers. In multiple hop, asynchronous message transfers, security checking occurs stepwise over each hop.

Resolution Rules

Resolution rules always start with a message being presented to a queue manager, with a specified destination queue manager name and a specified destination queue name. This is equivalent to the API call putMessage(queueManagerName, queueName, msg,....). The destinationQueueManagerName and destinationQueueName should identify a local queue onto which the message should eventually be placed.

Rule 1: Resolve queue manager aliases.

If the queue manager has an alias mapping destinationQueueManagerName to another name, for example, realQueueManagerName then this substitution is made first, and the call:

```
putMessage(destinationQueueManagerName, destinationQueueName
```

is effectively transformed to

```
putMessage(realQueueManagerName, destinationQueueName.
```

From this point on destinationQueueManagerName is completely forgotten, and realQueueManagerName is used.

Queue Resolution

The queue manager now looks for a queue to place the message on. The queue manager looks for a queue with the best match, following the these rules:

'Exact' match

Local queue or remote queue definition where the queue name matches the destinationQueueName and the queues queue manager name matches the destinationQueueManagerName.

The term 'queues queue manager name needs some explaining. For a local queue this is the same as the name of the queue manager where the queue resides. For a local queue localQ@localQM, localQM is the queues queue manager name.

For a remote queue definition remoteQ@remoteQM residing on localQM, the queues queue manager name is remoteQM.

Queue Alias Match

If a queue (remote definition or local) has a matching queue manager name and an alias and this alias matches destinationQueueName then this queue will considered a match. Effectively the put message call :

```
putMessage(destinationQueueManagerName, queueAliasName
```

is transformed to

```
putMessage(destinationQueueManagerName, realQueueName.
```

at this point. The original name of the queue used in the put call is entirely forgotten from this point on in the resolution.

S&F queue

If there is no exact match the queue manager searches for an inexact match. An inexact math is a Store and Forward queue that will accept messages for the given queue manager name. The search for a store and forward queue ignores the destinationQueueName. If an appropriate Store And Forward queue is found, then the message is put to it, using the destinationQueueManagerName and destinationQueueName, and the StoreAndForward queue stores the destination with the message.

Queue Discovery

If no queue has been found that will accept the message the and the message is not for a local queue, then the queue manager tries to find the remote destination queue and queue and create a remote queue definition for it automatically. This is called queue discovery. The queue manager can only perform discovery if:

- there is a connection definition to the destination queue manager
- there is an active communications path to the destination queue manager
- the destination queue exists
- (actually can also work under other circumstances - a via connection to a queue manager where a remote connection definition exists)

If discovery is successful the newly created remote queue definition is used. This then behaves as if an exact match on a remote queue definition had been found in the first place.

The remote queue definition created by discovery is always synchronous, even if the queue to which it resolves is asynchronous, or even a Store and forward queue.

Failure

If no queue has been found by the above steps then the message put is deemed to have failed.

Push Across Network

A message placed upon a remote queue is pushed across the network. The queue first locates a connection definition with the correct name, and then puts the message to the remote queue manager using the connection definition as the entry to the communications link.

The queue seeks a connection definition whose name is the same as the queues queue manager name. The connection may be a normal connection, or a via connection.

Normal

A normal connection points to a listener upon the destination queue manager. The put message command is routed directly to the destination queue manager. The putMessage call is then resolved just as if it had been placed on the queue manager via the API.

Via

A via connection points at another connection called the 'real' connection. All commands performed on the via connection will be delegated to the real connection. Via connections can be chained, and so the command may travel 'via' several indirections before reach a real connection. The names of the put message destination are not changed by the use of a via connection.

Eventually the command will be routed to a 'normal' connection definition, then across the network to a queue manager, where the message put will be resolved.

Home Server Pulling

Home server queues pull messages from Store and forward queues. The route of the pull only spans a single network hop. Only messages for the queue manager hosting the home server queue will be pulled down. (Q: can home server queue pull using a via connection?). Messages pulled from the store and forward queue are presented to the queue manager using a normal put method call, and are then resolved as normal. The messages pulled down this way should all be destined for local queues.

Chapter 15. Security

In addition to a basic WebSphere MQ Everyplace network, certain features can be further configured to enhance data security. Generally speaking, WebSphere MQ Everyplace provides two security mechanisms directly concerned with the transport of messages:

Message-based security

Messages are encrypted by the application, using WebSphere MQ Everyplace services, and passed to WebSphere MQ Everyplace for transport in a fully protected state. WebSphere MQ Everyplace delivers the messages to a target queue, from which they are removed by an application and subsequently decrypted, again using WebSphere MQ Everyplace services. Since the messages are fully protected when being directly handled by WebSphere MQ Everyplace, they can be flowed over clear channels and held on unprotected intermediate queues.

This security feature involves application programming and is beyond the scope of this book. Readers are referred to the WebSphere MQ Everyplace Application Programming Guide for further details.

Queue-based security

Messages are assumed to have been encrypted by the application when they are passed to WebSphere MQ Everyplace. WebSphere MQ Everyplace delivers the messages to a target queue, from which they are removed by an application. WebSphere MQ Everyplace protects the messages on receipt and flows them over secure channels; they are also held protected on any intermediate queues and on the destination queue.

This security feature does not involve application programming. As long as configurations have been set up properly, messages are automatically protected during transmission. This Chapter discusses the various configurations appropriate for queue-based security.

Queue-based security is currently only supported by the Java code base.

Background

Security properties

The level of queue-based security to be used is determined through the setting of attributes on queues. As a consequence of these attributes, WebSphere MQ Everyplace uses, if required, appropriate secure channels, and cryptors, compressors and controls access through authenticators.

The relevant queue properties are:

- Compressor: A compressor is optional. It determines whether the data should be compressed.
- Cryptor: A cryptor is optional. It determines whether the data should be encrypted to hide the significance of the contents.

- Authenticator: An authenticator is optional. It determines whether the data access should be controlled.
- Attribute rule: An attribute rule is optional in the sense that you can specify a null for this property. If a null is specified, a system default attribute rule is then used internally. An attribute rule determines whether an existing channel can be reused or upgraded to access a particular queue.

Private registries

Certain security property, such as com.ibm.mqe.attributes.MQeWTLSCertAuthenticator, prerequisite an appropriate private registry, where the entity's private/public keys can be found, and, in some cases the queue manager's public registry, where foreign entities' public keys can be found. This happens when a security attribute uses a public/private key based algorithm to perform encryption/authentication.

There are two types of private registries, queue manager owed and queue owed and each private registry only stores its owner's security credentials. The queue manager's credential, however, can be shared by the queues it owes. For this reason, if the com.ibm.mqe.attributes.MQeWTLSCertAuthenticator class authenticator is used, an additional parameter ″target registry″ on the queue attribute the authenticator is attached to must also be set. This parameter determines which registry is to supply the credentials for authentication, and can have the value of either ″Queue manager″ or ″Queue″.

If ″Queue manager″ is specified, the credentials used are those of the queue manager owning the queue, and come from the private registry of the queue manager. The queue manager originally obtains these credentials through auto-registration with the mini-certificate server (see the relevant ″Private Registry Service″ section the WebSphere MQ Everyplace Application Programming Guide for further details). This option is the recommended default.

If ″Queue″ is specified, the credentials used are those of the queue itself, and come from the private registry of the queue. The queue originally obtains these credentials through auto-registration with the mini-certificate server as well.

Please refer to the ″Mini-certificate issuance service″ in the WebSphere MQ Everyplace Application Programming Guide for issues related to mini-certificate management issues.

Effects of queue attributes

Queue attribute can be set on all queue definitions. They not only affect the way messages are stored one the queues in question but also affect the way messages are transmitted over communication channels. WebSphere MQ Everyplace creates security attributes internally based on target queue attributes. The actual effect they have depends upon the kind of queue definition the queue attributes are involved:

Local queue

Determines how the data is stored and whether the incoming channel characteristics are acceptable. If an authenticator is specified, an

authentication process using this authenticator occurs when the queue is accessed for the first time by any particular instance of a local queue manager.

Remote queue
Determines how the data is stored pending transmission, if applicable, and how the outgoing channel is established. If an authenticator is specified, an authentication process using this authenticator occurs whenever a new channel for transmitting messages on the queue is created.

Store-and-forward queue
Determines how the data is stored pending transmission, whether the incoming channel characteristics are acceptable, and how the outgoing channel is established, if applicable. An authenticator on a store-and-forward queue has the same effect that it has on a remote queue.

Home server queue
Determines how the outgoing channel is established. An authenticator on a home-server queue has the same effect that it has on a remote queue.

Communication channel security considerations

When data is sent between a queue manager and a remote queue, the queue manager opens a channel to the remote queue manager that owns the queue. By default, if the remote queue is protected, for example with a cryptor, the channel is given exactly the same level of protection as the queue. For efficiency in queue-based security, a WebSphere MQ Everyplace channel uses symmetric cryptors (for example, DES, 3DES, MARS, RC4, RC6); a consequence of which is that the two queue managers at either end must use the same encryption key. When such a channel is established, a protocol, called the Diffie Hellman key exchange, is used to establish a secret key that only the two queue managers know. This protocol is susceptible to a ″man in the middle″ attack, but for that to be successful, the ″man in the middle″ must know some of the data that is fed into the Diffie Hellman protocol. This data is held in the com.ibm.mqe.attributes.MQeDHk class. It is possible for an attacker to get hold of this data, by examining the shipped WebSphere MQ Everyplace classes. However, this data can be changed by running the com.ibm.mqe.attributes.MQeGenDH utility; it generates a new Java source file com.ibm.mqe.attributes.MQeDHk.java. This file can then be compiled into a replacement com.ibm.mqe.attributes.MQeDHk.class file. When the com.ibm.mqe.attributes.MQeWTLSCertAuthenticator is used, the two queue managers (or queues) swap certificates in order to authenticate each other. If this is used in conjunction with a cryptor on the queue, the exchanges which establish the secret key for the cryptor are protected with the public keys from the certificates, making a ″man in the middle″ attack even more difficult.

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

With asynchronous remote queues, especially Store-and-forward queues and Home-server queues, the transmitting and receiving queues are more likely to have different security attributes. These differences have to be managed during message transfer. Once a message has been put to an asynchronous queue it is transmitted

from one queue to another until it reaches its destination. A queue manager is responsible for requesting the transfer of the message between a pair of queues and another queue manager is responsible for responding to the request. If queue based security is used, the requesting queue manager establishes a channel with security attributes that match the queue that it owns. The queue manager receiving the request checks that the channel attributes are sufficient for its queue.

For example, suppose a client queue manager has a queue with a DES cryptor on it and messages are routed from this to a server's Store-and-forward queue that has a MARS cryptor. When the client is triggered to send a message it establishes a DES encrypted channel to the server; the server asks the Store-and-forward queue whether it will accept messages over a DES encrypted channel. If the Store-and-forward queue considers DES is not as strong as its own MARS cryptor (determined by the queue attribute rule), it would throw an ″attribute mismatch″ exception.

A Home-server queue trying to pull messages from a Store-and-forward queue needs a cryptor that is at least as strong as that on the Store-and-forward queue, because the Home-server queue is at the initiating end of the request. Once the Home-server queue has received the message it can store it on a local queue that has any level of protection. This behavior can be changed by using different attribute rules on the queues. For example, if the attribute rule always allows reuse, the queue will accept channels with any cryptor.

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

One slight exception to the above behavior is when a Store-and-forward queue is used to forward (push) messages to other queues. The Store-and-forward queue establishes a channel with security attributes that match its own. However, in this case the destination queue accepts the channel without checking its attributes against the queue's. For example, a Store-and-forward queue without a cryptor would establish a channel without a cryptor and this would be used to forward messages to a destination queue even if the queue had a cryptor on it. Normally, with other queue types, this would result in an ″attribute mismatch″ exception. When using a Store-and-forward queue in this way, you should ensure that it has a cryptor that is comparable to any cryptor on a destination queue. This does not apply when a Home-server queue polls for messages from a Store-and-forward queue (in this case the Home-server queue establishes the channel, not the Store-and-forward queue).

Channel attribute rules

To reduce the number of channels open concurrently, the queue manager can reuse an existing channel if its level of protection is adequate. If none of the channels has a suitable level of protection, the queue manager can also change (upgrade) the level of protection on an existing channel to match that required for the queue. This kind of behavior is goverened by the MQeattributeRule on both the queue and the channel.

These rules apply to the attribute on the queue (and channel), they are not the same as queue rules. Attribute rules are set on a queue when it is created or modified using administration messages.

The isAcceptable() method on the MQeAttributeRule class determines if a channel can be reused. This provides protection against inconsistency in the queue attribute rules on the local and target queue managers. If the isAcceptable() method returns true, the channel is used. Otherwise, the channel will not be reused.

If none of the existing channels can be reused, the queue manager checks if any of the channels can be upgraded to the required level. The permit() method on the MQeAttributeRule class determines this. If the permit() method returns true, the channel is upgraded. Otherwise, the channel is be upgraded.

WebSphere MQ Everyplace provides a default rule, com.ibm.mqe.MQeAttributeRule (identical to examples.rules.AttributeRule. This is specified as the attribute rule for a queue by WebSphere MQ Everyplace by default.

Note: This is different from setting attribute rule to null.
This rule allows a channel to be used for a queue if the following conditions are met:

1. If the queue has an authenticator, the channel must have the same type of authenticator. If the queue does not have an authenticator, it does not matter whether the channel has one or not.
2. If the queue has a cryptor, the channel must have a cryptor that is the same type as or better than that on the queue. If the queue does not have a cryptor it does not matter whether the channel has one or not. Here ″better″ is defined as:
   - Any cryptor is the same as or better than XOR.
   - Any cryptor, except XOR, is the same as or better then DES.
   - The remaining cryptors (Triple DES, RC4, RC6, and MARS) are considered equal to each other and all better than XOR and DES.
3. It does not matter what compressors are defined for the queue or channel.

This rule has the following upgrade behavior:

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

If the attribute rule is explicitly set to null, WebSphere MQ Everyplace adopts an internal rule, com.ibm.mqe.MQeAttributeDefaultRule. This rule only accepts a channel that has exactly the same authenticator (and authenticated to the same entity), cryptor, and compressor as itself for reuse and always allow channel upgrade.

Because of the way channel security works, when a specific attribute rule is specified for a target queue, it forces the local queue manager to create an instance of the same

attribute rule (examples.rules.AttributeRule and com.ibm.mqe.MQeAttributeRule are treated as the same rule for backward compatibility). Anull rule can be specified for the target queue, to avoid the need to have the same attribute rule available remotely.

While the com.ibm.mqe.MQeAttributeRule provides practical defaults, there may be a solution specific reason why different behavior is required. You can modify the way channels are reused by extending or replacing the default com.ibm.mqe.MQeAttributeRule with rules that define the desired behavior.

How to configure

This section shows how to configure a queue manager and a private registry with security features .

Setting up the queue manager

In order to configure a queue manager's private registry, which can be shared by its' queues, do the following:

1. When starting the queue manager, present the private registry logon PIN. If autoregistration with a mini-certificate server is required, the CertReqPIN, KeyRingPassword, and CAIPAddrPort parameters must also be presented, on opening the registry.
2. The mini-certificate server is running if autoregistration is required.

Setting up a private registry

A private registry is only relevant if one of the queue-attribute properties prerequisites it. In order to establish a queue manager private registry, which can be shared by its' queues, the following conditions must be met:

1. The owning queue manager must itself have a registry of type private registry.
2. The owning queue manager must have previously auto-registered with the mini-certificate server. This must have been primed to allow queue registry before the queue private registry can be established. if auto registration with a mini-certificate server is required.
3. In starting the queue manager, the queue manager private registry logon PIN, CertReqPIN, KeyRingPassword, and CAIPAddrPort were passed whilst opening the registry. If a CertReqPIN different from the queue manager's is used for the queue, it is currently necessary to first shutdown the owning queue manager, replace the original CertReqPIN with the new one, and then start the queue manager again. Auto-registration will then be triggered using the new CertReqPIN when the queue private registry is activated first time.
4. The mini-certificate server is running, if autoregistration with the mini-certificate server is required.

Refer to the WebSphere MQ Everyplace Application Programming Guide for operational details.

If queue private registry, instead of the queue manager's, is required, for example, the target registry property of the queue has been set to ″Queue″ for com.ibm.mqe.attributes.MQeWTLSCertAuthenticator.

Due to the intensity of numerical computation involved, auto-registration may take 10-20 minutes on a handheld device.

Setting up attribute properties

Security attribute properties can be added to a queue using the com.ibm.mqe.administration.MQeQueueAdminMsg class and its subclasses. The security attribute properties are defined as parameters of the administration message. The following example (examples.security.createSecureQueue) creates a new queue on an existing client queue manager. It creates the queue with a cryptor, compressor, authenticator, and attribute rule. It is not necessary to add all of these attributes and any of them could be omitted. A cryptor on a local queue uses a key seed based on the queue manager private registry logon PIN. Therefore, it is important to present the right PIN when starting the queue manager.

The example starts with a class header:

```
package examples.security;

import java.io.File;
import com.ibm.mqe.*;
import com.ibm.mqe.administration.*;
import examples.queuemanager.MQePrivateClient;

/** createSecureQueue.java
 * <p>This creates a secure queue on an existing queue manager. The queue is
 * created with an authenticator, cryptor, compressor and attribute rule.
 * The queue manager must have a private registry, so that the queue can be
 * given a cryptor.
 *
 * <p>The program requires two command line parameters.
 *
 * <p>The first parameter is a configuration file for the queue manager. This
 * is used to start the queue manager as a client.
 *
 * <p>The second parameter is the PIN for the queue manager's private
 * registry.
 *
 **/

public class createSecureQueue
{
```

First we define the name of the queue we want to add:

```
  // the name of the queue
  String qName = "protQueue";
```

The attributes are defined by their class names:

```
// define the attributes we want the queue to have. These are defined by
// their class names.
String cryptorType       = "com.ibm.mqe.attributes.MQeDESCryptor";
String compressorType    = "com.ibm.mqe.attributes.MQeGZIPCompressor";
String authenticatorType = "examples.attributes.NTAuthenticator";
String attributeRule     = "com.ibm.mqe.MQeAttributeRule";
```

They are followed by some definitions of local variables:

```
//local variables
MQePrivateClient client;
MQeQueueManager  clientQM;
String           clientQMName;
MQeQueueAdminMsg msg;
```

The example adds the queue directly to the local queue manager, so the queue manager must be activated:

```
/**
 * open the queue manager as a client
 *
 * @param configFile the configuration (.ini) file for the queue manager
 * @param qmPIN      the PIN for the queue manager's registry
 * @exception  java.lang.Exception  propagated from invoked methods
 **/
void openQM(String configFile, String qmPIN) throws Exception
{
  // start the queue manager as a client
  client = new MQePrivateClient(configFile, qmPIN, null, null);

  //save the queue manager and its name
  clientQM = client.queueManager;
  clientQMName = clientQM.getName();
}
```

The MQeQueueAdminMsg is created and values added to it as normal. A correlation id is added to the message to make it easy to find the reply message. All the security attributes are added as parameters to the message, that is, they are added to a separate MQeFields object which is passed to the msg.create(parms) method:

```
/**
 * create the admin message to add the queue attributes
 *
 * @exception  java.lang.Exception  propagated from invoked methods
 **/
void createAdminMsg() throws Exception
{
  // the file descriptor
  String FileDesc = "MsgLog:.";

  // create an Admin msg to add the queue
  msg = new MQeQueueAdminMsg();
  msg.setTargetQMgr(clientQMName);
  msg.setName(clientQMName, qName);
  msg.putInt(MQe.Msg_Style, MQe.Msg_Style_Request);
  msg.putAscii(MQe.Msg_ReplyToQ, MQe.Admin_Reply_Queue_Name);
```

```
    msg.putAscii(MQe.Msg_ReplyToQMgr, clientQMName);
    msg.putArrayOfByte(MQe.Msg_CorrelID,
    Long.toHexString(clientQM. uniqueValue()).getBytes());

    // define parameter values for the queue
    MQeFields parms = new MQeFields();
    parms.putUnicode(msg.Queue_Description, "DES protected queue");
    parms.putAscii(msg.Queue_FileDesc, FileDesc );

    // this is where we specify the queue attributes
    parms.putAscii(msg.Queue_Cryptor, cryptorType);
    parms.putAscii(msg.Queue_Compressor, compressorType);
    parms.putAscii(msg.Queue_Authenticator, authenticatorType);
    parms.putAscii(msg.Queue_AttrRule, attributeRule);

    //add the parameters to the message
    msg.create(parms);
  }
```

The message is sent to the Admin Queue on the local queue manager:

```
  /**
   * send the admin message to the client queue manager
   *
   * @exception  java.lang.Exception  propagated from invoked methods
   **/
  void sendAdminMsg() throws Exception
{
    // send the Admin msg
    System.out.println("putting Admin Msg to QM/queue:" +
    clientQMName + "/" + MQe.Admin_Queue_Name);
    clientQM.putMessage(clientQMName, MQe.Admin_Queue_Name, msg, null, 0);
  }
```

The correlation id is used in a filter to find the correct reply. The example waits up to 3 seconds for the reply:

```
  /**
   * wait for a reply message and process it to determine success or failure
   *
   * @exception  java.lang.Exception  propagated from invoked methods
   **/
  void processReply() throws Exception
    {
    // use the CorrelID to create a filter for the reply message
    MQeFields replyFilter = new MQeFields();
    replyFilter.putArrayOfByte(MQe.Msg_CorrelID,
     msg.getArrayOfByte(MQe.Msg_CorrelID));

    // get the Admin Reply msg
    MQeMsgObject reply = clientQM.waitForMessage(clientQMName,
                MQe.Admin_Reply_Queue_Name,
                replyFilter,
                null,
                0,
3000);
```

```
    if (reply instanceof MQeAdminMsg)
    {
      MQeAdminMsg adminReply = (MQeAdminMsg)reply;
      System.out.println("Admin Reply Msg received");
      if (adminReply.getRC() == MQeAdminMsg.RC_Success)
        System.out.println("Queue added OK");
      else
System.out.println("create Queue failed:" +
    adminReply.getReason());
    }
    else
      System.out.println("reply message is not an admin message");
  }
```

The queue manager needs to be closed:

```
  /**
   * close the queue manager
   *
   * @exception  java.lang.Exception  propagated from invoked method
   **/
  void close() throws Exception
  {
    clientQM.close();
  }
```

The main() method for the example is:

```
  /**
   * main method.
   *
   * @param  args  The command line arguments. The first is a configuration
   *               (.ini) file for the queue manager, the second is the PIN
   *               for the queue manager's private registry.
   *
   **/
  public static void main(String [] args)
  {
    createSecureQueue secQueue = new createSecureQueue();
    // check the command line arguments
    if (args.length < 2)
      System.err.println("usage: createSecureQueue configFile qmPIN");
    else
    {
      try
      {
        secQueue.openQM(args[0], args[1]);
        secQueue.createAdminMsg();
        secQueue.sendAdminMsg();
        secQueue.processReply();
        secQueue.close();
      }
      catch (Exception e)
      {
        System.out.println("Exception caught:" + e);
```

```
        }
      }
    }
}
```

Attribute rules can also be set on channels using the ChannelAttrRules keyword in the configuration file used at queue manager creation time. WebSphere MQ Everyplace defaults the keyword to com.ibm.mqe.MQeAttrubuteRule.

Chapter 16. Java Message Service (JMS) configuration

For JMS applications to be portable, they must be isolated from the administration of the underlying messaging provider. This is achieved by defining JMS 'administered objects' which encapsulate provider-specific information. Administered objects are created and configured using provider-specific facilities, but are used by clients through portable JMS interfaces.

There are two types of JMS administered object:

- A ConnectionFactory, used by a client to create a connection with a provider.
- A Destination, used by a client to specify the destination of messages it is sending and the source of messages that it receives.

In WebSphere MQ Everyplace JMS these correspond to two classes:

- MQeQueueConnectionFactory must be configured so that it can obtain a reference to a WebSphere MQ Everyplace queue manager.
- MQeJMSQueue can be configured with details of a WebSphere MQ Everyplace queue.

These classes are typically placed in a JNDI namespace by an administrator. However, on small devices access to a JNDI namespace may be impractical or may represent an unnecessary overhead, so these classes do not include the necessary methods to allow them to be bound by JNDI. Two subclasses, MQeJNDIQueueConnectionFactory and MQeJMSJNDIQueue extend these classes to allow them to be stored using JNDI.

Configuring MQeQueueConnectionFactory

MQeQueueConnectionFactory is the WebSphere MQ Everyplace implementation of the javax.jms.QueueConnectionFactory interface. It is used to generate instances of QueueConnection classes, which for WebSphere MQ Everyplace must have a reference to an active queue manager. The QueueConnectionFactory must be able to create a reference to an active queue manager in order to pass it on to the QueueConnection classes that it generates. The MQeQueueConnectionFactory class can be configured to obtain a reference to a queue manager in the following ways:

- It can start a client queue manager itself.
- It can look for a queue manager already running in the JVM.

However, if neither of these options are suitable then the MQeQueueConnectionFactory class can be extended to provide the required behavior. This is discussed later in this chapter.

To configure a connection factory to start a queue manager itself, it must be given a reference to an initialization (.ini) file that contains all the information it needs to start the queue manager. The connection factory is configured using its setIniFileName() method:

```
(MQeQueueConnectionFactory(factory)).setIniFileName(filename);
```

where 'filename' is the name of the initialization file. When the connection factory has been configured with the name of the initialization file, it can either be stored in a JNDI directory, so that it can be looked up by application programs, or it can be used directly in an application program. When the connection factory generates its first QueueConnection it starts the client queue manager using the initialization file and passes a reference to the active queue manager to the QueueConnection. If it generates more QueueConnection classes, it passes them a reference to the same active queue manager. When the last QueueConnection is closed, the connection factory closes the queue manager.

Note: Do not use the MQeQueueManager.close() methods to shut down a queue manager started by a connection factory.

To configure a connection factory to look for an existing queue manager, the initialization file name should be set to null. This is the default value when the MQeQueueConnectionFactory class is created, and it can also be set explicitly using the setIniFileName() method:

```
(MQeQueueConnectionFactory(factory)).setIniFileName(null);
```

In this case, when the connection factory generates a QueueConnection, it looks for a queue manager already running in the JVM and passes the QueueConnection a reference to it. An exception is thrown if no queue manager is running. If it generates more QueueConnection classes, it passes them a reference to the same queue manager. The connection factory does not close the queue manager when the last QueueConnection is closed.

Note: A JVM can run only one WebSphere MQ Everyplace queue manager at a time. Therefore, if you use a connection factory to start a queue manager, it should not be used to start the same queue manager in a different JVM, running on the same machine, while the first one is still active.

Configuring MQeJMSQueue

MQeJMSQueue is the WebSphere MQ Everyplace implementation of the Queue class. It is used to represent WebSphere MQ Everyplace queues within JMS applications. It is configured by its constructor:

```
public MQeJMSQueue(String mqeQMgrName, String mqeQueueName) throws JMSException
```

where:

- mqeQMgrName is the name of the WebSphere MQ Everyplace queue manager which owns the queue
- mqeQueueName is the name of the WebSphere MQ Everyplace queue

If the queue manager name is null, the local queue manager is used (that is, the queue manager that JMS is connected to). If the queue name is null, a JMSException is thrown.

When the queue has been configured, it can either be stored in a JNDI directory, so that it can be looked up by application programs, or it can be used directly in an application program. There is an alternative way to configure a queue within an application, by using the QueueSession.createQueue() method. This takes one parameter, which is the name of the queue. For WebSphere MQ Everyplace JMS this can either be the queue manager name followed by a plus sign followed by the queue name:

```
ioQueue =session.createQueue("myQM+myQueue");
```

or just the queue name:

```
ioQueue =session.createQueue("myQueue");
```

If the queue name is used on its own, the local queue manager is assumed.

Note: WebSphere MQ Everyplace JMS can only put messages to a local queue or an asynchronous remote queue and it can only receive messages from a local queue. It cannot put to or receive messages from a synchronous remote queue.

The JMS administration tool

The administration tool provides a simple way for administrators to define and edit the properties of WebSphere MQ Everyplace JMS administered objects. This tool is based on the administration tool shipped with JMS for WebSphere MQ, differing only in the properties that can be applied to JMS administered objects.

Configuration

You must configure the administration tool with values for the following three parameters:

INITIAL_CONTEXT_FACTORY

This indicates the service provider that the tool uses. There are currently two supported values for this property:

- com.sun.jndi.ldap.LdapCtxFactory (for LDAP)
- com.sun.jndi.fscontext.RefFSContextFactory (for file system context)

PROVIDER_URL

This indicates the URL of the session's initial context, the root of all JNDI operations carried out by the tool. Two forms of this property are currently supported:

- ldap://hostname/contextname (for LDAP)
- file:[drive:]/pathname (for file system context)

SECURITY_AUTHENTICATION

This indicates whether JNDI passes over security credentials to your service provider. This parameter is used only when an LDAP service provider is used. This property can currently take one of three values:

- none (anonymous authentication)
- simple (simple authentication)

- CRAM-MD5 (CRAM-MD5 authentication mechanism)

If a valid value is not supplied, the property defaults to none. If the parameter is set to either simple or CRAM-MD5, security credentials are passed through JNDI to the underlying service provider. These security credentials are in the form of a user distinguished name (User DN) and password. If security credentials are required, then the user will be prompted for these when the tool initializes.

Note: The text typed is echoed to the screen, and this includes the password. Therefore, take care that passwords are not disclosed to unauthorized users.

These parameters are set in a plaintext configuration file consisting of a set of key-value pairs, separated by an ″=″. This is shown in the following example:

```
#Set the service provider
INITIAL_CONTEXT_FACTORY=com.sun.jndi.ldap.LdapCtxFactory
#Set the initial context
PROVIDER_URL=ldap://polaris/o=ibm_us,c=us
#Set the authentication type
SECURITY_AUTHENTICATION=none
```

(A ″#″ in the first column of the line indicates a comment, or a line that is not used.)

Starting the JMS admin tool

To start the tool in interactive mode, enter the command:

```
java com.ibm.mqe.jms.admin.MQeJMSAdmin [-cfg config_filename]
```

where the -cfg option specifies the name of an alternative configuration file. If no configuration file is specified, then the tool looks for a file named MQeJMSAdmin.config in the current directory.

After authentication, if necessary, the tool displays a command prompt:

```
InitCtx>
```

indicating that the tool is using the initial context defined in the PROVIDER_URL configuration parameter.

To start the tool in batch mode, enter the command:

```
java com.ibm.mqe.jms.admin.MQeJMSAdmin < script.scp
```

where script.scp is a script file that contains administration commands. The last command in this file must be an END command.

Administration commands

When the command prompt is displayed, the tool is ready to accept commands. Administration commands are generally of the following form:

```
verb [param ]*
```

where verb is one of the administration verbs listed in Table xxx. All valid commands consist of at least one (and only one) verb, which appears at the beginning of the command in either its standard or short form.

The parameters a verb may take depend on the verb. For example, the END verb cannot take any parameters, but the DEFINE verb may take anything between 1 and 20 parameters. Details of the verbs that take at least one parameter are discussed later in this section.

Table 25. Administration verbs

Verb	Short form	Description
ALTER	ALT	Change at least one of the properties of a given administered object
DEFINE	DEF	Create and store an administered object, or create a new subcontext
DISPLAY	DIS	Display the properties of one or more stored administered objects, or the contents of the current context
DELETE	DEL	Remove one or more administered objects from the namespace, or remove an empty subcontext
CHANGE	CHG	Alter the current context, allowing the user to traverse the directory namespace anywhere below the initial context (pending security clearance)
COPY	CP	Make a copy of a stored administered object, storing it under an alternative name
MOVE	MV	Alter the name under which an administered object is stored
END		Close the administration tool

Verb names are not case-sensitive.

Usually, to terminate commands, you press the carriage return key. However, you can override this by typing the ″+″ symbol directly before the carriage return. This enables you to enter multi-line commands, as shown in the following example:

```
DEFINE Q(BookingsInputQueue)+
QMGR(ExampleQM)+
QUEUE(QUEUE.BOOKINGS)
```

Lines beginning with one of the characters *, #, or / are treated as comments.

Manipulating subcontexts

You can use the verbs CHANGE , DEFINE , DISPLAY and DELETE to manipulate directory namespace subcontexts. Their use is described in Table xxx.

Table 26. Syntax and description of commands used to manipulate subcontexts

Command syntax	Description
DEFINE CTX(ctxName)	Attempts to create a new child subcontext of the current context, having the name ctxName. Fails if there is a security violation, if the subcontext already exists, or if the name supplied is invalid.
DISPLAY CTX	Displays the contents of the current context. Administered objects are annotated with a 'a', subcontexts with '[D]'. The Java type of each object is also displayed.
DELETE CTX(ctxName)	Attempts to delete the current context's child context having the name ctxName. Fails if the context is not found, is non-empty, or if there is a security violation.
CHANGE CTX(ctxName)	Alters the current context, so that it now refers to the child context having the name ctxName. One of two special values of ctxName may be supplied: **=UP** which moves to the current context's parent **=INIT** which moves directly to the initial context Fails if the specified context does not exist, or if there is a security violation.

Administering JMS objects

Two object types can currently be manipulated by the administration tool. These are listed in table xxx.

Table 27. JMS administered objects

Object type	Keyword	Description
MQeJNDIQueueConnectionFactory	QCF	The WebSphere MQ Everyplace implementation of the JMS QueueConnectionFactory interface. This represents a factory object for creating connections in the JMS Point-to-Point messaging domain.

Table 27. JMS administered objects (continued)

Object type	Keyword	Description
MQeJMSJNDIQueue	Q	The WebSphere MQ Everyplace implementation of the JMS Queue interface. This represents a message Destination in the JMS Point-to-Point messaging domain.

Verbs used with JMS objects

You can use the verbs ALTER, DEFINE, DISPLAY, DELETE, COPY and MOVE to manipulate administered objects in the directory namespace. Table xxx summarizes their use. Substitute TYPE with the keyword that represents the required administered object, as listed in Table ***the previous table***.

Table 28. Syntax and description of commands used to manipulate administered objects

Command syntax	Description
ALTER TYPE(name) [property]*	Attempts to update the given administered object's properties with the ones supplied. Fails if there is a security violation, if the specified object cannot be found, or if the new properties supplied are invalid.
DEFINE TYPE(name) [property]*	Attempts to create an administered object of type TYPE with the supplied properties, and tries to store it under the name name in the current context. Fails if there is a security violation, if the supplied name is invalid or already exists, or if the properties supplied are invalid.
DISPLAY TYPE(name)	Displays the properties of the administered object of type TYPE , bound under the name name in the current context. Fails if the object does not exist, or if there is a security violation.
DELETE TYPE(name)	Attempts to remove the administered object of type TYPE, having the name name, from the current context. Fails if the object does not exist, or if there is a security violation.
COPY TYPE(nameA) TYPE(nameB)	Makes a copy of the administered object of type TYPE, having the name nameA, naming the copy nameB. This all occurs within the scope of the current context. Fails if the object to be copied does not exist, if an object of name nameB already exists, or if there is a security violation.

Table 28. Syntax and description of commands used to manipulate administered objects (continued)

Command syntax	Description
MOVE TYPE(nameA) TYPE(nameB)	Moves (renames) the administered object of type TYPE, having the name nameA , to nameB . This all occurs within the scope of the current context. Fails if the object to be moved does not exist, if an object of name nameB already exists, or if there is a security violation.

Creating objects

Objects are created and stored in a JNDI namespace using the following command syntax:

```
DEFINE TYPE (name)[property ]*
```

That is, the DEFINE verb, followed by a TYPE (name) administered object reference, followed by zero or more properties.

LDAP naming considerations

To store your objects in an LDAP environment, their names must comply with certain conventions. One of these is that object and subcontext names must include a prefix, such as cn=(common name), or ou=(organizational unit). The administration tool simplifies the use of LDAP service providers by allowing you to refer to object and context names without a prefix. If you do not supply a prefix, the tool automatically adds a default prefix (currently cn=) to the name you supply.

This is shown in the following example.

```
InitCtx>DEFINE Q(testQueue)
InitCtx>DISPLAY CTX
Contents of InitCtx

          a cn=testQueue com.ibm.mqe.jms.MQeJMSJNDIQueue

1 Object(s)
0 Context(s)
1 Binding(s),1 Administered
```

Note that although the object name supplied does not have a prefix, the tool automatically adds one to ensure compliance with the LDAP naming convention. Likewise, submitting the command DISPLAY Q(testQueue) also causes this prefix to be added.

You may need to configure your LDAP server to store Java objects. Information to assist with this configuration is provided later in this Chapter.

Properties

A property consists of a name-value pair in the format:

```
PROPERTY_NAME(property_value)
```

Names are not case sensitive, but are restricted to a set of recognized names shown in table xxx.

Table 29. Property names and valid values

Property	Short form	Valid values
CLIENTIDCID		Any String
DESCRIPTION	DESC	Any String
DUPSOKCOUNT	DOC	Any positive integer
INIFILE	INI	Any String
QUEUE	QU	Any String
QMANAGER	QMGR	Any String
LOGGERURL	URL	Any String

Most of these properties only apply to specific object types. These are listed below, along with a short description.

Table 30.

Property	QCF	Q	Description
CLIENTID	Y		A string identifier for the client
DESCRIPTION	Y	Y	A description of the stored object
DUPSOKCOUNT	Y		The number of messages to receive before acknowledgment in a DUPS_OK_ACKNOWLEDGE Session.
INIFILE	Y		An initialization (.ini) file for a WebSphere MQ Everyplace Queue Manager
QUEUE		Y	The name of an WebSphere MQ Everyplace queue
QMANAGER		Y	The name of an WebSphere MQ Everyplace queue manager

Table 30. (continued)

Property	QCF	Q	Description
LOGGERURL	Y		A URL to be passed to the JMS transaction logger, of the format file://<path>, defining where the transaction log should be located.

Extending MQeQueueConnectionFactory

By default MQeQueueConnectionFactory will either look for a queue manager already running in the JVM, or will start its own using an initialization (.ini) file. A third option is to extend MQeQueueConnectionFactory to provide the desired behavior. The preferred way to do this is to override two internal methods, startQueueManager() and stopQueueManager(). The first method is called to start and configure a WebSphere MQ Everyplace queue manager when a QueueConnection is first created, while the second shuts it down cleanly when the final QueueConnection is closed. These methods are both public to make them easy to override, but they should not normally be called by an application.

The following class shows a simple way of extending MQeQueueConnectionFactory to start its own queue manager without the need for an initialization file:

```
import javax.jms.*;
import examples.config.*;
import com.ibm.mqe.jms.MQeQueueConnectionFactory;
import com.ibm.mqe.MQeQueueManager;
import java.io.File;

// type on one line
public class MQeExtendedQueueConnectionFactory
   extends MQeQueueConnectionFactory {

private static final String queueManagerName = "ExampleQM";
 // Queue Manager Name
   private static final String registryLocation = ".\\ExampleQM";
 // Location of the registry
   private static final String queueStore = "MsgLog:" +
registryLocation + File.separator + "Queues";
 // Queue store
private static MQeQueueManager queueManager = null;
// the WebSphere MQ Everyplace Queue Manager

public MQeQueueManager startQueueManager() throws JMSException {
       try {
CreateQueueManager.createQueueManagerDefinition(
queueManagerName, registryLocation, queueStore);
queueManager=CreateQueueManager.startQueueManager(
queueManagerName, registryLocation);
}
catch (Exception e) {
```

```
JMSException je = new JMSException("QMgr start  failed");
          je.setLinkedException(e);
          throw je;
       }
     return queueManager;
    }

   public void stopQueueManager() throws Exception {
       CreateQueueManager.stopQueueManager(queueManager);
    }
}
}
```

In this example the actual queue manager startup and shutdown has been delegated to the CreateQueueManager examples described in an earlier chapter.

LDAP schema definition for storing Java objects

This section gives details of the schema definitions (attribute and objectClass definitions) needed in an LDAP directory in order for it to store Java objects. These are required if you wish to use an LDAP server as your JNDI service provider for storing WebSphere MQ Everyplace JMS administered objects.

Some servers may already contain these definitions in their schema. The exact procedure to check whether your server contains them, and to add them if they are not there, will vary from server to server. Please read the documentation that comes with your LDAP server and your LDAP JNDI service provider.

Much of the data contained in this section has been taken from RFC 2713 Schema for Representing Java Objects in an LDAP Directory, which can be found at http://www.faqs.org/rfcs/rfc2713.html. Please note that some LDAP servers may require you to turn off schema checking, even after these definitions have been added.

Attribute definitions

Table 31. Attribute settings for javaCodebase

Attribute	Value
OID (Object Identifier)	1.3.6.1.4.1.42.2.27.4.1.7
Syntax	IA5 String (1.3.6.1.4.1.1466.115.121.1.26)
Maximum length	2,048
Single/multi-valued	Multi-valued
User modifiable?	Yes
Matching rules	caseExactIA5match
Access class	Normal
Usage	userApplications
Description	URL(s) specifying the location of class definition

Table 32. Attribute settings for javaClassName

Attribute	Value
OID (Object Identifier)	1.3.6.1.4.1.42.2.27.4.1.6
Syntax	Directory String (1.3.6.1.4.1.1466.115.121.1.15)
Maximum length	2,048
Single/multi-valued	Single-valued
User modifiable?	Yes
Matching rules	caseExactMatch
Access class	Normal
Usage	userApplications
Description	Fully qualified name of distinguished Java class or interface

Table 33. Attribute settings for javaClassNames

Attribute	Value
OID (Object Identifier)	1.3.6.1.4.1.42.2.27.4.1.13
Syntax	Directory String (1.3.6.1.4.1.1466.115.121.1.15)
Maximum length	2,048
Single/multi-valued	Multi-valued
User modifiable?	Yes
Matching rules	caseExactMatch
Access class	Normal
Usage	userApplications
Description	Fully qualified Java class or interface name

Table 34. Attribute settings for javaFactory

Attribute	Value
OID (Object Identifier)	1.3.6.1.4.1.42.2.27.4.1.10
Syntax	Directory String (1.3.6.1.4.1.1466.115.121.1.15)
Maximum length	2,048
Single/multi-valued	Single-valued
User modifiable?	Yes
Matching rules	caseExactMatch
Access class	Normal
Usage	userApplications
Description	Fully qualified Java class name of a JNDI object Factory

Table 35. Attribute settings for javaReferenceAddress

Attribute	Value
OID (Object Identifier)	1.3.6.1.4.1.42.2.27.4.1.11
Syntax	Directory String (1.3.6.1.4.1.1466.115.121.1.15)
Maximum length	2,048
Single/multi-valued	Multi-valued
User modifiable?	Yes
Matching rules	caseExactMatch
Access class	Normal
Usage	userApplications
Description	Addresses associated with a JNDI Reference

Table 36. Attribute settings for javaSerializedData

Attribute	Value
OID (Object Identifier)	1.3.6.1.4.1.42.2.27.4.1.8
Syntax	Octet String (1.3.6.1.4.1.1466.115.121.1.40)
Single/multi-valued	Single-valued
User modifiable?	Yes
Access class	Normal
Usage	userApplications
Description	Serialized form of a Java object

objectClass definitions

Table 37. objectClass definition for javaSerializedObject

Definition	Value
OID (Object Identifier)	1.3.6.1.4.1.42.2.27.4.2.5
Extends/superior	javaObject
Type	AUXILIARY
Required attributes	javaSerializedData

Table 38. objectClass definition for javaObject

Definition	Value
OID (Object Identifier)	1.3.6.1.4.1.42.2.27.4.2.4
Extends/superior	Top
Type	ABSTRACT
Required attributes	javaClassName
Optional attributes	javaClassNames, javaCodebase, javaDoc description

Table 39. objectClass definition for javaContainer

Definition	Value
OID (Object Identifier)	1.3.6.1.4.1.42.2.27.4.2.1
Extends/superior	Top
Type	STRUCTURAL
Required attributes	cn

Table 40. objectClass definition for javaNamingReference

Definition	Value
OID (Object Identifier)	1.3.6.1.4.1.42.2.27.4.2.7
Extends/superior	javaObject
Type	AUXILIARY
Optional attributes	attrs javaReferenceAddress javaFactory

Chapter 17. Packaging and deployment

WebSphere MQ Everyplace is a flexible messaging system that can be deployed to a wide variety of operating systems and devices. This chapter provides information to assist in the build, packaging and deployment of WebSphere MQ Everyplace. It is split into two sections covering the Java code base and the native code base. As WebSphere MQ Everyplace can be deployed on a variety of devices, operating systems, and runtimes, it is not possible to detail each application. Therefore, regarding some topics, only a brief outline and introduction is provided. For further information on any particular topic refer to the corresponding documentation.

Java code base

The WebSphere MQ Everyplace Java code base can be deployed onto a large variety of Java runtimes. These include:

- J2ME CLDC/MIDP
- J2ME CDC/Foundation
- PersonalJava V1.1
- Java 1.1
- J2SE 1.2 (or later)
- IBM® WebSphere Studio Custom Environment (WSCE) jclGateway (or better)

The way that WebSphere MQ Everyplace, the application and other classes are packaged and deployed is dependant on the type of Java runtime, the operating system and processor type of the device that is being deployed to. This section provides information to assist in packaging and deploying Java based WebSphere MQ Everyplace applications to different environments.

Supplied jar files

WebSphere MQ Everyplace is supplied with a set of class libraries in the form of jar files that can be used when deploying applications that utilize WebSphere MQ Everyplace. There are two types of jar file; base jar files and extension jar files. The base jar files allow a usable queue manager to be created, administered and run. The extension jar files can be used in addition to the base jar files to provide additional capability.

Base jar files

MQeBase.jar

Contains classes that provide for a basic queue manager running in client and server mode on a J2ME CDC/Foundation or J2SE or better Java runtime.

MQeMidp.jar

Similar to MQeBase.jar but for use on a J2ME CLDC/MIDP Java runtime. Allows a queue manager to run in client mode. All MIDP compliant classes are included in this jar. No extension jars can be used with this one, as they are not MIDP compliant.

MQeGateway.jar
Contains classes that provide for a basic queue manager running in client, server and bridge mode on a J2SE or better Java runtime.

Extension jar files

MQeJMS.jar
Contains the classes that extend an WebSphere MQ Everyplace queue manager to provide a JMS programming interface.

MQeRetail.jar
Contains extra classes for use in retail environments. In particular, these classes are useful on a 4690 retail system.

MQeSecurity.jar
A set of classes that are used to provide both queue and message based security. It contains a set of cryptors, compressors and authenticators.

MQeBindings.jar
This file contains all C bindings specific information. It is required if access to a Java queue manager from a C application is needed (only on Win32 platforms).

MQeMigration.jar
Contains classes that assist in migrating from an earlier version of WebSphere MQ Everyplace.

MQeDeprecated.jar
This contains all of the deprecated class files that are no longer needed by a WebSphere MQ Everyplace application. These deprecated class files help you run applications written using a previous version of WebSphere MQ Everyplace, without making any changes.

MQeDiagnostics.jar
This file helps to diagnose problems with WebSphere MQ Everyplace classes. It contains tooling to search the class path to find out the level of each class found.

Other jar files

MQeExamples.jar
A packaging of all the WebSphere MQ Everyplace examples into one jar file. This includes all of the examples supplied with WebSphere MQ Everyplace, but excludes the deprecated classes.

MQeCore.jar
This contains a minimal set of classes. On its own it is not usable but it can be used as a base for building a small footprint WebSphere MQ Everyplace system. More details on reducing footprint can be found in the ″Optimizing footprint″ section.

Optimizing footprint

In many cases the supplied jar files can be used without change, however there are instances where this is not the case. In particular, on some environments where

footprint is limited, the set of classes that are deployed must be reduced to the smallest possible size. The supplied jar files are general purpose and contain more than is necessary for an optimized environment. This section covers how to optimize the set of classes down to only those that are required for a particular application.

The table below separates the classes into groups associated with a particular function or configuration and will help determine which classes will be required to optimize an applications footprint. Using this table the minimum required set of classes can be deduced by taking the mandatory classes for the required categories and then adding in required optional classes for that category.

Due to the wide ranging set of Java runtimes that are now available, not all classes can run on all runtimes. The table lists all classes, unless otherwise stated, each class will run on a J2SE runtime. Due to the differences between a J2SE and a J2ME runtime, some of the classes are not appropriate for a J2ME runtime. There are two columns that show which classes can be used on J2ME MIDP and J2ME CDC/Foundation runtimes.

Table 41.

Category	Detail	Classes required (com.ibm.mqe.)	Midp compliant	CDC / Foundation compliant
Mandatory classes	For all queue managers	MQe MQeAdapter MQeAttribute MQeAttributeDefaultRule MQeAttributeRule MQeAuthenticator MQeCompressor MQeCryptor MQeEnumeration MQeException MQeExceptionCodes MQeField MQeFields MQeKey MQeLoaderMQeProperties MQePropertyProvider MQeQueueControlBlock MQeQueueProxy MQeQueueManager MQeQueueManagerRule MQeResourceControlBlock MQeRule MQeRunnable MQeRunnableInstance MQeThread MQeThreadPool$1 MQeThreadPool$PooledThread MQeThreadPool$Target MQeThreadPool MQeTrace MQeTraceHandler MQeTraceInterface registry.MQeRegistry	X	X
Registry type	One option in this category must be selected			
File registry	Add required: Storage adapter	registry.MQeFileSession registry.MQeRegistrySession	X	X

Table 41. (continued)

Category	Detail	Classes required (com.ibm.mqe.)	Midp compliant	CDC / Foundation compliant
Private registry w/o credentials	Add: File registry	registry.MQePrivateRegistry registry.MQePrivateSession		X
Private registry with credentials	Add: Private registry w/o credentials	attributes.MQeMiniCertRequest attributes.MQeSharedKey attributes.MQeWTLSCertificate		X
	Mini-certificate management functions	attributes.MQeListCertificates registry.MQePrivateRegistryConfigure		X
Public registry	Applicable to types of message-level security Add: Private registry with credentials	registry.MQePublicRegistry		X
Queue manager type	For all types add required: Administration Storage adapters Message store Authenticators Cryptors Compressors Rules Security			
Standalone qMgr.		No additional classes		
Client qMgr.	Add required: Communications	MQeTransporter adapters.MQeCommunicationsAdapter communications.MQeChannel communications.MQeChannelCommandInterface communications.MQeChannelControlBlock communications.MQeCommunicationsException communications.MQeCommunicationsManager communications.MQeConnectionDefinition communications.MQeListener communications.MQeListenerSlave	X	X

Table 41. (continued)

Category	Detail	Classes required (com.ibm.mqe.)	Midp compliant	CDC / Foundation compliant
Server qMgr.	Add: Client qMgr. Add required: Communications	Note: whilst MQeListener is not used in the Client, they need to be included when preverifying a J2ME application		X
Gateway qMgr.	Add: Server qMgr. Add required Communications Transformers	MQeBridgeLoadable MQeBridgeManager mqbridge.*		
Communications				
TCP/IP w/o history & persistence		adapters.MQeTcpipAdapter adapters.MQeTcpipLengthAdapter		X
TCP/IP with history & persistence	Add: TCP/IP w/o history and persistence	adapters.MQeTcpipHistoryAdapter adapters.MQeTcpipHistoryAdapterElement		X
HTTP 1.0 Not to WES Proxy Authentication server		adapters.MQeTcpipAdapter adapters.MQeTcpipHttpAdapter		X
HTTP To WES Proxy Authentication server		adapters.MQeTcpipAdapter adapters.MQeWESAuthenticationAdapter		X
HTTP 1.1/1.0 J2ME	MIDP only	adapters.MQeMidpHttpAdapter	X	
UDP		adapters.MQeUdpipBasicAdapter$Initiator adapters.MQeUdpipBasicAdapter$InternalAdapter adapters.MQeUdpipBasicAdapter$Responder adapters.MQeUdpipBasicAdapter$Writer adapters.MQeUdpipBasicAdapter		X
Queue types	For all queue types add required: Authenticators Cryptors Compressors Rules			

Table 41. (continued)

Category	Detail	Classes required (com.ibm.mqe.)	Midp compliant	CDC / Foundation compliant
Local	Add: Storage adapter Message storage	MQeAbstractQueueImplementation MQeEventTrigger MQeMessageEvent MQeMessageListenerInterface MQeQueue MQeQueueRule (or replacement)	X	X
Remote	Add: Local queue (storage adapter & msg. storage only if needed)	MQeRemoteQueue	X	X
Home server	Add: Remote queue (no storage adapter or msg. storage)	MQeHomeServerQueue	X	X
Store and forward	Add: Remote queue	MQeStoreAndForwardQueue	X	X
Bridge queue	Add: Remote queue	mqbridge.MQeMQBridgeAdminMsg mqbridge.MQeBridgeServices mqbridge.MQeMQBridgeQueue mqbridge.MQeMQQMgrName mqbridge.MQeMQQName		
Message storage				
Base		MQeMessageStoreException MQeAbstractMessageStore messagestore.MqeIndexEntry	X	X
Standard	Add: Base	messagestore.MQeMessageStore	X	X
Short filename. Always use 8.3 file name for messages.	Add: Standard	messagestore.MQeShortFilenameMessageStore		X
4690 specific	Add: Short filename	messagestore.MQe4690ShortFilenameMessageStore		
Message type				
Basic		Support for MQeMsgObject is in Mandatory classes	X	X
MQSeries		mqemqmessage.*		

Table 41. (continued)

Category	Detail	Classes required (com.ibm.mqe.)	Midp compliant	CDC / Foundation compliant
Storage adapters				
Assured disk	Independence from OS lazy writes	adapters.MQeDiskFieldsAdapter		X
Non-assured disk	Dependence on OS lazy writes Add: Assured disk	adapters.MQeReducedDiskFieldsAdapter		X
Case-Insensitive	Add: Assured disk	adapters.MQeCaseInsensitiveAdapter		X
Long to Short Filename Mapping		adapters.MQeMappingAdapter		X
Midp RMS Storage	MIDP Only	adapters.MQeMidpFieldsAdapter com.ibm.mqe.adapters.MQeMidpFieldsAdapter$RMSFile	X	
Memory	Volatile storage	adapters.MQeMemoryFieldsAdapter	X	X
Administration				
Basic administration capability	Add: Local queue	MQeAdministrator MQeAdminMsg MQeAdminQueue MQeAdminQueue$1 MQeAdminQueue$Timer	X	X
Manage queue manager	Add: Basic administration capability	administration.MQeQueueManagerAdminMsg	X	X
Manage connection definitions	Add: Basic administration capability	administration.MQeConnectionAdminMsg	X	X
Manage communications listeners	Add: Basic administration capability	administration.MQeCommunicationsListenerAdminMsg	X	X
Manage local queue	Add: Basic administration capability	administration.MQeQueueAdminMsg	X	X
Manage administration queue	Add: Manage local queue	administration.MQeAdminQueueAdminMsg	X	X
Manage remote queue	Add: Manage local queue	administration.MQeRemoteQueueAdminMsg	X	X

Table 41. (continued)

Category	Detail	Classes required (com.ibm.mqe.)	Midp compliant	CDC / Foundation compliant
Manage home server queue	Add: Manage remote queue	administration.MQeHomeServerQueueAdminMsg	X	X
Manage store and forward queue	Add: Manage remote queue	administration.MQeStoreAndForwardQueueAdminMsg	X	X
Manage bridge queue	Add: Manage remote queue	mqbridge.MQeMQBridgeQueueAdminMsg mqbridge.MQeCharacteristicLabels		X
Manage a bridge to MQSeries	Add: Remote queues	mqbridge.*AdminMsg mqbridge.MqeCharacteristicLabels mqbridge.MqeRunState mqbridge.MqeBridgeServices mqbridge.MQeBridgeExceptionCodes		
Queue manager creation and deletion		MQeQueueManagerConfigure	X	X
Authenticators				
mini-certificate		attributes.DHk (source may be generated) attributes.MQeSharedKey attributes.MQeRandom attributes.MQeWTLSCertificate attributes.MQeWTLSCertAuthenticator		X
Compressors				
GZIP		attributes.MQeGZIPCompressor		X
LZW		attributes.MQeLZWCompressor attributes.MQeLZWDictionaryItem	X	X
RLE		attributes.MQeRleCompressor	X	X
Cryptors				
triple DES		attributes.MQe3DESCryptor		X
DES		attributes.MQeDESCryptor		X
MARS		attributes.MQeMARSCryptor		X
RC4		attributes.MQeRC4Cryptor		X
RC6		attributes.MQeRC6Cryptor		X
XOR		attributes.MQeXorCryptor	X	X
Application security services				

Table 41. (continued)

Category	Detail	Classes required (com.ibm.mqe.)	Midp compliant	CDC / Foundation compliant
Local security	Add required: Cryptors	attributes.MQeLocalSecure	X	X
Message-level security	Add required: Cryptors	attributes.MQeMAttribute		X
Message-level security with digital signature & validation	Add: Public registry. Add required: Cryptors	attributes.MQeMTrustAttribute		X
Trace				
Collect binary trace in J2SE/CDC		trace.MQeTraceToBinary trace.MQeTraceToBinaryFile		X
Collect binary trace to Midp RMS Store And or send to MIDP Trace servlet		trace.MQeTraceToBinary trace.MQeTraceToBinaryMidp	X	
Base trace renderer		trace.MQeTracePoint trace.MQeTracePointGroup trace.MQeTraceRenderer		X
Decode a binary file to readable form	Add: Base trace renderer	trace.MQeTraceToReadable trace.MQeTraceFromBinaryFile		X
Trace to a readable output stream	Add: Base trace renderer	trace.MqeTraceToReadable		X
Servlet collection of Midp binary trace	Add Base trace renderer	trace.MQeTraceToReadable examples.trace.MQeServlet		
Miscellaneous				
Cryptographic support	Application or installation use only	attributes.MQeCL (footnote?) attributes.MQeGenDH (generates a version of attributes.MQeDHk.java)		X
Mini-certificate server SupportPac ES03	MQe_MiniCertServer (or command line tool) See ES03 installation instructions			

Table 41. (continued)

Category	Detail	Classes required (com.ibm.mqe.)	Midp compliant	CDC / Foundation compliant
MQe_Explorer SupportPac ES02	MQe_Explorer See ES02 installation instructions			
Bindings	Access to Java classes from other languages			
C language		bindings.*		
JMS	Support for the Java Message Service API	jms.* transaction.*		X X
User-defined MQe extensions				
		Authenticators Communications adapters Compressors Cryptors Logging classes Message classes Rule classes Security control Storage adapters Trace handler		

JMS requirements

In order to use the WebSphere MQ Everyplace JMS programming interface, the JMS interface classes are required. These are contained typically in JMS.jar. WebSphere MQ Everyplace does not ship with jms.jar, and this must be downloaded before JMS can be used. At the time of writing, this can be freely downloaded from http://java.sun.com/products/jms/docs.html. The JMS Version 1.0.2b jar file is required.

In addition, if JMS administered objects are to be stored and retrieved using the Java Naming and Directory Interface (JNDI), the javax.naming.* classes must be available. If Java 1 is being used, for example, a 1.1.8 JRE, jndi.jar must be obtained and added to the classpath. If Java 2 is being used, a 1.2 or later JRE, the JRE might contain these classes. You can use WebSphere MQ Everyplace without JNDI, but at the cost of a small degree of provider dependence. WebSphere MQ Everyplace-specific classes must be used for the ConnectionFactory and Destination objects. You can download JNDI jar files from http://java.sun.com/products/jndi

WebSphere MQ Classes for Java requirements

To use the WebSphere MQ bridge the WebSphere MQ Classes for Java are required, version 5.1 or later. These are packaged with WebSphere MQ 5.3 and above. If using an earlier version of WebSphere MQ then they are available for free download from the

Web as supportpac MA88. The Web address for the download is: http://www.ibm.com/software/mqseries/txppacs/ma88.html.

For an example of how to setup the classpath to include WebSphere MQ jar files, see batch files:

- <MqeInstallDir>\Java\Demo\Windows\javaenv.bat
- <MqeInstallDir>\Java\Demo\UNIX\javaenv

Occasionally, the jar files change between versions of the WebSphere MQ, if problems are encountered as a result of this, consult the documentation for WebSphere MQ classes in order to determine the correct jar files to use.

Using WebSphere studio device developer smart linker

The smart linker tool that ships with WebSphere Studio Device Developer is used in the process of building and packaging an application into a jar or jxe file. The smart linker can remove classes (and methods) that are deemed not to be required; this will happen to classes that are dynamically loaded. WebSphere MQ Everyplace makes use of dynamic loading so care should be taken to either avoid this feature or to explicitly name classes that must be present, even though not explicitly referenced in the code.

To prevent unused classes being removed use the - noRemoveUnused option. If the -removeUnused option is set then any class that is dynamically loaded must be specifically included. One option that can be use to achieve this is -includeWholeClass. For example -includeWholeClass ″com.ibm.mqe.adapters.*″ will include all classes in the adapters package and -includeWholeClass ″com.ibm.mqe.adapters.MQeTcpipHttpAdapter″ will only include the http adapter. Multiple include (or exclude) options can be specified in the smart linker options file.

The following guidelines can be used to determine which classes are dynamically loaded. The basic guideline is any class that is referenced though an WebSphere MQ Everyplace class alias or any class that is set as a parameter when administering WebSphere MQ Everyplace resources will be dynamically loaded. This includes:

- Communications adapters
- Storage adapters
- Message stores
- Rules
- Aliases
- Cryptors
- Compressors
- Authenticators
- Queues
- Transporter
- Connection (refer to the following example)

An example of a set of includes needed for a simple MIDP application is:

```
-includeWholeClass "com.ibm.mqe.MQeQueue"
-includeWholeClass "com.ibm.mqe.MQeRemoteQueue"
-includeWholeClass "com.ibm.mqe.MQeHomeServerQueue"
-includeWholeClass "com.ibm.mqe.MQeTransporter"
-includeWholeClass "com.ibm.mqe.communications.MQeConnectionDefinition"
-includeWholeClass "com.ibm.mqe.adapters.MQeMidpFieldsAdapter"
-includeWholeClass "com.ibm.mqe.adapters.MQeMidpHttpAdapter"
-includeWholeClass "com.ibm.mqe.messagestore.MQeMessageStore"
-includeWholeClass "com.ibm.mqe.registry.MQeFileSession"
```

J2ME Midp specifics

When deploying the Java Application for the Midp environment a few additional comments are worth mentioning.

- The developer must use the Midp specific Storage and Communication adapters (see above Table) and exclude any classes that are not Midp compliant.
- The user can either use the prepackaged MQeMidp.jar file or their own reduced version, however a JAD file (Java application descriptor) must also be included detailing the Midlets available within the application. When deploying to the device all classes should be packaged and preverified in one jar before deploying. However, whilst testing using an emulator several jars can be used by including them in the classpath
- Sun and IBM also provide tools that will generate the required .prc file for Palm Devices. See documentation within either Sun's Wireless Toolkit or IBM's WebSphere Studio Device Developer
- Care must be taken to ensure that all the required classes are included in either the jar/prc file or other executable. Some classes are dynamically loaded and may be missed when using any Smart-Linker. See section 15.1.2 Using WebSphere Studio Device Developer Smart Linker for more details.

4690 specifics

Take the following requirements into account when configuring WebSphere MQ Everyplace for use with 4690.

- Terminal Applications are restricted to 24 char maximun path length, but Store Controller Applications can have 127 chars. Java Apps may are also restricted to the 24 length.
- The virtual file system (VFS) cannot hold greater than 64000 files. With GB disk sizes being used, the C: drive may not have a limit on the number of files, depending on your operating system.
- When you want to access a file, you must specify the path that leads to it. The path consists of directory names that are separated by a backslash character ″\″ or a forward slash ″/″.

  Note: Although your system accepts both the ″\″ and the ″/″ character, it is probably less confusing to use one or the other.
- Examples elsewhere in this manual demonstrate how to configure your queue manager such that the data describing its' resources, certificates, and other configuration data is stored in files with long filenames. These filenames are for a single top-level directory, which can also be located on the VFS drive namespace.

- YUsing the 8.3 format, the total character length of the fully-qualified filename exceeds the allowable limits imposed by the 4960 native file system. Therefore, in VFS :
  - The maximum length of a filename is 256 characters.
  - The maximum path length, including directories and files, is 260 characters.
  - The maximum directory depth is 60 levels including the root directory.
- WebSphere MQ Everyplace classes can be stored in long format names in VFS. However, for performance and convenience, as there are lots of class files, we would reccomend that the application and WebSphere MQ Everyplace classes are packaged into a .jar files and deployed.
- According to the VFS manual ″The operating system provides support for file names greater than eight characters in length through the use of a 4690 Virtual File System (VFS)″.
- The VFS manual states: ″The VFS drive setting must be enabled through system configuration. On enabling VFS drive settings, the operating system creates two logical drives. C: and D:. The drive determines where the VFS directory is located. However, the information is actually stored on drives C: and D:. Drive M: information is stored on drive C:, and drive N: information is stored on drive D:. Once you have enabled VFS, you can use drives M: and N: to provide long file name support locally.″

It is recommended that you use the `MQeCaseInsensitiveDiskAdapter` on the 4690 OS. This class implements a disk adapter that is insensitive to the 'case' of the filename used during matching. Some JVM or OS combinations list files with different case to that in which they were created. This means that the simple filtering in the superclass ignores them. This class converts both the compator and the comperand to lowercase before performing the comparison. This ensures the best chance of finding a valid match. Note that the conversion to lower case may be innapropriate on platforms where the case is honoured, and where there are non-mqe files stored that could be confused by case. In summary, as this adapter is more suited for use with the 4690 due its filesystem.

Packaging

Following is a list of some of the techniques and tools that can be used to package applications ready for deployment to a device. The list is not a full list and does not go into any detail but is intended to provide an introduction to some of the ways a Java application can be packaged.

Single Jar file

Build a self-contained application with WebSphere MQ Everyplace embedded in it. This option minimizes the footprint and ensures that the classpath is kept to a minimum.

Multiple Jar files

Put application into one jar file and use either the supplied WebSphere MQ Everyplace jar files or construct a separate WebSphere MQ Everyplace jar file. Keeping WebSphere MQ Everyplace in one or more separate jars makes it easy to use WebSphere MQ Everyplace from multiple independent applications.

JNLP JNLP or Java Network Launching Protocol and API, is an emerging standard, for use in packaging and deploying Java applications. It is designed to automate the deployment, via the web, for applications written to the J2SE platform.

OSGI OSGi or Open Services Gateway Initiative defines define a platform for the packaging of and dynamic delivery of Java software services to networked devices. This is achieved via a consistent, component-based, architecture for the development and delivery of Java software components known as bundles and services. Both WebSphere MQ Everyplace components and applications can be turned into OSGi bundles and services for use in an OSGi environment. The bundles are delivered from a bundle server There are a number of products that provide bundle servers together with the client code to handle the installation and lifecycle of bundles. Depending on implementation the bundles can be downloaded on demand, and updated automatically when a new version is available. IBM WebSphere Studio Device Developer ships with SMF (service management framework), which assists in the creation and testing of bundles together with a bundle server.

Midlet An WebSphere MQ Everyplace J2ME MIDP application must be packaged as a midlet or midlet suite (.jad and .jar).

Palm specific
In order to run on a Palm device a Java application must to be packaged in a prc file, which is a Palm specific format. The IBM WebSphere Studio Device Developer product ships with a tool that will package a Java application as a prc file.

JXE IBM WebSphere Studio Device Developer has a SmartLinker tool that can produce an optimized packaging of an application that contains the minimum set of required classes and methods for the deployment platform. The output from the smartlinker is stored in a .JXE file which is understood by the IBM j9 Java runtime.

Installer
There are a number of tools that will package an application ready for installation on one or more platforms. A couple of examples of these are InstallShield and self extracting zip files.

Roll you own distribution mechanism
For instance using a Java class loader that can load classes over a network.

Deployment to devices

Following is a list of some of the techniques and tools that can be used to deploy applications to devices. The list is by no means complete and does not go into any detail but is intended to provide an introduction to some of the ways a Java application can be deployed.

Device specific tools
Most devices ship with tools that allow applications to be copied across and installed. For instance:

- ActiveSync for PocketPC

- Hotsync for Palm

Development tools
Many development environments (IDEs) like WSDD (IBM WebSphere Studio Device Developer) provide tools that allow deployment of applications onto a device and debugging of the application from the development environment.

OSGi related management
OSGi or Open Services Gateway Initiative defines define a platform for the packaging of and dynamic delivery of Java software services to networked devices. This is achieved via a consistent, component-based, architecture for the development and delivery of Java software components known as bundles and services. Both WebSphere MQ Everyplace components and applications can be turned into OSGi bundles and services for use in an OSGi environment. The bundles are delivered from a bundle server There are a number of products that provide bundle servers together with the client code to handle the installation and lifecycle of bundles. Depending on implementation the bundles can be downloaded on demand, and updated automatically when a new version is available. IBM WebSphere Studio Device Developer ships with SMF (service management framework), which assists in the creation and testing of bundles together with a bundle server.

JNLP JNLP or Java Network Launching Protocol and API, is an emerging standard, for use in packaging and deploying Java applications. It is designed to automate the deployment, via the web, for applications written to the J2SE platform.

Device management products
There are a number of products on the market that can be used for large-scale deployment of software. One example is Tivoli® Configuration Manager from IBM.

C codebase

Chapter 18. Configuring WebSphere MQ Everyplace queuemanagers as servlets

WebSphere MQ Everyplace queuemanagers can run within a servlet. For more information on writing servlets that use WebSphere MQ Everyplace queuemanagers, see the servlet section in the WebSphere MQ Everyplace System Programming Guide.

Note: In WebSphere MQ Everyplace version 2.0, the deprecated jar must be in the classpath for servlets to work.

An example servlet that receives trace from the com.ibm.mqe.trace.MQeTraceToBinaryMidp trace handler is included with the example classes. It is examples.trace.MQeTraceServlet. Using this as an example, the following information explains how to configure it to work with WAS 4.0. Other application servers will require different steps.

Configuring examples.trace.MQeTraceServlet for use with WAS 4.0

First of all, the servlet code must be packaged into a form that suits the application server. This example will create a web module for use with WAS 4.0.

From the WebSphere Administrative Console, choose the menu item Application Assembly tool from the Tools menu. The Application assembly tool should appear.

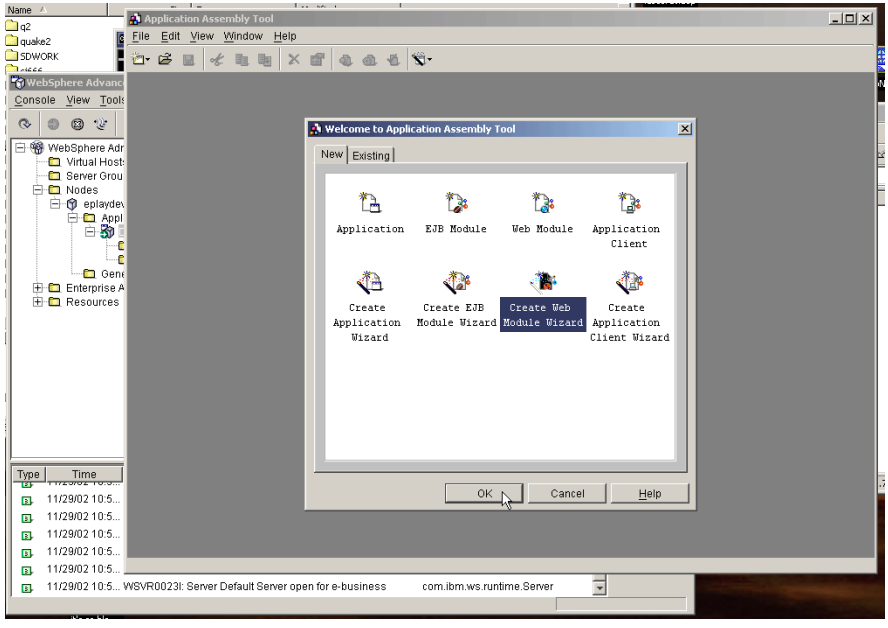

Figure 102. The WebSphere administrative console

Select ″Create Web Module Wizard″, and click OK. In specifying the properties, enter the file name, and more information, if you wish.

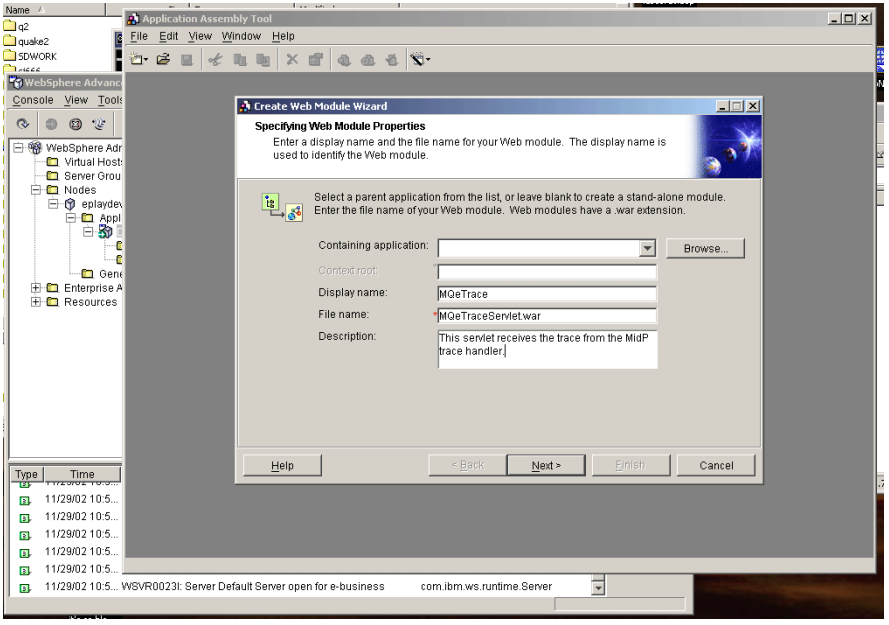

Figure 103. Specifying Web module properties

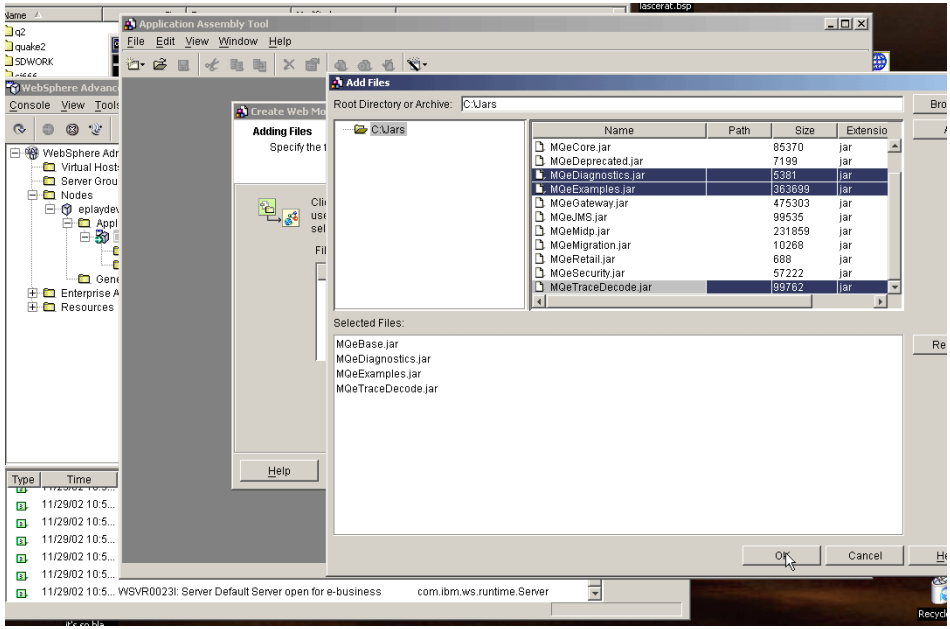

Figure 104. Adding files to the application

The next step is to add files to the application. The examples.trace.MQeTraceServlet is in the MQeExamples.jar and relies on classes from MQeGateway.jar, MQeExamples.jar and MQeTraceDecode.jar.

Since you've included all the classes you need, the next panel that asks you if you want to make distributable, or set a classpath, can be left blank, just click next. The next panel is to set any icons for this web application. If you don't have any, just click next.

Next you have to specify the component properties.

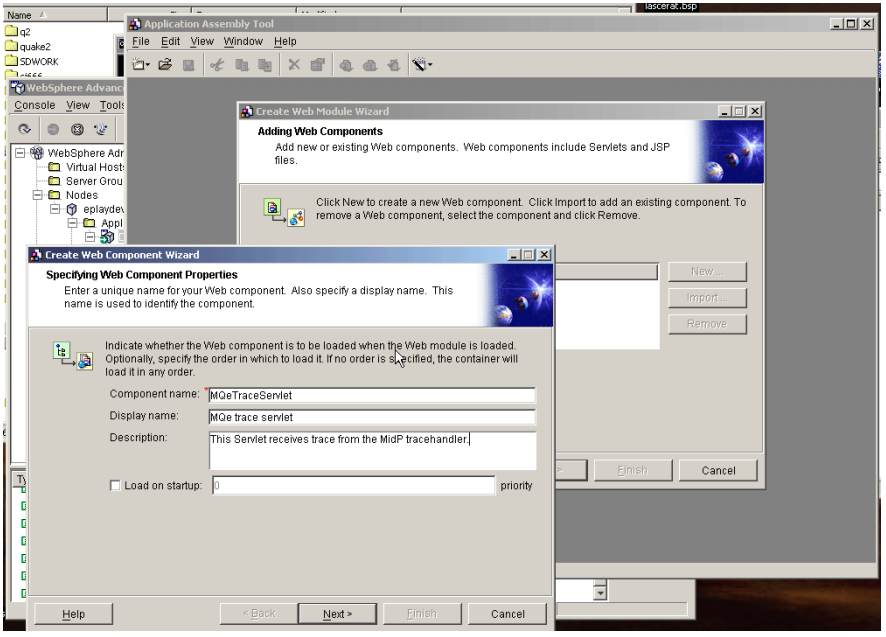

Figure 105. Adding web comopnents

Only the component name is compulsory, but you may want to add a display name and a description.

The next panel allows you to specify which class is the servlet to run.

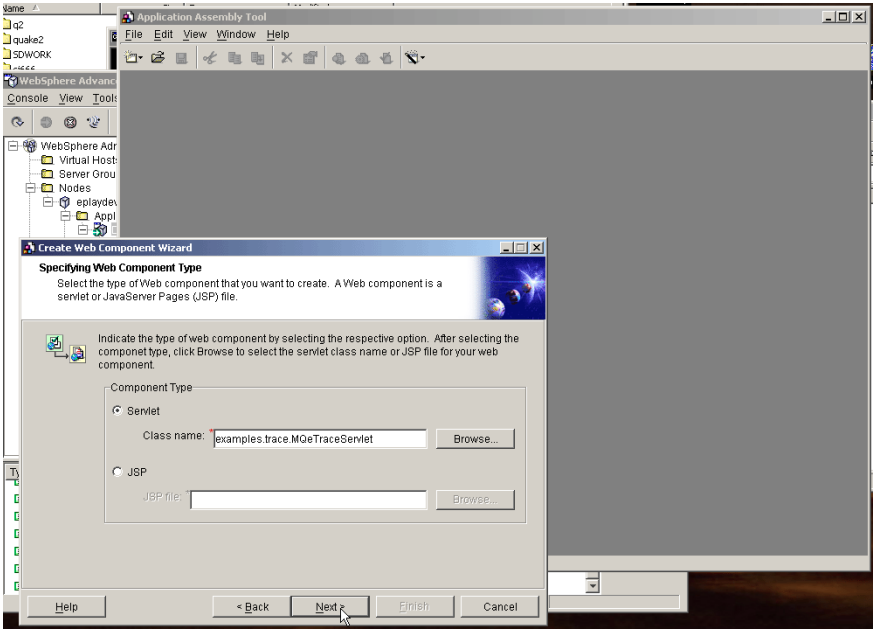

Figure 106. Specifying component type and class name

The next four panels can safely be left blank, they are for specifying icons, security roles and initialization parameters.

After this, you must specify what URL will map to your servlet. The final URL will be of the form http://hostname:port/specified_dir/specified_url_pattern

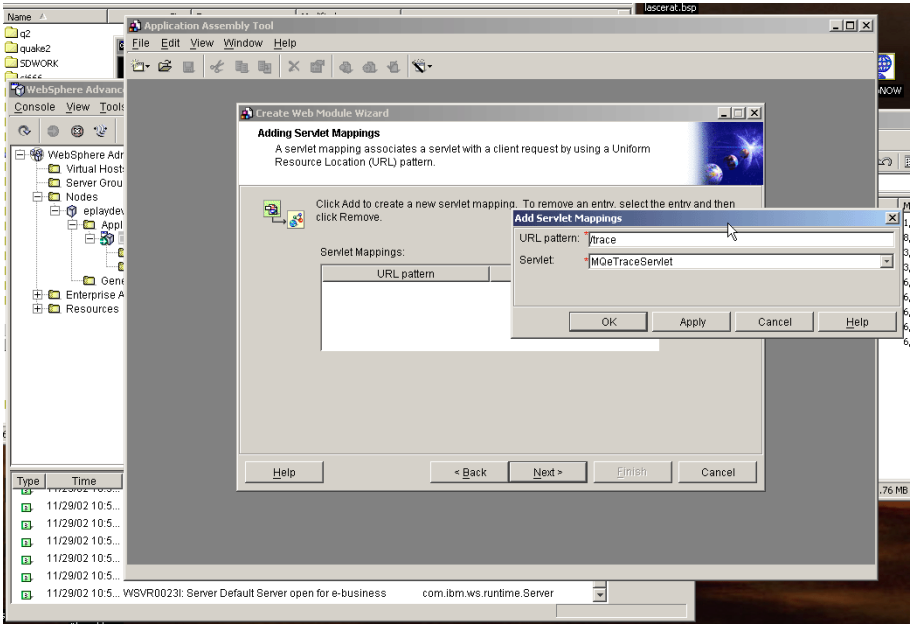

Figure 107. Specifying a URL to map to your servlet

All of the subsequent panels can be left blank. They are for adding resources, context parameters, error pages, MIME mappings, tag libraries, welcome files and EJB references.

Click Finish, and then save the file. If you save the file to \AppServer\InstallableApps\ where you installed WebSphere application server, then it will automatically appear in the list of servlets in the administration panel.

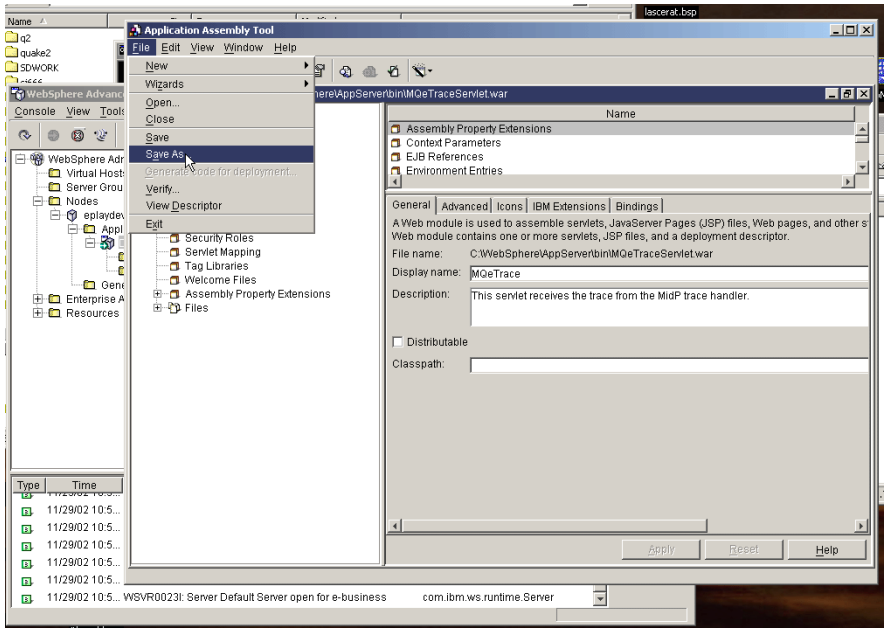

Figure 108. Saving the file

Next, this component needs to be imported and started. From the wizards button, select ″Install Enterprise Application″.

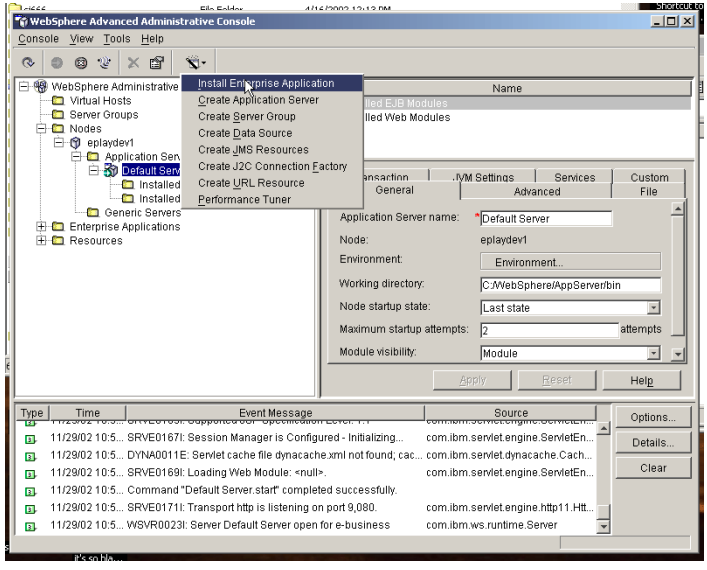

Figure 109. Install enterprise application

Install your component as a standalone module.

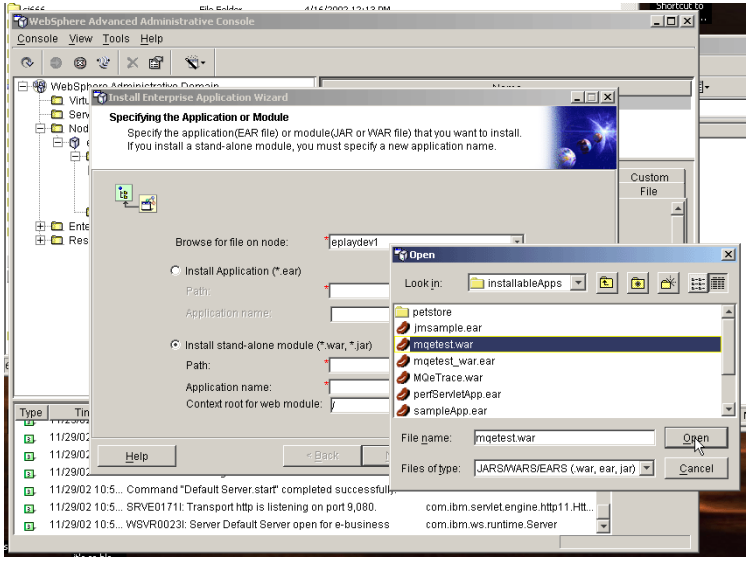

Figure 110. Installing your component as a standalone module

Specify an application name, and a root for the web module. This is the part of the URL immediately after the http://hostname:portnumber/ and shouldn't be left as /

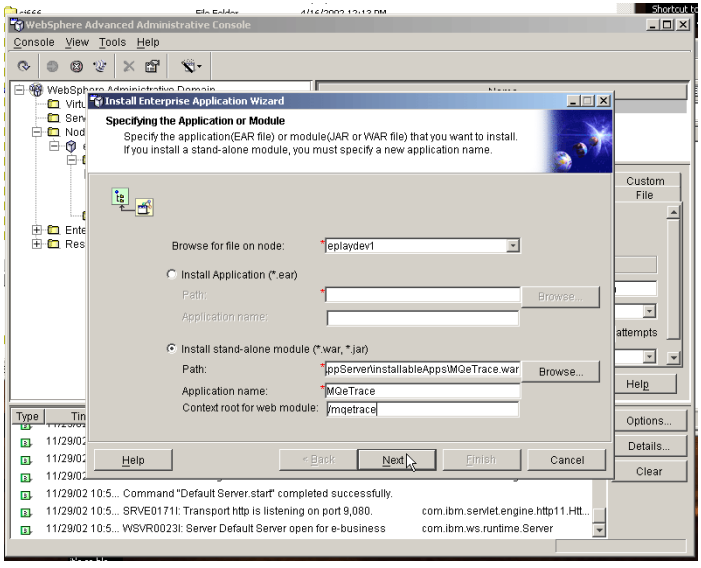

Figure 111. Specifying an applilcation name

All of the subsequent panels can be left blank, they are about controlling users, EJB roles, JNDI bindings, EJB mappings, resource references, datasources for EJB, data sources for CMP, and virtual hosts.

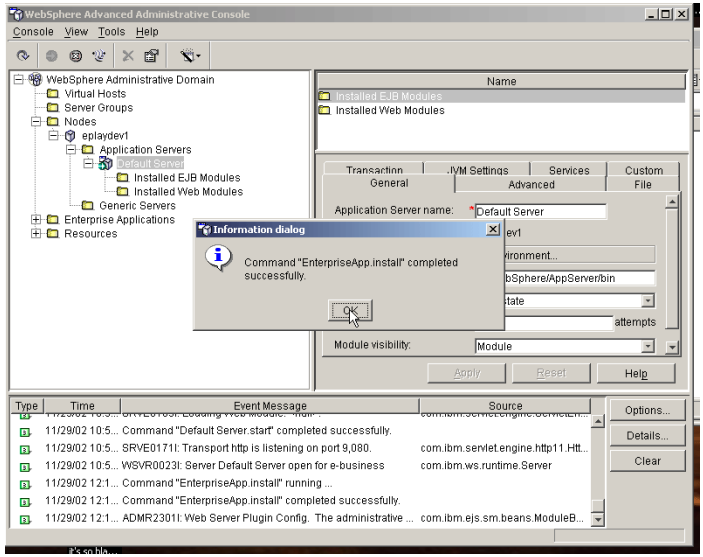

Figure 112. Information dialog

Next, the web module has to be started. Select the application server that it has been configured for. It should appear under Installed Web Modules.

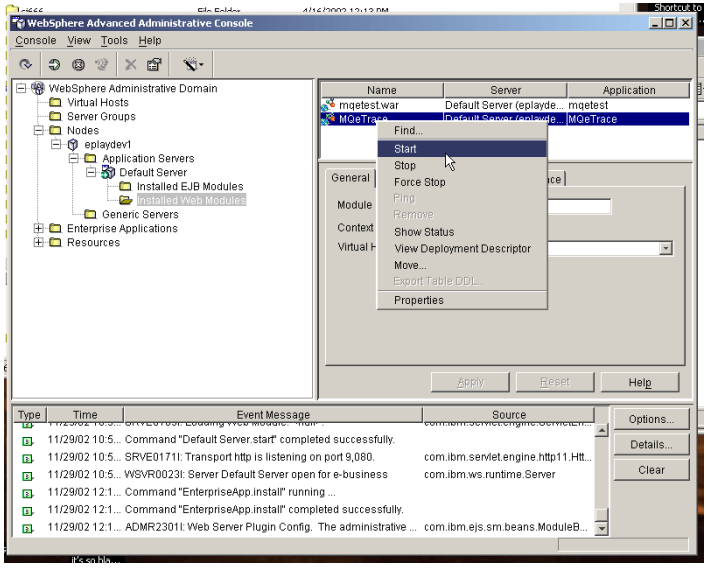

Figure 113. Starting the web module

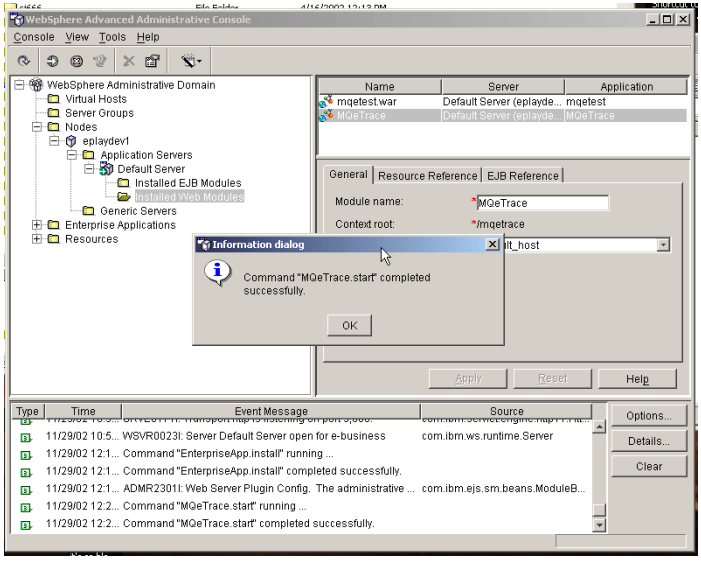

Figure 114. Information dialog success message

If everything went well, it should now be available for use from the com.ibm.mqe.trace.MQeTraceToBinaryMidp. Since this servlet doesn't support get, then viewing the URL with a web browser will result in a 405 error. This is normal.

If your application server is set up with the defaults, the URL for the servlet is http://localhost:9080/mqetrace/trace.

Chapter 19. Configuring WebSphere MQ Everyplace for performance

WebSphere MQ Everyplace can be used in a number of different configurations, and the performance you can expect will vary a great deal depending on your adapters and manner of use.

The main thing to be aware of when configuring WebSphere MQ Everyplace is that disk accesses are the single biggest cause of slowdown in a WebSphere MQ Everyplace system. All unnecessary disk accesses should be designed out from the beginning.

Try to split the messages that you'll be dealing with into messages that it's important are persistent and messages that do not need to be persistent. The persistent messages need to use a disk fields adapter for storage, but the non-persistent ones should use a memory fields adapter. Non-persistent messages stored in memory can go around 100 times faster than messages stored to disk.

When possible, distribute queues across different physical hard discs, so that reads and writes to different queues can take place using different hardware and happen simultaneously.

When multiple clients are accessing a single server, use multiple queues, as only one client can use a queue at a time. Avoid very large numbers of queues, as this increases the time to do any WebSphere MQ Everyplace access.

Keep polling systems such as trigger transmit rules or home serve queue polls to a minimum. Unless you need a specific performance characteristic, the intervals between these can often be configured to be quite large. If you are using them together, then the trigger transmit rule, which is only used to automatically recover a home server queue from network stoppage can often be set to have a much larger interval. If you are designing an application that makes use of home server queues and you are using a trigger transmission rule, then consider replacing it with a user interaction to cause the trigger transmission.

Most JVMs can have their initial memory settings tweaked. These settings are often on -msX and -mxX. Executing java -X will give you more information. Try increasing the initial and maximum heap size to as much as you can without causing the machine to start paging.

If you are running some application with a queue manager that is under a lot of external load, be aware that your own application may suffer from reduced performance as many threads to deal with incoming messages are started. Making sure your own application is multithreaded can reduce this problem.

Trademarks

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

AIX Everyplace IBM iSeries MQSeries WebSphere z/OS zSeries

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

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

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

Appendix. Sending your comments to IBM

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

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

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

To make comments about the functions of IBM products or systems, talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

You can send your comments to IBM in any of the following ways:

- By mail, to this address:

  User Technologies Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER,
  Hampshire
  SO21 2JN
  United Kingdom
- By fax:
  - From outside the U.K., after your international access code use 44–1962–842327
  - From within the U.K., use 01962–842327
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink™: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever method you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

Glossary

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

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

Connection resolution. Performed by a remote queue reference when routing a message (or request) to the real destination queue.

late resolution. The resolution of a queue alias is performed just before the message is routed to the queue. This resolution is as late as possible.

queue alias. You can set an alternative name for a queue. This allows you to refer to a queue by more than one name.

queue manager alias. You can set an alternative name for a queue manager. This allows you to refer to a queue manager by more than one name.

queue resolution. The process by which a queue manager chooses which queue to place a message on.

Printed in U.S.A.

SC34-6283-01