WebSphere MQ Everyplace V2.0.2

Contents

Chapter 1. How to configure MQe objects

Overview of configuring MQe queues, queue managers, and networks

This part of the information center provides the basic information necessary in order to configure MQe queue managers and networks. It is also designed to help you to customize a configuration matching your specific business requirements. It describes how individual MQe components can be created and administered and how components may be used together in various topologies.

Introduction

This book provides the basic information necessary in order to configure MQe 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 MQe 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 an MQe 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 MQe 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 MQe queue manager and associated components follows.

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
Carry out steps 1, 2, 3, 4 and 5

To configure a Server
Carry out steps 1, 2, 6 and 7

To configure a queue manager with both Server and Client functionality
Carry out steps 1 through 7 inclusive

Table 1. Configuring clients, servers, and queue managers

Requisite steps	Topics
Basic queue manager	
1. Create and start the queue manager	“Configuring with messages” on page 14

Table 1. Configuring clients, servers, and queue managers (continued)

Requisite steps	Topics
2. Create a local queue	"Configuring queue managers" on page 39 "Configuring local queues" on page 48
Client queue manager	
3. Create a connection definition to a server	"Configuring connection definitions" on page 72
4. Create a remote queue definition	"Configuring remote queues" on page 58
5. Create a home server queue for triggered transmission (required for remote asynchronous queues)	"Configuring home server queues" on page 64
Server queue manager	
6. Create a listener	"Configuring a listener" on page 79
7. Create a store-and-forward queue (optional)	"Configuring store-and-forward queues" on page 67
8. Add bridge functionality	"Configuring bridge/gateway resources" on page 81

Overview of MQe objects

Queue manager

A queue manager owns and controls MQe 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 MQe 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.

MQe 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 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 an MQe 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 managers

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 2. 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		

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 MQe 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:

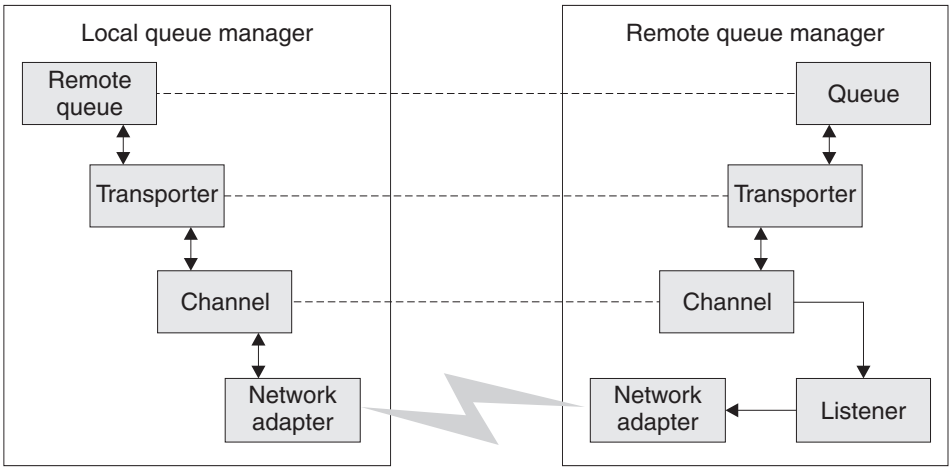


Figure 1. Queue manager connections

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;
}
```

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 );

}
```

Routing and aliases

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 to 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 manipulate 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 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 MQe. 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 following diagram 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. The following diagram shows a local queue:

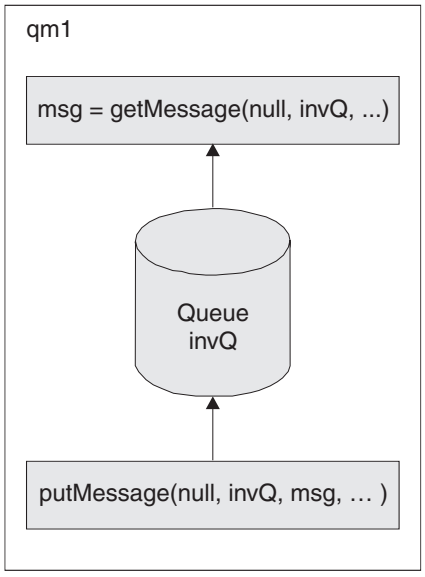


Figure 2. Local queue

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
```

MQe Version 2.1 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 (" .\"), whilst 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 the other queue managers in the network to send messages to the queue, or receive messages from the queue. The following characteristics are used in setting up queue security:

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

If either a cryptor or authenticator has been specified on a queue, the queue manager must have a private registry defined. Any other queue manager that has remote queues which are directed to a queue with security must also have a private registry. The only exception to this requirement is when using remote synchronous queues.

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 Java API Programming Reference.

Aliases:

Queue names can have aliases similar to those described for connections in "Routing and aliases" on page 5. 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:

Some administrative actions can be performed only 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 message” on page 17 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 polled only 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 stops trying until a trigger transmit call is made.

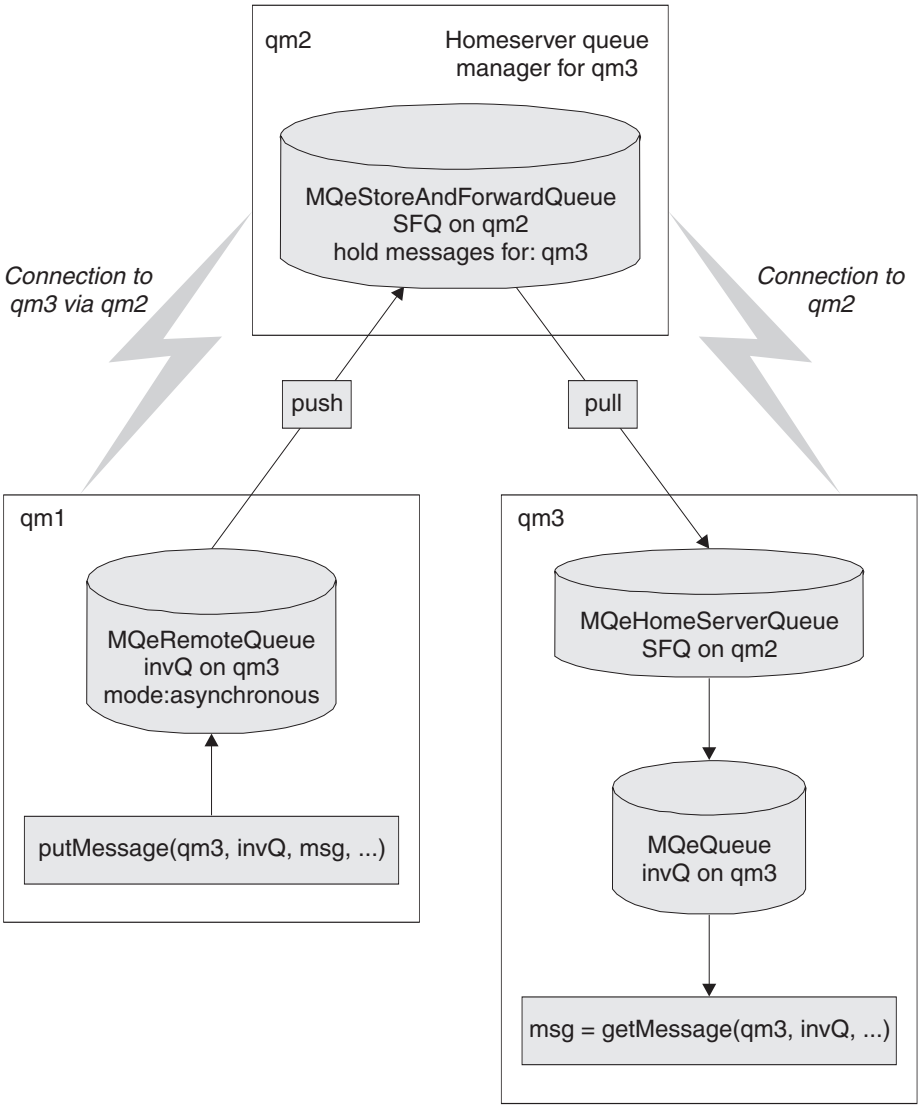


Figure 3. Home-server queue

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 3 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.

MQ bridge queue

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

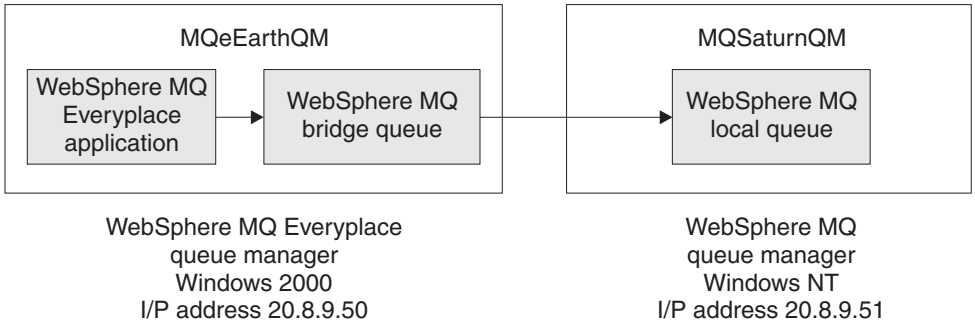


Figure 4. MQ bridge queue

- The MQSaturnQM MQ queue manager has a local queue MQSaturnQ defined .
- The MQeEarthQM must have an MQ bridge queue defined called MQSaturnQ on the MQSaturnQM queue manager.
- Applications attached to the MQeEarthQM queue manager put messages to the MQSaturnQ 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, 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 17 on page 83 for more information. This information identifies how the MQ bridge accesses the MQ queue manager, to manipulate an MQ queue.

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

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

The following table details the list of operations supported by the MQ bridge queue, once it has been configured:

Table 3. Message operations supported by MQ—bridge queue

Type of operation	**Supported by MQ bridge queue**
getMessage()	yes*
putMessage()	yes
browseMessage()	Yes*
browseAndLockMessage	no
Note: * These functions have restrictions on their use.	

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

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

If you do not want to use synchronous operations against the MQ bridge queue, you can set up an asynchronous remote queue definition that refers to the MQ bridge queue. Alternatively, you can set up a store-and-forward queue, and home-server queue. These two alternative configurations provide the application with an asynchronous queue to which it can put messages. With these configurations, when your `putMessage()` method returns, the message may not necessarily have passed to the MQ queue manager.

An example of MQ bridge queue usage is described in "Bridge configuration example" on page 92.

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 message" on page 17 provides details on setting up the count for the maximum number of attempts. 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 MQe 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. MQe provides the basic facilities for securing administration using queue-based security, as described in this information center.

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.

Configuring with messages

This topic explains how you can administer MQe resources, locally or remotely, using administration messages.

Configuration by messages overview

You can administer MQe resources using specialized messages called administration messages (admin messages). Using these messages allows you to administer resources locally or remotely. The native code base, 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 "Configuring with the C administrator API" on page 30. Java is administered by admin messages. C can be, but has an administration interface for local administration.

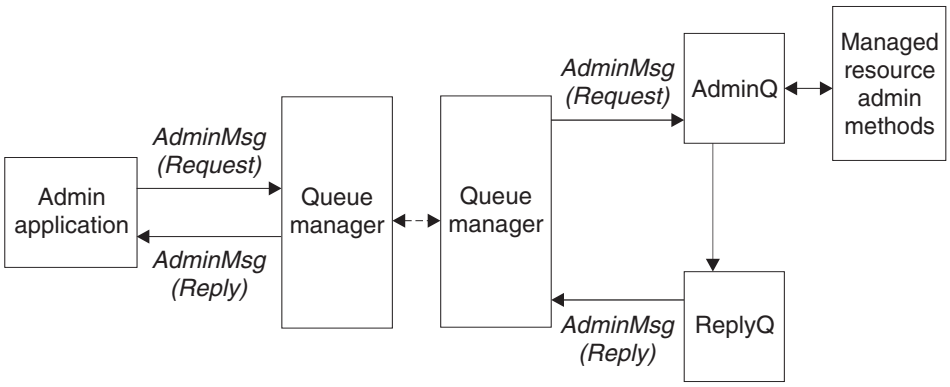


Figure 5. MQe administration using administration messages

These are the steps you need to follow when using administration messages to administer a resource:

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 code base, 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 code base.

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 topic 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 needs 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 code base, as in the Java code base, 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 code base.

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 know how to perform administration of individual resources. This information 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 MQe resources:

Table 4. 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
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 MQ bridge is provided in the com.ibm.mqe.mqbridge package.

C

In the C code base, 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. You need to create an admin message of the appropriate type from new, inserting all of the required fields. Alternatively, for local administration, use the native administration API. The native code base can respond correctly to all administration messages but the native administration API is usually used for local administration.

Set the required fields in a message - Java

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 message

Every request to administer an MQe resource takes the same basic form. The following table 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.

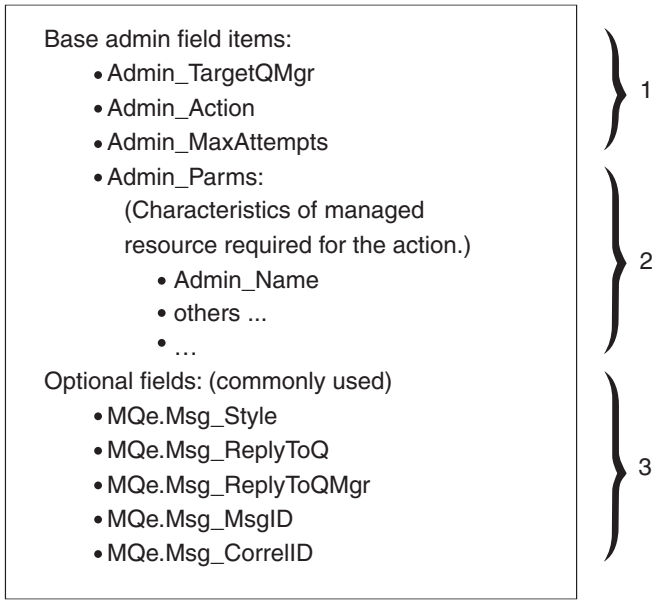


Figure 6. Administration request message

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 5. 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
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 6. 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 24 for more details on using the reply data.

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

Table 7. 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. MQe 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.

Administration message Java examples 1

As this is a frequently performed process, this code example combines each step in the `primeAdminMsg()` method, that can be invoked in other sections of this documentation (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 message on the target 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.

```
//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 MQe message processing is used to wait for a reply or notification of a reply. In the Java code base, 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 - C

This section applies to the C code base 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.

Analyzing the data in the 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:

1. **Total success** - the action happened as requested. For enquire requests the messages contains the data requested.
2. **Total failure** - the action failed. The message contains a reason why the action failed.
3. **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.

Total success

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.

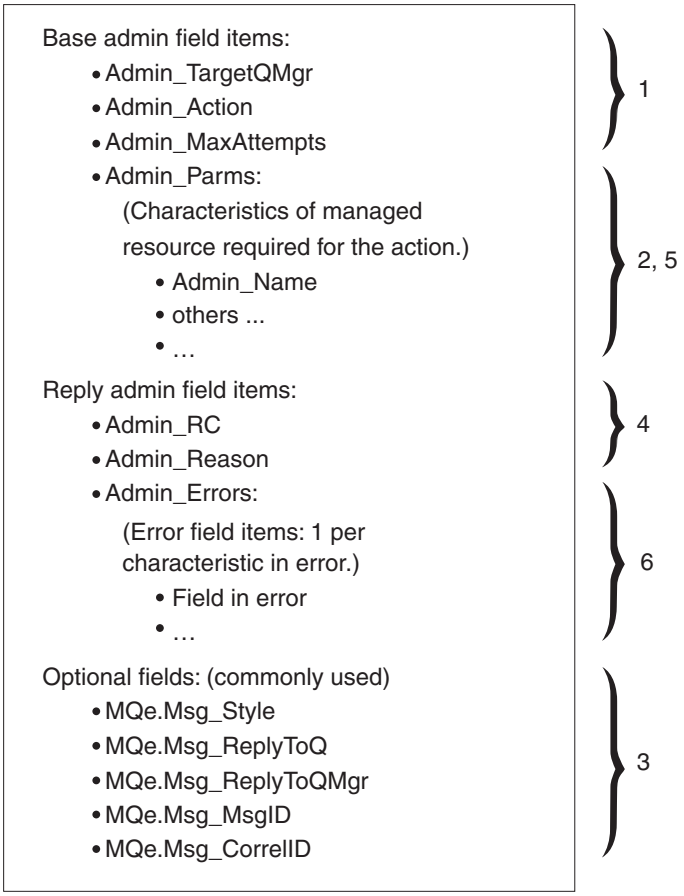


Figure 7. Administration reply message

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 message" on page 17. 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 8. 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 9. 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 10. 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"
```

Administration message Java examples - 2

This method shows how you might analyze a reply message, and return a boolean to indicate whether or not the action was successful. Error messages are printed 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;
}
```

Decorating the queue manager

This method is implemented in class examples.config.BasicAdministration. It addresses the administration message, requests a reply, and adds 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

Use the MQeQueueManager API call `putMessage()`, specifying the destination queue manager and the standard administration queue name. You can ignore the attribute, and confirmed parameters in the 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

This method is implemented in class `examples.config.BasicAdministration`. It is a simple wrapper for the MQeQueueManager API call `waitForMessage()`, which 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

This method is implemented in class `examples.config.BasicAdministration`. It shows how you might analyze a reply message, and return a reply that indicates whether or not the action was successful. Any error messages are printed 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;
}
```

Updating a queue manager description

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);
    }
```

Configuring with the C 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 code base responds to admin messages correctly but no provision is provided for creating them. Therefore, the Administrator API is the recommended method for local administration.

For complete documentation on the Administrator API and all the available options, refer to the 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, or synchronous remote queue
- `action` is the operation to be performed, for example, create, delete, inquire, or update.

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 that initialize these opFlag structures to appropriate values, and macros for each bit that 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 that 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 8. Creating an Administrator Handle for a new Queue Manager

When a handle has been freed, set it to NULL. If this handle is then reused accidentally, 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 9. 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
	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

Configuring from the command line

MQe includes some tools that enable the administration of MQe 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
Opens a server queue manager, adds a home-server queue, and closes the queue manager.

ConnectionCreator
Allows a connection to be added to an MQe 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 MQ bridge on an MQe queue manager.

MQQMgrProxyCreator
Creates an MQ queue manager proxy for a bridge.

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

MQListenerCreator
Creates an MQ transmit queue listener to pull messages from MQ.

MQBridgeQueueCreator
Creates an MQe queue that can reference messages on an 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 an 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 use of command-line tools

You can use the command-line tools to create an initial queue manager configuration using a script, without needing to know how to program in Java.

The following example demonstrates how to use these tools to configure the network topology shown in the following figure:

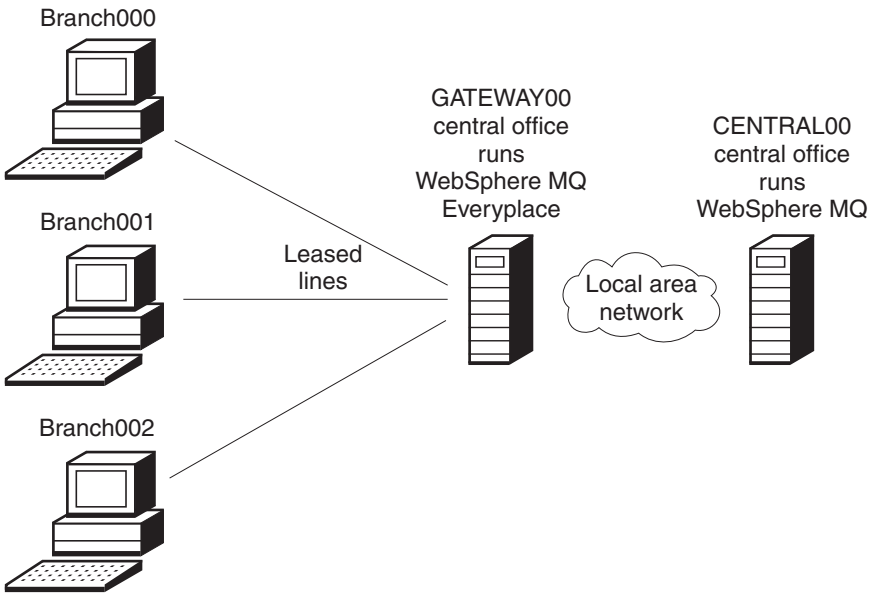


Figure 10. MQe administration scenario

In this scenario:

- The branch offices need to send sales information to the central site for processing by applications on the MQ server
- Each branch has a single machine with DNS names `BRANCH000`, `BRANCH001`, and `BRANCH002` respectively. These machines all run MQe, and each has a single queue manager called `BRANCH000QM`, `BRANCH001QM`, and `BRANCH002QM` respectively.
- The central office machine `GATEWAY00` runs a single gateway queue manager `GATEWAY00QM`
- The central office machine `CENTRAL00` runs MQ with a single queue manager called `CENTRAL00QM`
- When a sale occurs, a message is sent to the MQ queue manager called `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 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 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.

MQe and MQ objects defined by the scripts

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

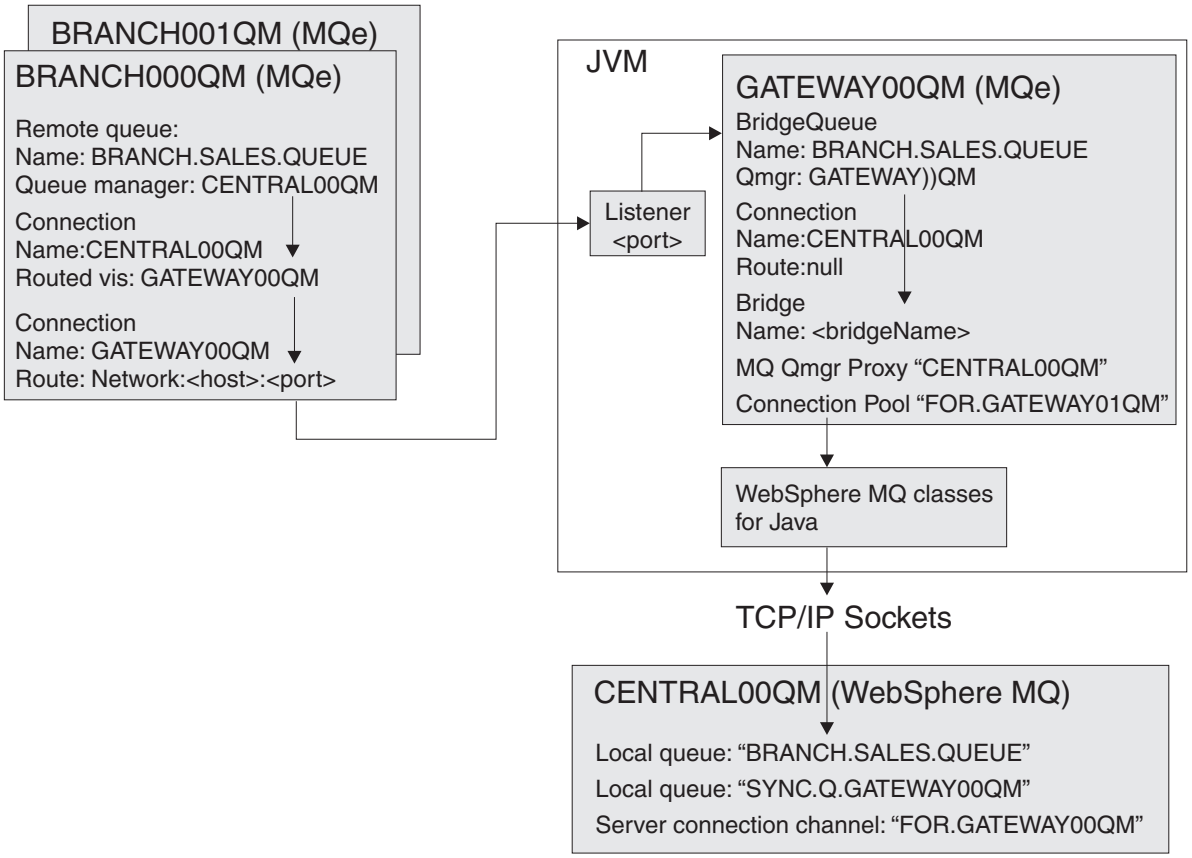


Figure 11. Branch to central routing

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

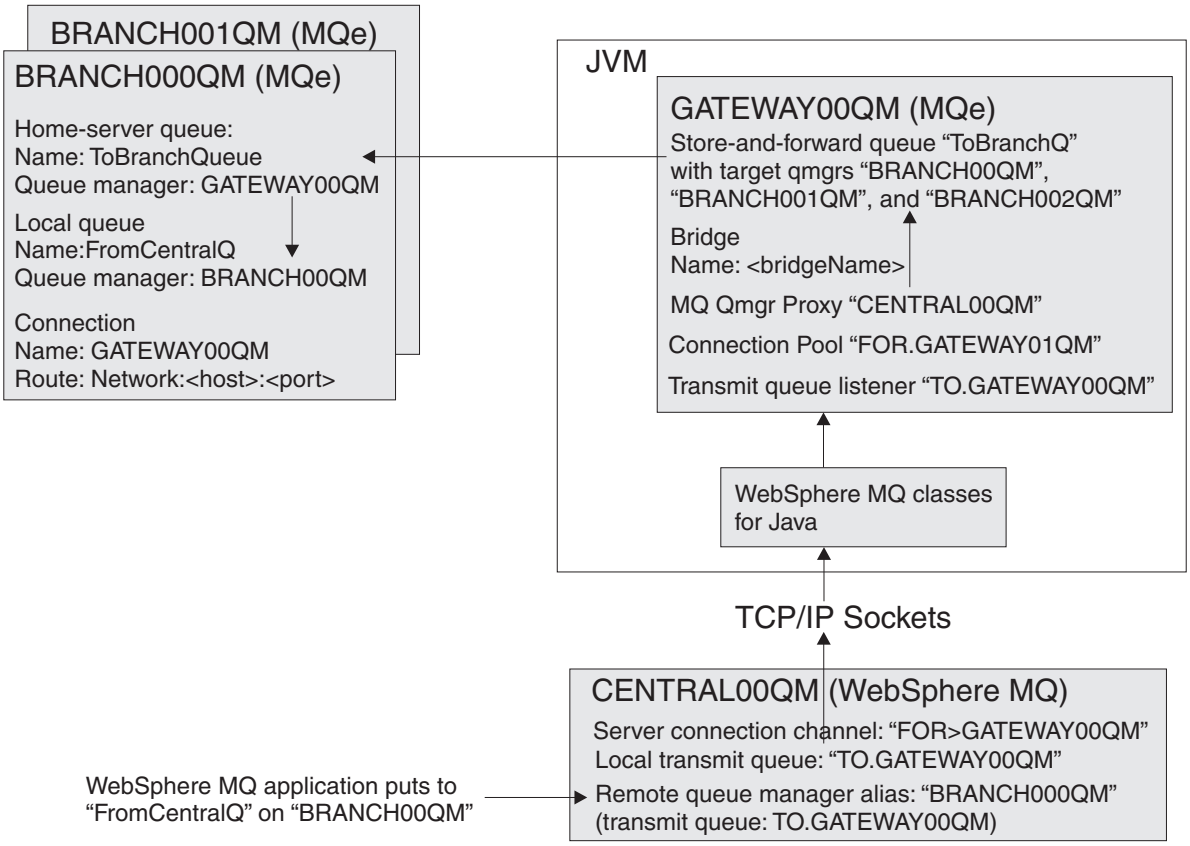


Figure 12. Central to branch routing

How to use the script files

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

1. **Edit the JavaEnv.bat**

   Make sure you have edited the JavaEnv.bat file to set your required working environment.
2. **Create a command-line session**

   Create a command-line session, and invoke the JavaEnv.bat to make the settings available in the current environment.
3. **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.
4. **Create an MQ queue manager**

   By default, the scripts assume this is called `CENTRAL00QM` listening on port 1414 for client channel connections.
5. **Describe the MQ queue manager**

   Edit and review the CentralQMDetails.bat file to make sure that its details match those of the MQ queue manager you have just created. All values, except the name of the machine on which the MQ queue manager sits, are defaulted in the script file.
6. **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.

7. **Review the central.tst file**

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

8. **Distribute all the scripts to all machines**

   Copy all of the scripts to all of the machines on which you will be running MQe 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 MQe queue managers and restart from this point in the instructions.

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

   The central.tst script is in a format used by the runmqsc sample program supplied with MQ. Pipe the central.tst file into runmqsc to configure your MQ queue manger For example:

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

   Use the MQ Explorer to view the resultant MQ objects that are created. **Milestone:** You have now set up your MQ system.

10. **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.

11. **Check for the test message**

    The script that creates the queue manager sends a test message to the MQ system. Use the 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 MQe gateway queue manager.

12. **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 MQe gateway queue manager it represents is also active. Closing the window closes the MQe gateway queue manager and breaks the path from the branch queue managers to the MQ queue manager.

13. **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 MQ system. Use the 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 MQe 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.

14. **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 MQe_Explorer to view the configuration." **Milestone:** You now have a branch queue manager set up.

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

    You can now use the MQe_Explorer to view the configuration.

How to use MQe_Explorer to view the configuration

To use the MQe_Explorer to view your configuration:

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

Chapter 2. Configuring MQe objects

Configuring queue managers

Introduction to configuring queue managers

The queue manager is the central component of MQe.

It provides the main programming interface for application programs, and it also owns queues, communication and MQ bridge subsystems.

Java and C differ significantly in the area of creating and deleting queue managers:

- In Java, general queue manager configuration is performed using administration messages, but creation 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 administration message to achieve the required result. The message then needs to be sent, and the administration 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* is 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 or MQeField 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 C	Read/Write
Bridge Capable	Determines if the queue manager 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
Description	A free-format textual description of this queue manager.	Yes	Yes	Read/Write
Maximum Transmission Threads	The maximum number of background transmission threads supported by this queue manager.	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

Table 12. Queue Manager attributes (continued)

Attribute	Description	Java	Native C	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 public static final 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_MaximumTransmissionThreads	maximumTransmissionThreads
fields arrayEach element contains a fields object containing {QMgr_QueueName, QMgr_QueueQMgrName, QMgr_QueueType}	QMgr_Queues	queues
ascii	QMgr_QueueStore	queueStore
ascii	QMgr_Rules	rules

C

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

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

C

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();
```

C

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 */
         }
   }
```

Inquire and inquire all

In general, when inquiring on objects in MQe, you can:

- ask for particular parameters which are of interest using `inquire`
- ask for all information using `inquireAll`.

Java

Inquire

```
//inquire

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

    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)

    } catch (Exception e) {
    System.err.println("Failure ! " + e.toString());
}
```

Inquire all

```
//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

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 an MQeQueueType:

- String: QueueQueueManager Name. Must be freed
- String: QueueName. Must be freed
- Constant string: The Java Class Name - need not be freed
- Primitive: MQeQueueType.

The Queue Info structure must be freed using the mqeMemory_free function. Please see C Programming 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

```
//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

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 new 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: 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 Configuring a Connection section for more information.

C
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 Configuring a Connection section for more information.

C
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 Configuring a Connection section for more information.

C
Lists all aliases, into a new MQeVector. These are the Alias names.

Note that when the vector is freed, its contents will automatically also 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 Configuring a Connection section for more information.

C

```
MQEBOOL isAlias;

rc =  mqeAdministrator_QueueManager_isAlias(hAdministrator,
                                            &exceptBlk,
                                            hName,
                                            &isAlias);

if (isAlias==MQE_TRUE) {
   /* name is alias */
}
```

Configuring a queue manager using memory only

This topic applies only to the Java code base.

It is sometimes required that applications have a queue manager which exists in memory only. MQe 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.

An MQe 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 MQe queue managers have the following characteristics:

- Functionally they can do everything other MQe 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 MQe 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.

Configuring local queues

Introduction

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 required result. The messages needs then to be sent, and the admin reply messages checked as required.

C

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

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

Where *action* can be one of the following:

create Create a Queue

delete Delete a Queue

update
Update the properties of a queue

inquire
Inquire 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 "Configuring with the C administrator API" on page 30.

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 code base	Read
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

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

Property	Description	Java	Native	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
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

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

Property	Description	Java	Native	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. Queue properties available in Java

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

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. Queue properties available in C

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
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 20,000ms, and a description.

Java

First of all create the `MQeQueueAdminMsg` object. This needs to be primed 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 );
```

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

C

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` would mean 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);
```

Delete a local queue

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

Java

```
/* 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 );
```

C

The deletion of a queue requires that the queue be empty of messages.

Note that 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 */
   }
```

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.

Java

To add an alias name to a queue, use the `addAlias` method on the `MQeQueueAdminMsg`.

With admin messages multiple add alias 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 13. Adding an alias to a queue in Java

C

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 */
     }
```

List aliases

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

Java

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

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);

   }
```

Remove alias

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

Java

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

/* Prime the 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 );
```

C

```
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 */
       }
```

Update

Some of the properties of a queue can be updated.

This is only those properties which are marked as writable in the table of properties.

A similar technique is used to update and inquire upon other types of queues, such as remote and home server queues.

Java

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 the 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 );
```

C

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);
```

Inquire and inquire all

It is possible to inquire the properties of queue by using the `inquire` action.

The details that are required are set.

When using the Java administration message, the administration reply message contains a fields object with the required information.

When using the C API, a structure will be filled out with the requested information.

Java

There are two ways of inquiring on a queue: `inquire` and `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, particular 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();
```

`outputFields` now contains the field `Queue_Description` only.

C

The API takes the same parameter structure that the other APIs (such as `create`) take.

To specify the elements that are of interest, set `opFlags` accordingly.

To get, for example, the queue maximum depth, expiry, and description, set `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;
}
```

Message storage adapter

A local queue uses a queue store adapter to handle its communications with the storage device. Adapters are interfaces between MQe 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 MQe 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 code base, 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, and on UNIX systems use "/". In some cases it may be possible to use either but this is dependent upon the support provided by the JVM (Java Virtual Machine) that the queue manager runs in. As well as file separator differences, some systems such as Windows use drive letters, but others such as 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.

Configuring remote queues

Introduction

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 *QueueQueueManager* is used to refer to QM_**B**, that is, the QueueQueueManager is the QueueManager upon which the LocalQueue referenced by the Remote Queue Definition resides.

In summary, remote queues are references to queues that reside on a queue manager that is remote to where the definition is. The remote queue has the same name as the target queue but the remote queue definition also 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 the *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, MQe 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.

Structures

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

```
typedef struct MQeRemoteAsyncQParms
{
    /**< Queue Parms Structure - for general parameters */

    MQeQueueParms      baseParms;

    /**< Transport Class (Read/Write) */

    MQeStringHndl      hQTransporterClass;

} MQeRemoteAsyncQParms;
```

Synchronous and asynchronous

The difference between the two types of remote queue definition is that with synchronous a message put to a remote queue definition is sent over the network in real-time and put to the queue on the remote queue manager, whereas with asynchronous the message is put to a temporary store and transmitted when a network connection becomes available.

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, MQe is no longer responsible for once-only assured delivery.

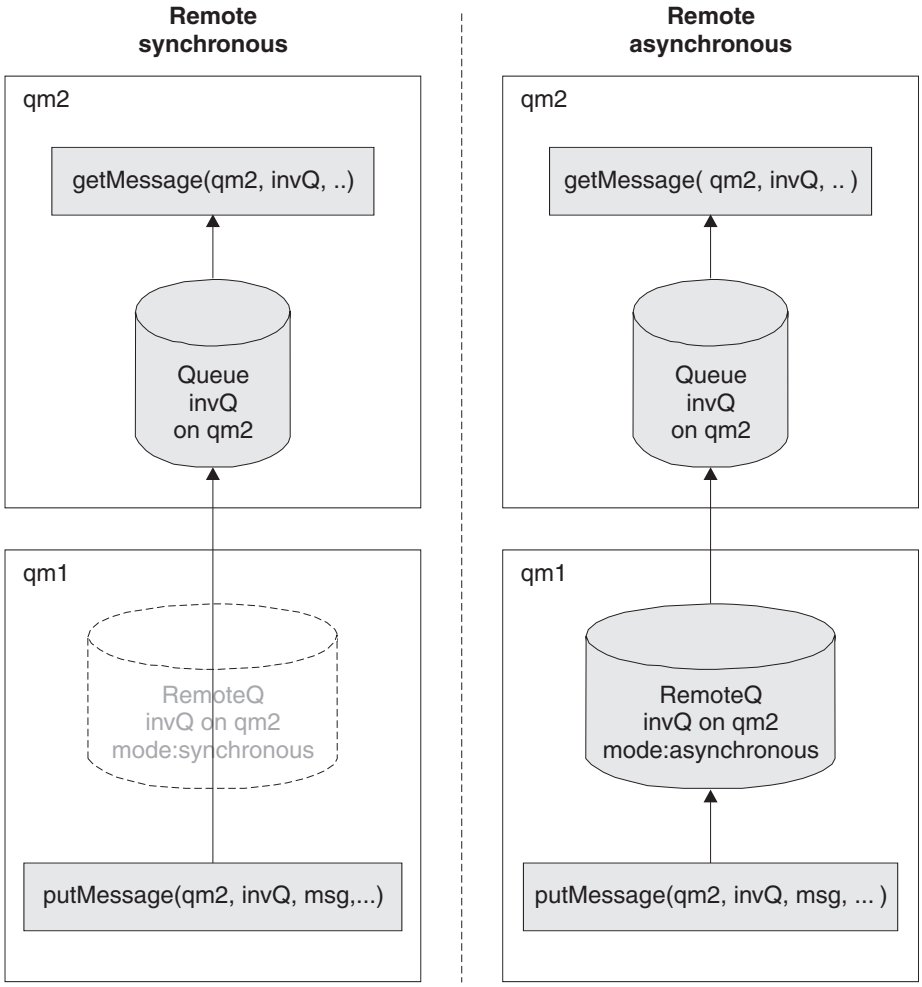
Asynchronous

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 offline. 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 code base, 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 code base, two distinct sets of APIs are used to create and administer synchronous and asynchronous remote queues.

This diagram shows an example of a remote queue set up for synchronous operation and a remote queue setup for asynchronous operation.



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

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

Creating a remote queue

The following code fragments show how to setup an administration message to create a remote queue.

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

Java

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
   * characteristics 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);
}
```

C

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
{
   /*< Queue Parms Structure for general parameters */

   MQeQueueParms     baseParms;

   /*< Transporter Class (Read/Write) */

   MQeStringHndl     hQTransporterClass;

} MQeRemoteSyncQParms;
```

Create synchronous

Java

First create the remote queue administration message.

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

Then prime the administration message, as explained in Chapter 1, “How to configure MQe objects,” on page 1.

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

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

Java

```
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

This is the C API to create an 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 this cannot be modified after the Queue has been created.

Queue aliases

The administration of aliases is the same as for LocalQueues, because 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
```

Configuring home server queues

Introduction

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..

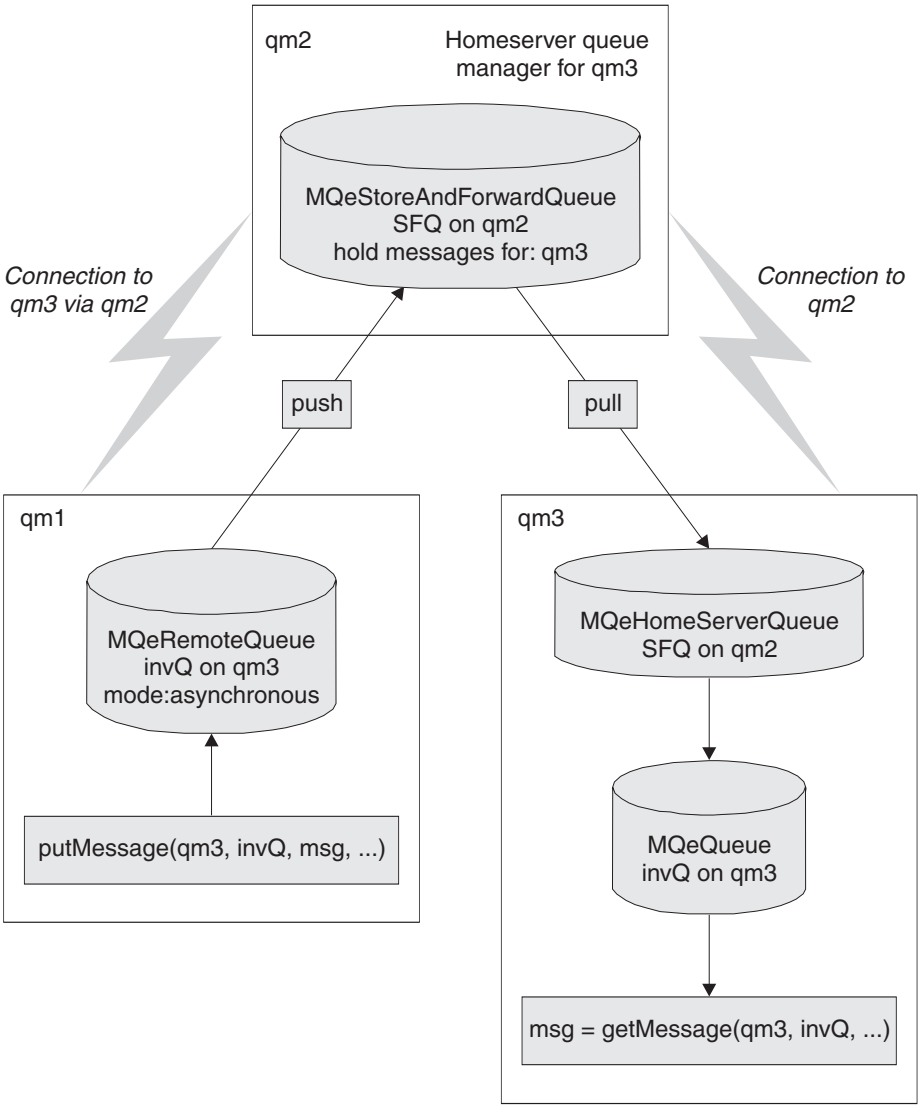


Figure 14. Home-server queue

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.

Configuration 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.

C

The C code base 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 a home server queue

Java

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

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

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

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
{

   /**<Remote Queue Parameters to be filled in  */

   MQeRemoteSyncQParms     remoteQParms;

   /**<Time Interval - for Java compatibility only*/
   MQEINT64              qTimerInterval;

} MQeHomeServerQParms;
```

Configuring store-and-forward queues

Introduction

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

A store and forward 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 Configuring home server queues - Introduction. 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.

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.

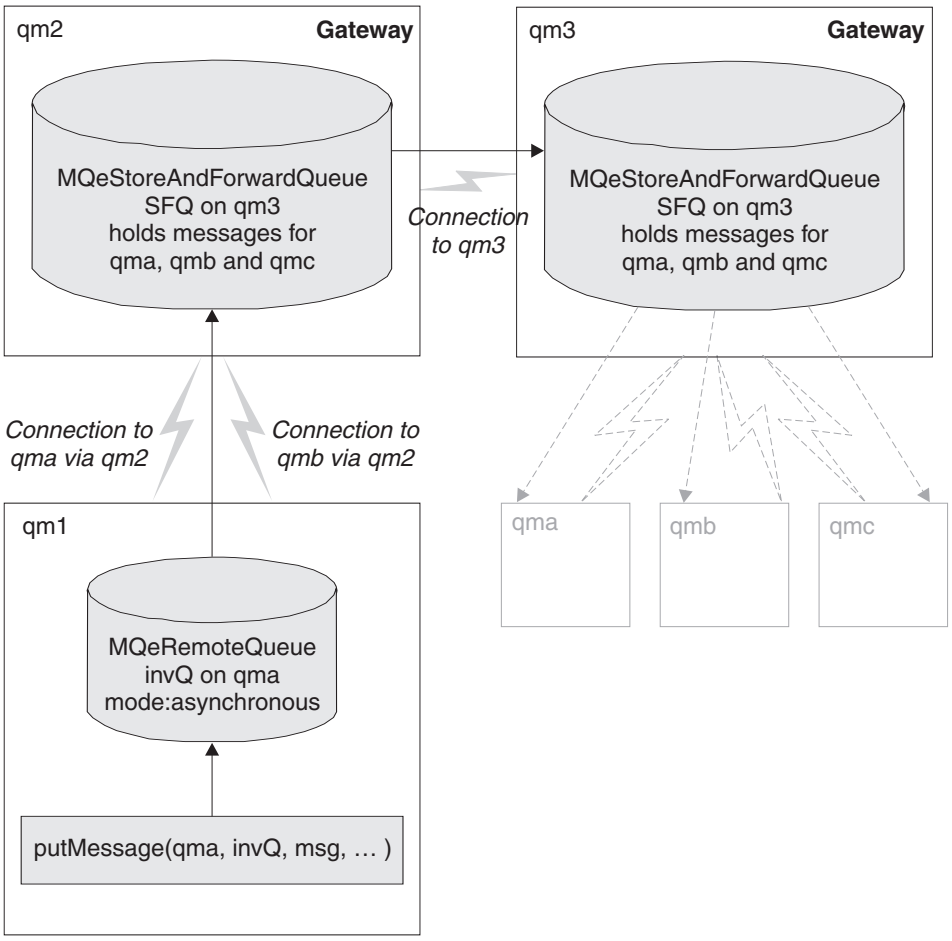


Figure 15. Store-and-forward queue

The diagram 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 the diagram, 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 Configuring home server queues - Introduction for how to set this up.

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 code base.

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. Java parameters

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

Create store and forward queue

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.

Java

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 “Configuring with messages” on page 14.

```
/* 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 );
```

After the administration message has been created, it needs to be sent to the local administration queue.

Delete store and forward queue

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.

Java

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 */

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() );
```

Add queue manager

You can add and delete queue manager names with the following actions:

- `Action_AddQueueManager`
- `Action_RemoveQueueManager`

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 methods:

- `addQueueManager()`
- `removeQueueManager()`

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 the following 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.

Java

```
/* Create an empty store and forward queue admin message and parameters field */

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);
```

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.

Java

```
/* 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);
```

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.

Java

```
/* 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);
```

Inquire

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

Java

```
/* 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);
```

Configuring connection definitions

Introduction

Connection definitions provide MQe 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 MQe 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.

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.

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.

Note:

As will be seen from the examples 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 supplied with the MQe product.

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 MQe 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 an MQ queue manager as opposed to an MQe queue manager. For further information on the Bridge functionality of MQe, refer to "Configuring bridge/gateway resources" on page 81.

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.

Configuring 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 an 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 an 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 MQe 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);
```

Configuring connection definitions 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 MQe. More information may be found abut the administration API in "Configuring with the C administrator API" on page 30, 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 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.

Creating 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,
                                         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);
```

Altering and deleting connection definitions
Altering a connection definition

As has been previously stated it is strongly 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);
```

Deleting connection definitions

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);
```

Configuring a 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.

Note: 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.

Java

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. MQe 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.

Configuring bridge/gateway resources

Introduction to the MQ bridge

This section describes how MQe can be made to interact with MQ using a `gateway`.

- A `gateway` is an MQe queue manager configured with a `bridge` that allows it to interact with MQ.
- The bridge is an MQe `object` (in the same sense that queues, connections and so on are `objects`).
- The MQ queue manager does not require any special configuration. It is referred to within MQe as the `queue queue manager`.
- The gateway runs on a machine acting as a server that can connect to another machine running MQ. The gateway cannot run on a device.
- The gateway and MQ can both be running on the same machine if required.
- In a complex setup, a gateway can have multiple bridges configured (this is very unusual).

What makes a queue manager bridge-enabled

Some MQe queue managers are capable of exchanging messages with 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 the MQ Java classes, and when the MQ bridge software is available for the JVM to load.

When an MQe queue manager is activated, it attempts to load the MQ bridge software component. If the MQe 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 MQ Java classes, or parts of the MQ bridge software are not available on the classpath.

  Changing the classpath to reference the MQ Java and MQ bridge classes, and restarting the JVM in which the MQe 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 to bridge-enable a queue manager

To use the MQ bridge you must have these two arrangements:

1. MQ Classes for Java version 5.1 or later, installed on your MQe system, and available on the classpath for JVMs to use. MQ Classes for Java is available for free download from the Web as SupportPac™ MA88. This can be downloaded free. The MQ classes for Java are also shipped with MQ software, but might not be installed depending on the options selected when 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 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;
```

2. MQe 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
   ```

Overview of configuring the bridge

The configuration of the MQ bridge requires you to perform some actions on the MQ queue manager, and some on the MQe queue manager. The bridge can be divided into two pieces:

- Configuration of resources required to route a message from MQe to MQ
- Configuration of resources required to route a message from MQ to MQe

Configuration of both types of routes is discussed in the following sections.

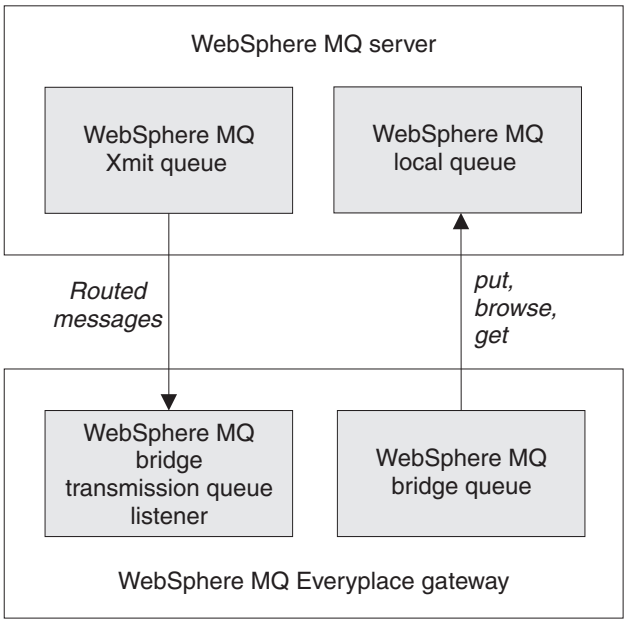


Figure 16. Bridge configuration

The bridge objects are defined in a hierarchy as shown in the following diagram:

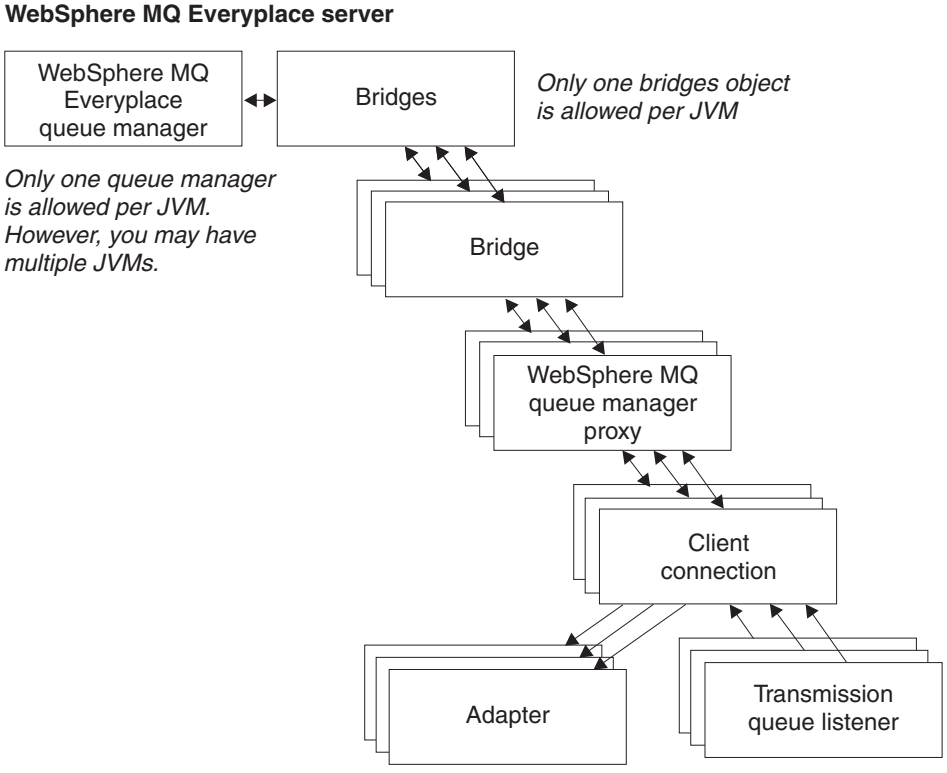


Figure 17. Bridge object hierarchy

The following rules govern the relationship between the various objects in that diagram:

- An MQe `bridges` object is associated with a single MQe queue manager.
- A single `bridges` object may have more than one `bridge` object associated with it. You might want to configure several MQ bridge instances with different routings.

- Each bridge can have a number of MQ queue manager proxy definitions.
- Each MQ queue manager proxy definition can have a number of client connections that allow communication with MQe.
- Each client connection connects to a single MQ queue manager. Each connection may use a different *server connection* on the MQ queue manager, or a different set of security, send, and receive exits, ports or other parameters.
- An MQ bridge client connection may have a number of transmission queue listeners that use that bridge service to connect to the MQ queue manager.
- A listener uses only one client connection to establish its connection.
- Each listener connects to a single transmission queue on the MQ system.
- Each listener moves messages from a single MQ transmission queue to anywhere on the MQe network, (through the MQe queue manager its bridge is associated with). So an MQ bridge can funnel multiple MQ message sources through one MQe queue manager onto the MQe network.
- When moving MQe messages to the MQ network, the MQe queue manager creates a number of *adapter* objects. Each adapter object can connect to any MQ queue manager (providing it is configured) and can send its messages to any queue. So an MQ bridge can dispatch MQe messages routed through a single MQe queue manager to any MQ queue manager.

The bridge configuration option allows an MQe queue manager to communicate with 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 MQe queue managers in a single network.

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

When moving messages in the other direction, from MQe to 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 MQe messages routed through a single MQe queue manager to any MQ queue manager, either directly or indirectly.

The bridge objects and hierarchy

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 MQe queue manager. If the MQe 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 MQe 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 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.

Bridge resource

The bridge resource is responsible for holding a number of persistent property values, and a list of 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 MQe to MQ, or vice versa, if no other transformer class has been associated with the destination queue
Heartbeat interval	The basic timing unit to be used for performing actions against bridges
Name	Name of the bridge
Run state	Status: running or stopped
Startup rule class	Rule class used when the bridge is started
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 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 MQ to MQe) 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 MQe Java Programming Reference. This applies both to MQe and MQ target queues.

MQ queue manager proxy

The MQ queue manager proxy holds the properties specific to a single MQ queue manager. The proxy properties are shown in the following table:

Table 21. MQ queue manager proxy properties

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

Client connection resource

The client connection definition holds the detailed information required to make a connection to an 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 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
MQ password*	Password for use by the Java client
MQ port*	IP port number used to create connections to the MQ queue manager via the Java client classes. If not specified then the MQ queue manager is assumed to be on the same machine as the bridge and the Java bindings are used
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
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
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
MQ user ID*	user ID for use by the Java client
Client connection service name	Name of the server connection channel on the 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 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 MQe to 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 MQ to MQe , 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 MQe 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 MQ system. When an 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 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 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`.

Transmit queue listener resource

The listener moves messages from MQ to MQe.

Table 23. Listener properties

Property	Explanation
Class	Listener class
Dead letter queue name	Queue used to hold messages from MQ to MQe that cannot be delivered
Listener state store adapter	Class name of the adapter used to store state information
Listener name	Name of the 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 an MQ message to MQe
Undelivered message rule class	Rule class used to determine action when messages from MQ to MQe cannot be delivered
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 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 `examples.mqbridge.administration.programming.AdminHelperMQTransmitQueueListener`

The `undelivered message rule class` determines what action is taken when a message from MQ to MQe cannot be delivered. Typically it is placed in the `dead letter queue` of the 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 MQe system or in the sync queue of the MQ system.

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

Bridge queue

An MQ bridge queue definition can contain the following attributes.

Table 24. 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
MQ queue manager proxy	Name of the MQ queue manager to which the message should first be sent
MQ bridge	Name of the bridge to convey the message to MQ
Name	Name by which the remote MQ queue is known to MQe
Owning queue manager	Queue manager owning the definition
Priority	Priority to be used for messages, unless overridden by a message value
Remote MQ queue name	Name of the remote MQ queue
Rule	Rule class used for queue operations
Queue manager target	MQ queue manager owning the queue
Transformer	Name of the transformer class that converts the message from MQe format to MQ format
Type	MQ bridge queue

More detail of each property can be found in the 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 MQe that the message is sent initially, to the time when the message is passed to the MQ bridge queue, the message is protected with at least the queue level of security. These security levels are *not* applicable when the MQ bridge queue passes the message to the 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.

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

Applications make use of MQ bridge queues like any other MQe 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 MQ bridge queue, and the queue manager name parameter is the name of

the MQ queue manager. However, in order for this queue manager name to be accepted by the local MQe server, a connection definition with this 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 MQ bridge queues. It is not possible to get or browse messages from MQ bridge queues that point to MQ remote queue definitions. Nor is it possible to use nonzero Confirm IDs on MQ bridge queue gets. This means that the getMessage operation on 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 MQ.

Administration of the MQ bridge is handled in the same way as the administration of a normal MQe queue manager, through the use of administration messages. New classes of messages are defined as appropriate to the queue.

Naming recommendations for interoperability with MQ

To create an MQe network that can interoperate with an MQ network and avoid problems, adopt the same limitations in naming convention for both systems. The following differences are relevant:

- MQ queue and queue manager names can have a forward slash (/) character. This character is not valid in MQe object names. Do not use this character in the name of any MQ queue or queue manager.
- MQ queue and queue manager names have a limit of 48 characters, but MQe names have no length restrictions. Do not define MQe queues or queue managers with names that contain more than 48 characters.
- MQ queue names can have leading or trailing period (.) character. This is not allowed in MQe. Do not define any MQ queue or queue manager with a name that starts or ends with this character.
- Name queue managers uniquely, such that a queue manager with the same name does not exist on the MQe network and the MQ network.

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

Configuring a basic MQ bridge

To configure a very basic installation of the MQ bridge complete the following steps:

1. Make sure you have an MQ system installed and that you understand local routing conventions, and how to configure the system.
2. Install MQe on a system (it can be on the same system as MQ)
3. If MQ Classes for Java is not already installed, download it from the Web and install it on the MQ system.
4. Set up your MQe system and verify that it is working properly before you try to connect it to MQ.
5. Update the MQe_java\Classes\JavaEnv.bat file so that it points to the Java classes that are part of the 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 MQ queue managers are going to talk to which MQe queue managers.
7. Decide on a naming convention for MQe objects and MQ objects and document it for future use.
8. Modify your MQe system to define an MQ bridge on your chosen MQe server. See Java Programming Reference for information on using examples.mqbridge.awt.AwtMQBridgeServer to define a bridge.
9. Connect the chosen MQ queue manager to the bridge on the MQe server as follows:
   - On the MQ queue manager:

– Define one or more Java server connections so that MQe can use the 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 MQe to use to provide assured delivery to the MQ system. You need one of these for each server connection that the MQe system can use.

- On the MQe server:
  a. Define an MQ queue manager proxy object which holds information about the MQ queue manager. This involves the following steps:
     1) Collect the Hostname of the MQ queue manager.
     2) Put the name in the MQ queue manager proxy object.
  b. Define a Client Connection object that holds information about how to use the MQ Classes for Java to connect to the server connection on the 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 MQ queue manager.

10. Modify your configuration on both MQe and MQ to allow messages to pass from MQ to MQe.
    a. Decide on the number of routes from MQ to your MQe 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 MQ system. DO NOT define a connection for these transmission queues.
       2) For each route create a matching transmission queue listener on your MQe system.
       3) Define a number of remote queue definitions, (such as remote queue manager aliases and queue aliases) to allow MQ messages to be routed onto the various MQe-bound transmission queues that you defined in step b. 1.
11. Modify your configuration on MQe to allow messages to pass from MQe to MQ:
    a. Publish details about all the queue managers on your MQ network you want to send messages to from the MQe network. Each MQ queue manager requires a connections definition on your MQe server. All fields except the Queue manager name should be null, to indicate that the normal MQe communications connections should not be used to talk to this queue manager.
    b. Publish details about all the queues on your MQ network you want to send messages to from the MQe network. Each MQ queue requires an MQ bridge queue definition on your MQe server. This is the MQe equivalent of a `DEFINE QREMOTE` in MQ.
       - The queue name is the name of the MQ queue to which the bridge should send any messages arriving on this MQ bridge queue.
       - The queue manager name is the name of the MQ queue manager on which the queue is located.
       - The bridge name indicates which bridge should be used to send messages to the MQ network.
       - The MQ queue manager proxy name is the name of the MQ queue manager proxy object, in the MQe configuration, that can connect to an MQ queue manager.
       - The 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 MQ and MQe systems to allow messages to flow. The MQ system client channel listener must be started. All the objects you have defined on the MQe must be started. These objects can be started in any of the following ways:
    - Explicitly using the Administration GUI described in MQe Configuration Guide.
    - Configuring the rules class, as described in MQe System Programming Guide, to indicate the startup state (running or stopped), and restarting the MQe 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 MQe to MQ, start the client connection objects in MQe.
- To allow messages to pass from MQ to MQe, start both the client connection objects, and the relevant transmission queue listeners.

13. Create transformer classes, and modify your MQe configuration to use them. A transformer class converts messages from MQ message formats into an MQe message format, and vice versa. These format-converters must be written in Java and configured in several places in the MQ bridge configuration.
    a. Create transformer classes
        - Determine the message formats of the MQ messages that need to pass over the bridge.
        - Write a transformer class, or a set of transformer classes to convert each MQ message format into an MQe message. Transformers are not directly supported by the C bindings.
    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 MQ bridge queue definition. Use the administration GUI to update the *transformer* field of each MQ bridge queue in the configuration.
    d. You can specify a non-default transformer for each 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.

Using MQe administration messages and MQ PCF messages

PCF messages are administration messages used by MQ queue managers. SupportPac MS0B: "MQSeries® Java classes for PCF" contains Java code, which supplies PCF message support. This code is available as a free download.

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 MQ resources. When PCF messages are combined with MQe administration messages, complete programmatic configuration of bridge resources, and corresponding resources on an MQe 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` demonstrates how to do this. This example code has been added to the `examples.awt.AwtMQeServer` program, such that clicking menu **View->Connect local MQ default queue manager** will:

- Ensure that a bridge object exists, creating one as required.
- Query properties from the default MQ queue manager.
- Attempt to connect that queue manager to the currently running MQe queue manager.
- Ensure a proxy object representing the default MQ queue manager exists, creating one if necessary.
- Ensure an MQe client connection exists, and that a corresponding MQ server connection channel exists also, creating these resources if necessary.
- Ensure a 'sync queue' exists on the MQ queue manager.
- Ensure a transmit queue on MQ exists, and create if necessary.
- Ensure a matching MQ transmit queue listener exists in the configuration of the current MQe queue manager, creating one if necessary.
- Ensure all the bridge resources are started.
- Ensure a test queue on the MQ queue manager exists, creating one if necessary.
- Ensure a matching MQe 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.

Bridge configuration example

This section describes an example configuration of 4 systems.

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.

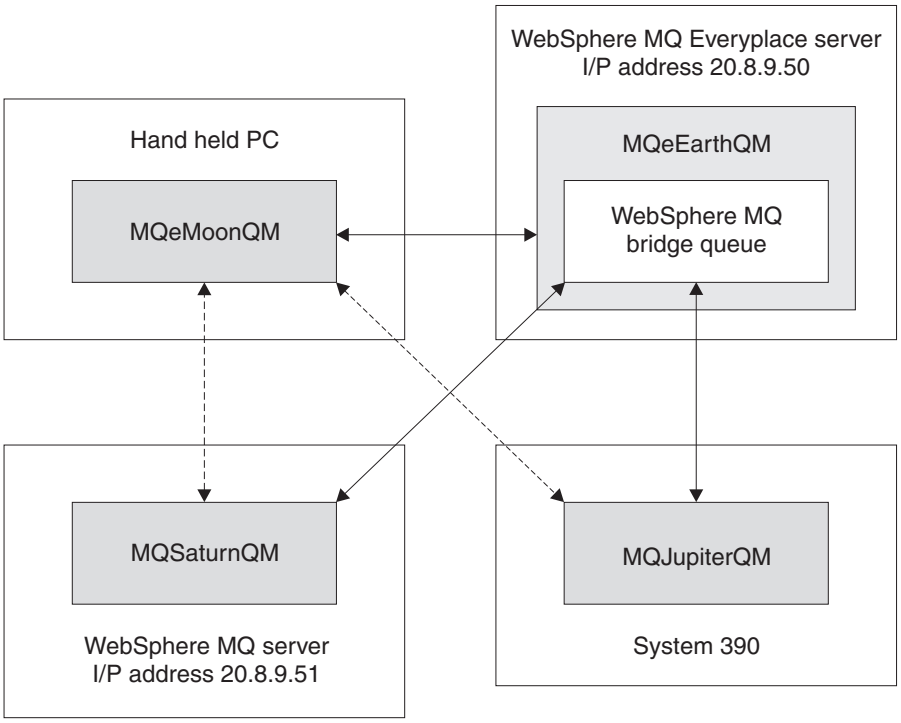


Figure 18. Configuration example

The four systems are:

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

MQeEarthQM
This is on a Windows 2000 machine, with an IP address of 20.8.9.50 This is an MQe gateway (server) queue manager.

MQSaturnQM
This is an MQ queue manager, installed on a Windows NT platform. The IP address is 20.8.9.51

MQJupiterQM
This is an MQ queue manager, installed on a System/390® platform.

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 MQeMoonQM
- MQeEarthQ on MQeEarthQM
- MQSaturnQ on MQSaturnQM
- MQJupiterQ on MQJupiterQM

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

MQeMoonQM to/from MQeEarthQM

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.

MQeEarthQM to MQeMoonQM

Because 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, its 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.

MQeEarthQM to MQSaturnQ

On MQeEarthQM:

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

   Bridge name: MQeEarthQMBridge
2. Define an **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 **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.

MQeEarthQM to MQJupiterQ

On MQeEarthQM:

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

ConnectionName: MQeJupiterQM
Channel: null
Adapter: null

2. Define an **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.

MQeMoonQM 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.

MQSaturnQM to 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 MQe transmission queue listener MQeEarth.XMITQ gets them, and moves them onto the MQe network.

MQSaturnQM to 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 MQe transmission queue listener MQeEarth.XMITQ gets them, and posts them onto the MQe 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.

MQJupiterQM to MQeMoonQ

On MQJupiterQM:

Set up **remote queue manager aliases** for the MQeEarthQM and MQeMoonQM to route messages to MQSaturnQM using normal MQ routing techniques.

Administration of the bridge

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 MQ bridge queue references a bridge administered object that is *stopped*, it is unable to convey an MQe message onto the MQ network until the bridge, 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.

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,

A sequence of:

```
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`.

Bridge considerations when stopping an MQ queue manager

Before you stop an MQ queue manager, issue a `stop` administration message to all the MQ queue-manager-proxy bridge objects. This stops the MQe network from trying to use the MQ queue manager and possibly interfering with the shutdown of the 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 MQ queue-manager-proxy bridge object before you shut the MQ queue manager, the behavior of the MQ shutdown and the MQ bridge depends on the type of MQ queue manager shutdown you choose, immediate shutdown or controlled shutdown.

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

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

Any MQ bridge queues that are active retain a broken connection to the 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 MQ bridge queue is told to perform some action, such as put a message to MQ, that attempts to use the broken connection. The `putMessage` operation fails and the broken connection is closed.

When an MQ bridge queue has no connection, the next operation on that queue causes a new connection to be obtained. If the MQ queue manager is not available, the operation on the queue fails synchronously. If the 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 MQ queue manager is established, and the operation executes as expected.

Controlled shutdown: Stopping an 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 MQ bridge transmission queue listeners notice that the MQ system is quiescing, and stop with a relevant warning.

Any MQ bridge queues that are active retain a connection to the 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 MQ queue manager completes.
- The MQ bridge queue is told to perform some action, such as put a message to MQ, that attempts to use the broken connection. The `putMessage` operation fails, the broken connection is closed, and the controlled shutdown of the 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 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 MQ queue manager is closed, the MQ controlled shutdown completes.

Administered objects and their characteristics

This section describes the characteristics of the different types of administered objects associated with the MQe 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.

Characteristics of bridges objects

Refer to Java Programming Reference for information on the `com.ibm.mqe.mqbridge.MQeMQBridgesAdminMsg`.

Characteristics of bridge objects

Refer to Java Programming Reference for information on the `com.ibm.mqe.mqbridge.MQeMQBridgeAdminMsg`.

Characteristics of MQ queue manager proxy objects

Refer to Java Programming Reference for information on the `com.ibm.mqe.mqbridge.MQeMQQMgrProxyAdminMsg`.

Characteristics of client connection objects

Refer to Java Programming Reference for information on the `com.ibm.mqe.mqbridge.MQeClientConnectionAdminMsg`.

Characteristics of MQ transmission queue listener objects

Refer to Java Programming Reference for information on the `com.ibm.mqe.mqbridge.MQeListenerAdminMsg`.

Configuring a bridge for optimal throughput

Using MQe, you can create a gateway to allow messages to flow to MQ. You will need to create certain MQe objects to configure a gateway. These objects include a bridge, queue manager proxy, client connection, bridge queue, and optionally a transmission queue listener if messages are also to be sent from MQ back to MQe.

In a standard configuration, a single client connection is created along with a single bridge queue. This is sufficient to start sending messages from MQe to MQ. If there are a large number of clients connecting to the gateway, the number of messages sent per second through to MQ decreases.

Gateway configuration

As Figure 19 on page 100 shows, if there are multiple clients connecting to the gateway queue manager through the communications listener, there may be a bottleneck created by the bridge queue. To alleviate this problem, you can create multiple bridge queues, client connections, server connection channels, and sync queues.

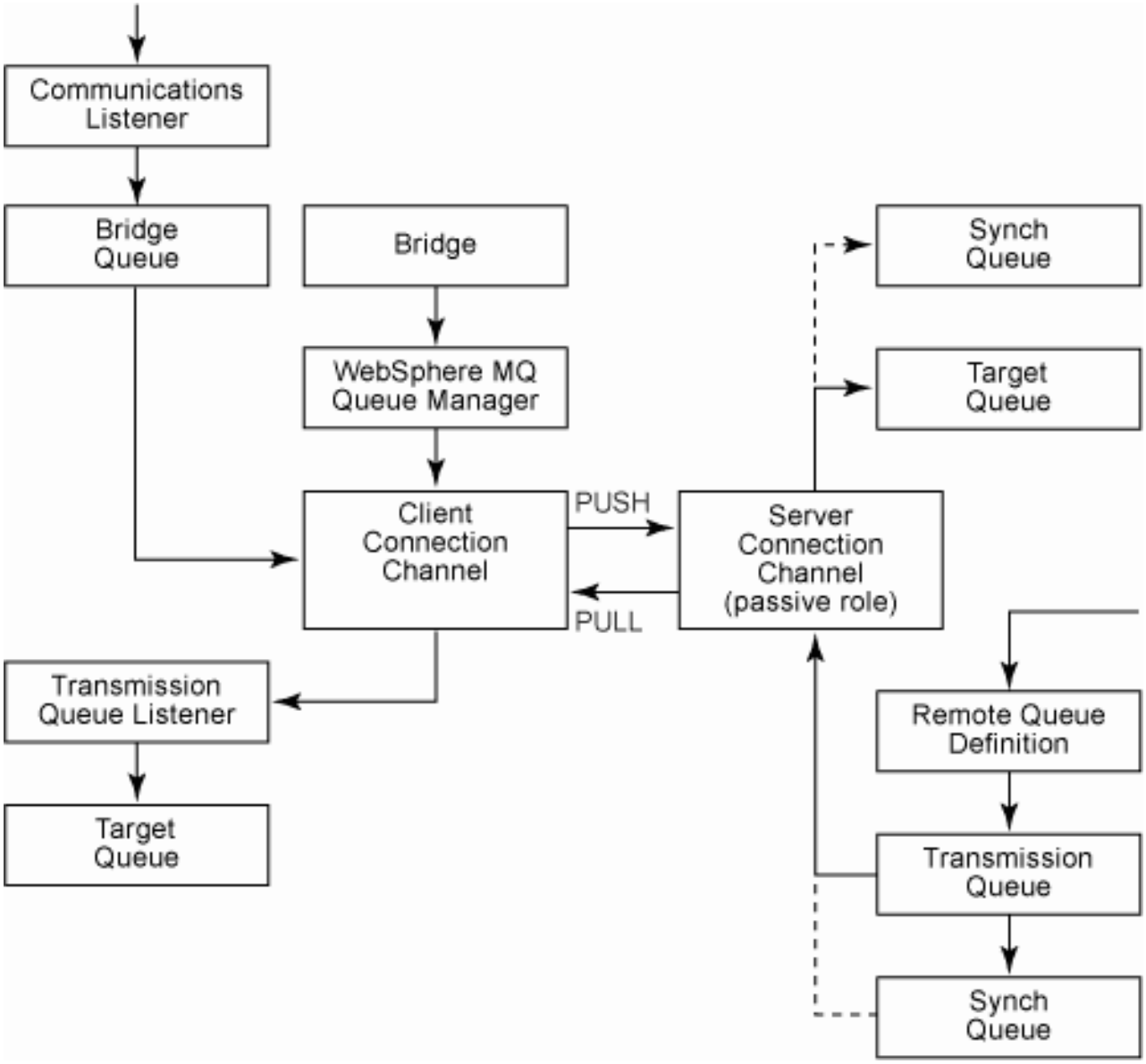


Figure 19. Standard configuration of a MQe gateway

Care needs to be taken on deciding the number of bridge queues to create as there is a point at which the performance of the gateway decreases with an increased number of bridge queues.

Finding and creating the optimal configuration

To find the optimal configuration for your setup, you need to create the MQe objects and perform tests to measure the throughput of messages. This empirical testing should ideally take place on a similar network to the production network, using production type message data.

Use the SupportPac MQe_Script to create tests that are both easy to configure and repeatable. This SupportPac is a command line based tool for creating and administering MQe resources. MQe_Script embeds a scripting language called TCL that you can use to write intelligent scripts that add logic and control. You can download this support from the from the Business Integration SupportPacs Web site:http://www.ibm.com/software/integration/support/supportpacs . To access the download from this site, select the WebSphere® MQ Everyplace® product from the list box.

Using MQe_Script, you can write scripts that create all the necessary objects on the gateway, and optionally the clients. Parameters can be passed into a script in such a way that the script takes in the number of required bridge queues. As a result, the bridge queues and all the other relevant MQe objects are created.

By using this method to create MQe configurations, you can quickly and easily experiment with the number of bridge queues required to achieve the desired throughput. Once you write the script, no extra programming is required.

MQe_Script can even be used to modify any existing configurations that may exist.

Although MQe_Script does not modify MQ objects, these can be created using MQSC commands. MQe_Script does come with an example MQSC script showing how to create the MQ objects necessary to communicate with an MQ Everyplace gateway .

Setting up the gateway

Follow these steps to set up the gateway:

- The gateway requires a single instance of a bridge object and a single instance of a WebSphere MQ queue manager proxy object.
- Next, you need to create the client connection objects. How many client connection objects you need to create will depend on how many bridge queues are required.
- Finally, you need to create the bridge queues with each one using a different client connection.

Setting up the clients

Each client must have a remote queue definition to the MQ queue. If multiple bridge queues are being used on the gateway, an alias is needed on the remote queue definition so that any application putting messages using a client gets the correct path to the MQ queue. This also helps as applications do not have to be aware of the clients putting to different queues.

For example, if three clients existed, each putting to separate bridge queues, *b1*, *b2*, and *b3*, all remote queue definitions could have an alias of *b*. Any application putting to the clients can then simply put to queue b. This allows changes in the underlying network without having to make any changes to the application.

To utilize all the bridge queues created on the gateway, and therefore achieve optimal message throughput, the bridge queues should be equally divided amongst all the clients. Therefore, if there are 50 bridge queues and 5000 clients, 100 clients should create a remote queue definition to *bridgeq1*, another 100 should create a remote queue definition to *bridgeq2* and so on.

Creating the gateway

This section provides the basic steps and code snippets to create a gateway. The MQe_Script commands used in this section are provided as a full script along with any undefined variables in “Sample script to create a gateway” on page 104.

1. Create a queue manager.

   To create a basic queue manager, provide a name and disk location. The disk location is where the registry will be saved to and any messages for queues.

   ```
   mqe_script_qm -create -qmname $GATEWAYQM -qmpath $PATH
   ```

2. Start the queue manager.

   The command used here does not specify a queue manager name to load. This is because if used directly after a create command, the queue manager details are cached. For more information on how to load a previously created queue manager, please see the documentation accompanying MQe_Script when you download the SupportPac.

   ```
   mqe_script_qm –load
   ```

3. Create a listener.

The listener must be given a name and a port to listen on. Optionally, the type of adapter can be specified if it is something other than the default (TCPIP HTTP).

```
mqe_script_listen -create -listenname $LISTENER -port $GATEWAYPORT
```

4. Start the listener.

   By default listeners are not started after they are created. Once the queue manager has been stopped and restarted however, the listener will then start automatically.

   ```
   mqe_script_listen –start -listenname $LISTENER
   ```

5. Create a bridge.

   An arbitrary name is needed for the bridge object. This acts as the parent for all the other bridge-related objects.

   ```
   mqe_script_bridge -create -bridgename $BRIDGE
   ```

6. Create a queue manager proxy.

   The queue manager proxy must be associated with the bridge object previously created. It must also be named according to the name of the MQ queue manager with which the connection will take place. Finally, the IP address or hostname of the machine on which the MQ queue manager is defined is required.

   ```
   mqe_script_mqproxy -create -proxyname $PROXY -bridgename $BRIDGE -hostname $ADDRESS
   ```

7. Create a connection definition to the MQ queue manager.

   A special MQ connection must be created to define the MQ queue manager. This needs to have the same name as the MQ queue manager.

   ```
   mqe_script_condef -create -cdname $PROXY -type mq
   ```

8. Create multiple client connection channels.

   These must be associated with the queue manager proxy and bridge objects previously created. They must have the same name as a server connection channel on the MQ queue manager. Finally, the MQ sync queue, which it will use, must be defined. This will be different for each client connection channel. If the MQ queue manager is not listening on the default port of 1414, the port number must also be defined.

   ```
   mqe_script_mqconn -create -clientconnname $CC$j -proxyname $PROXY -bridgename $BRIDGE -syncqname
       $SYNCQ$j -port $PORT
   ```

9. Optionally, to receive messages back from MQ, create multiple transmission queue listeners.

   If a transmission queue listener is required, it must be associated with the client connection channel, queue manager proxy and bridge objects previously created. It must have the same name as the transmission queue on the MQ queue manager.

   ```
   mqe_script_mqlisten -create -listenname $LISTEN -clientconnname $CC$j -proxyname
       $PROXY -bridgename $BRIDGE
   ```

10. Create multiple bridge queues.

    A name must be provided for the bridge queue. Under normal circumstances, the bridge queue names reflect the name of the MQ destination queue. An alternative way of linking the bridge queue to the MQ queue is to use an additional parameter, the MQ Queue Name, which allows the name of the bridge queue to be something different. It is this latter approach that needs to be taken when defining multiple bridge queues. Each bridge queue must be associated with an individual client connection. In other words, there is a one-to-one relationship. The bridge queue also needs to be associated with the bridge and the queue manager proxy, thus linking the bridge queue to a specific MQ queue manager, queue and connection.

    ```
    mqe_script_bridgeq -create -qname $BRIDGEQ$j -bridgename $BRIDGE -destination $PROXY
        -mqqname $REALBRIDGEQ -clientconnname $CC$j
    ```

11. Start the bridge.

    Starting the bridge will by default start all the child objects too. If you have created transmission queue listeners and your MQ queue manager is not contactable, this may result in the transmission queue listener failing to start. This is something to watch out for as the command may come back as

successful even if all the child objects were not started. On the other hand, an error may be thrown and you may wish to ignore it and carry on with the script. Each of the child objects can be started individually if preferred.

```
mqe_script_bridge -start -bridgename $BRIDGE
```

Steps to create the client

The basic steps to create a client are defined below. The MQe_Script commands to accompany these are provided along with any undefined variables in "Sample script to create a client" on page 106.

1. Create a queue manager.

   Provide a name and disk location.

   ```
   mqe_script_qm –create –qmname $CLIENT –qmpath $PATH
   ```

2. Start the queue manager.

   See the code example below.

   ```
   mqe_script_qm –load
   ```

3. Create a connection definition to the gateway queue manager.

   The connection definition must have the same name as the gateway queue manager. Define the port on which the gateway is listening and, if the gateway is listening on an adapter other than the default, this must also be defined.

   ```
   mqe_script_condef –create –cdname $GATEWAYQM –port $PORT –address $ADDRESS
   ```

4. Create a via connection definition to the MQ queue manager.

   The client must know about the MQ queue manager to put messages to its queues. A via connection can be created where the name of the connection is the MQ queue manager name and the via name is the gateway queue manager.

   ```
   mqe_script_condef –create –cdname $MQNAME –viaqmname $GATEWAYQM
   ```

5. Create a remote queue definition to the bridge queue and add an alias.

   The name of the remote queue definition must match one of the bridge queues defined on the gateway. The queue manager name of the MQ queue manager must also be defined.

   To use multiple bridge queues, where the bridge queue name is not the same as the real MQ queue name, aliases must be used. An application can not use a bridge queue name on the gateway as the queue name of the MQ queue because that queue does not exist on the MQ queue manager.

   Also, the remote definition of the queue can not be called the real MQ queue name as that reference does not exist on the gateway queue manager. It is therefore useful to add the actual name of the MQ queue as an alias to the remote queue definition so that applications know exactly where the message should be sent to.

   ```
   mqe_script_sproxyq –create –qname $BRIDGEQ$BRIDGEQNUM  –destination $MQNAME –alias $REALBRIDGENAME
   ```

   When using MQe_Script, the alias can be added at the time of creation or as an update in a later stage if the remote queue definition already exists.

6. Test the connection by sending a message from the client to the MQ queue manager.

   For applications to make the same call, independent of which client they are putting messages to, the message can be put using the alias of the remote queue definition.

   ```
   mqe_script_msg –put –qname $REALBRIDGENAME –qmname $MQNAME
   ```

Tips on writing a script

When writing a script, it is often easier to define a set of variables at the beginning of the script. This makes any changes to the naming conventions or number of MQe objects easier to manage.

In TCL variables are defined using the set command and referenced using a $ in front of the variable name.

It is also useful to define variables in capital letters so that they can easily be identified within a script.

```
set PATH "C:\\MQeScript\\gateway"
```

When calling MQe_Script commands, they can be defined on their own or TCL control structures can be added to provide feedback on the success of the command and potentially exit the script if errors occur.

One method of checking the success of a command is to use an if / else block with a catch command.

```
if { [catch {mqe_script_qm -create -qmname $GATEWAYQM -qmpath
$PATH} error] } {
           puts "An error occurred creating queue manager";
            puts "The reason was: $error"
           exit
} else {
      puts "Queue manager created"
}
```

As the above code snippet shows, the puts command is used to print text to the screen and the exit command stops the execution of the script. If any errors are thrown, the error text is saved in the variable named “error” and can then be accessed.

If a script is being written to create a number of bridge objects defined by a variable, the creation of those objects can be placed inside a loop. The easiest method of creating multiple objects is to have a standard name for each of the objects then add a number to each of them so they are unique.

```
for {set j 1} {$j <= $QNUM} {incr j} {

      #create all the client connections

mqe_script_mqconn -create -clientconnname $CC$j -proxyname $PROXY -bridgename $BRIDGE
      -syncqname $SYNCQ$j -port $PORT

}
```

The above snippet shows how names can be created using a loop variable. $CC is already defined as a client connection name prefix and $SYNCQ as a sync queue name prefix.

The code snippet also introduces the use of the # to define a comment.

In order to supply MQe_Script with a script file, it must be saved with a .tcl extension and the command to supply a script is the source command.

```
source {C:\MQeScript\gatewayscript.tcl}
```

For more information on TCL and writing scripts, refer to the documentation accompanying MQe_Script.

Sample script to create a gateway

```
set PATH "C:\\MQeScript\\gateway"
set ADDRESS 127.0.0.1
set GATEWAYQM gatewayqm
set LISTENER listener
set GATEWAYPORT 1881
set PROXY QM_jane
set BRIDGE bridge
set REALBRIDGEQ mqlocalq
set CC FOR.GATEWAYQM.
set LISTEN togateway
set BRIDGEQ bridgeq
set SYNCQ SYNC.QUEUE.
set PORT 1414 set QNUM 50

#create the gateway queue manager

if { [catch {mqe_script_qm -create -qmname $GATEWAYQM -qmpath $PATH} error] } {
   puts "An error occurred creating queue manager";
   puts "The reason was: $error"
   exit
```

```
} else {
     puts "Queue manager created"
}

#load the queue manager

if { [catch {mqe_script_qm -load} error] } {
    puts "An error occurred loading queue manager";
   puts "The reason was: $error"
   exit
} else {
     puts "Queue manager loaded"
}

#create the listener

if { [catch {mqe_script_listen -create -listenname $LISTENER -port
$GATEWAYPORT} error] } {
   puts "An error occurred creating a listener";
   puts "The reason was: $error"
   exit
} else {
      puts "Listener created" }

#create a bridge

if { [catch {mqe_script_bridge -create -bridgename $BRIDGE} error] }
{
            puts "An error occurred creating bridge";
            puts "The reason was: $error"
            exit
} else {
      puts "Bridge created"
}

#create a mq proxy

if { [catch {mqe_script_mqproxy -create -proxyname $PROXY -bridgename $BRIDGE
            -hostname $ADDRESS} error] } {
            puts "An error occurred creating proxy";
            puts "The reason was: $error"
            exit
} else {
      puts "MQ queue manager proxy created"
}

#create a connection to the WebSphere MQ queue manager

if { [catch {mqe_script_condef -create -cdname $PROXY -type mq} error] } {
            puts "An error occurred creating connection for the MQ queue manager";
            puts "The reason was: $error"
            exit
} else {
      puts "Connection to MQ queue manager created"
}

#create the client connections, listeners and bridge queues

for {set j 1} {$j <= $QNUM} {incr j} {
      #create all the client connections

if { [catch {mqe_script_mqconn -create -clientconnname $CC$j -proxyname $PROXY
-bridgename $BRIDGE -syncqname $SYNCQ$j -port $PORT} error] } {
            puts "An error occurred creating client connection $CC$j";
            puts "The reason was: $error"
            exit
} else {
```

```
            puts "client connection created"
}

#create all the listeners on the new client connections

if { [catch {mqe_script_mqlisten -create -listenname $LISTEN -clientconnname $CC$j
-proxyname $PROXY -bridgename $BRIDGE} error] } {
            puts "An error occurred creating listener $LISTEN on client connection $CC$j";
            puts "The reason was: $error"
            exit
} else {
            puts "listener created"
}

#create all the bridge queues

if { [catch {mqe_script_bridgeq -create -qname $BRIDGEQ$j -bridgename $BRIDGE
-destination $PROXY -mqqname $REALBRIDGEQ -clientconnname $CC$j} error] } {
            puts "An error occurred creating bridge queue $BRIDGEQ ";
            puts "The reason was: $error"
            exit
            } else {
                   puts "bridge queue created"
            }
}

# We've finished the script... let's close the queue manager

if { [catch {mqe_script_qm -unload} error] } {
            puts "Failed to stop the queue manager"
            puts "The reason was: $error"
            exit }
else {
            puts "Queue manager stopped"
}
puts "CreateGatewayQM script completed successfully"
exit 0
```

Sample script to create a client

This script only shows the basic MQe_Script commands to save duplication. Control structures could be placed around the commands or they could be run as is.

For this script to run successfully, the queue manager created by the gateway script must be running as must the MQ queue manager.

```
set CLIENT client1
set PATH "C:\\MQeScript\\client"
set ADDRESS 127.0.0.1
set GATEWAYQM gatewayqm
set PORT 1881
set BRIDGEQ bridgeq
set BRIDGEQNUM 1
set REALBRIDGENAME mqlocalq
set MQNAME QM_jane

mqe_script_qm –create –qmname $CLIENT –qmpath $PATH

mqe_script_qm –load

mqe_script_condef –create –cdname $GATEWAYQM –port $PORT –address $ADDRESS

mqe_script_condef –create –cdname $MQNAME –viaqmname $GATEWAYQM

mqe_script_sproxyq –create –qname $BRIDGEQ$BRIDGEQNUM –destination $MQNAME –alias $REALBRIDGENAME

mqe_script_msg –put –qname $REALBRIDGENAME –qmname $MQNAME
```

Performance increase

By creating an MQe network that uses multiple bridge queues, the number of messages per second that can be sent through the gateway could be dramatically increased.

The number of messages sent through the gateway is not only dependant on the configuration of your gateway, but on other issues such as the type of messages being sent, network configuration, and hardware.

In one test scenario, it was found that when approximately 1350 clients were connecting to a gateway, the increase in the number of messages being sent per second between a single bridge queue and 10 bridge queues was 34 fold.

This rose to an 86-fold increase when comparing a single bridge queue to 50 bridge queues but comparing a single bridge queue to 1350 bridge queues the increase dropped back to 34 fold.

Note this increase is only related to messages being sent to MQ and not messages received from MQ.

Conclusion

Using multiple bridge queues and client connection objects can dramatically increase the throughput of messages from MQe to MQ. However, there are no concrete figures for these performance increases since it is dependant on more than the MQe configuration.

MQe_Script can be used to create these configurations with the ability to make simple changes to affect how many objects are created which is an important part of achieving the optimal number for your individual setup.

Handling undeliverable messages

The MQ bridge's transmission queue listener acts in a similar way to an MQ channel, pulling messages from an MQ transmission queue, and delivering them to the MQe network.

It follows the MQe 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 MQ queue, on the *sending* queue manager.

Bridge National Language Support

This section describes how the MQ bridge handles messages flowing between MQSeries systems that use different national languages. The following diagram depicts the flow of a message from an MQe client application to an MQ application.

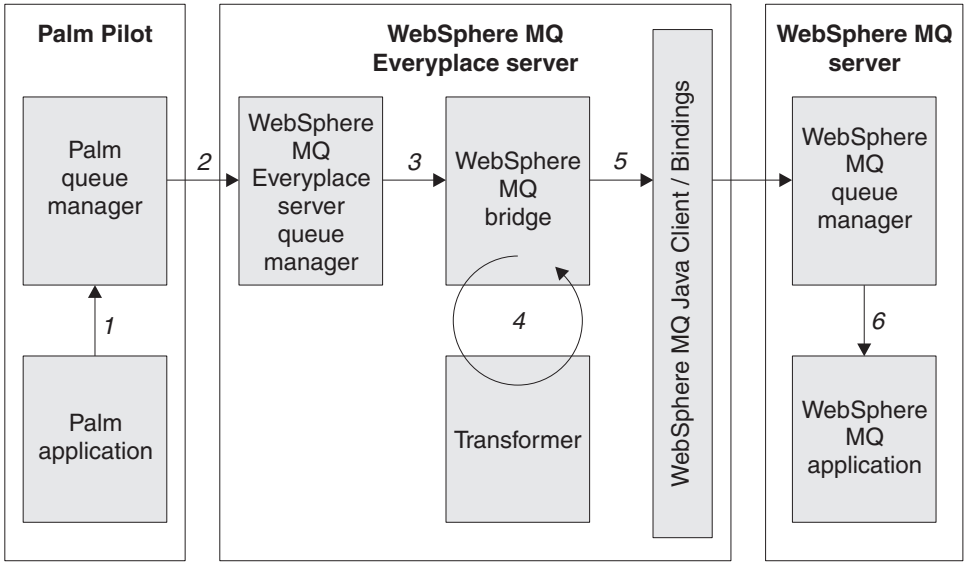


Figure 20. Message flow from MQe to MQ

1. **Client application**
   a. The client application builds an MQe 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 code pages.

   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. **MQe server puts the message onto the MQ bridge queue**
   The message is not translated at all through this step.
4. **MQ bridge passes the MQe message to the user-written transformer**

   Note: The examples in this section are in Java because transformers can only be written in Java.

   The transformer creates an MQ message as follows:
   - The Unicode field in the MQe message is retrieved using:
     `String value = MQemsg.GetUnicode(fieldname)`
   - The retrieved value is copied to the MQ message using `MQmsg.writeChars( value )`
   - The byte field in the MQe message is retrieved using:
     `Byte value = MQemsg.getByte(fieldName)`
   - The retrieved value is copied to the MQ message using `MQmsg.writeByte(value)`
   - The ascii field in the MQe 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 MQ message.

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

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

   Unicode values in the MQ message are translated from big-endian to little-endian, and vice versa, as required. Byte values in the 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 MQ, using conversion routines inside the 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 MQ Classes for Java client connection, and the CCSID of the receiving MQ server connection.

6. **The message is got by an 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 MQe application, and wish to convey character-related data from MQe to 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 code pages**, the character for a codepoint changes as you change between the various ASCII code pages, then use either putUnicode, which is never subject to translation between code pages, or putArrayOfByte, in which case you have to handle the translation between the sender's code page and the receiver's code page.

  Note: ***DO NOT USE*** putAscii() as the characters in the variant parts of the ASCII code pages are subject to translation.

- **If your data contains only characters in the invariant ranges of the ASCII character code pages**, then you can use putUnicode (which is never subject to translation between code pages) or putAscii, which is never subject to translation between code pages, as all your data lies within the invariant range of the ASCII code pages.

Configuring queue managers as servlets

Introduction

An MQe queue manager can run within a servlet.

Note: In MQe version 2.0, the *deprecated* jar must be in the classpath for servlets to work.

This section describes an example servlet that is included with MQe, and how to configure it using WebSphere Application Server 4.0 (WAS).

An example servlet configuration using WAS

An MQe queue manager can run within a servlet.

Note: In MQe version 2.0, the *deprecated* jar must be in the classpath for servlets to work.

This section describes an example servlet that is included with MQe, and how to configure it using WebSphere Application Server 4.0 (WAS).

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.

Start the Application Assembly tool

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.

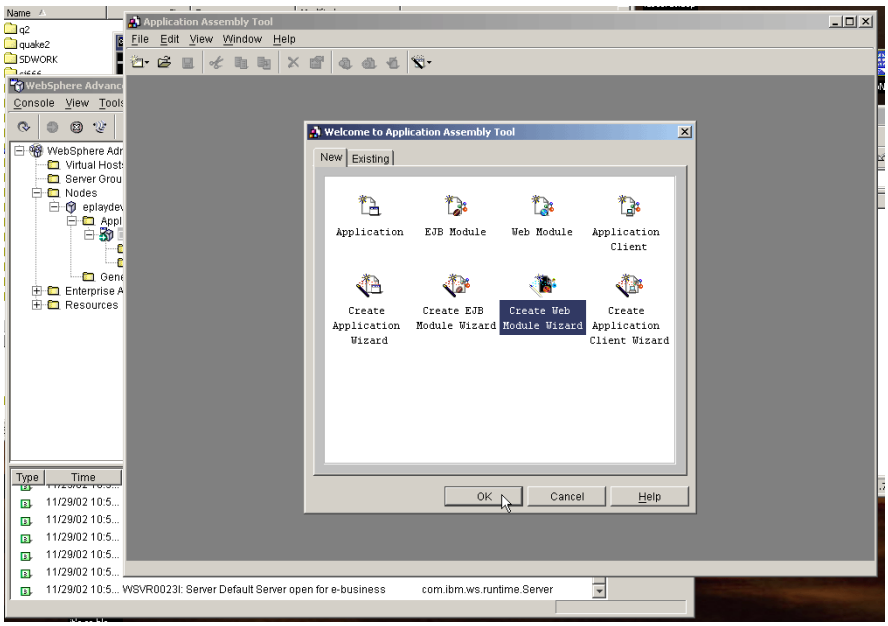


Figure 21. The WebSphere administrative console

Select "Create Web Module Wizard", and click OK. In specifying the properties, enter the file name, and more information, if you wish.

Specifying web module properties

In specifying the properties, enter the file name, and more information, if you wish.

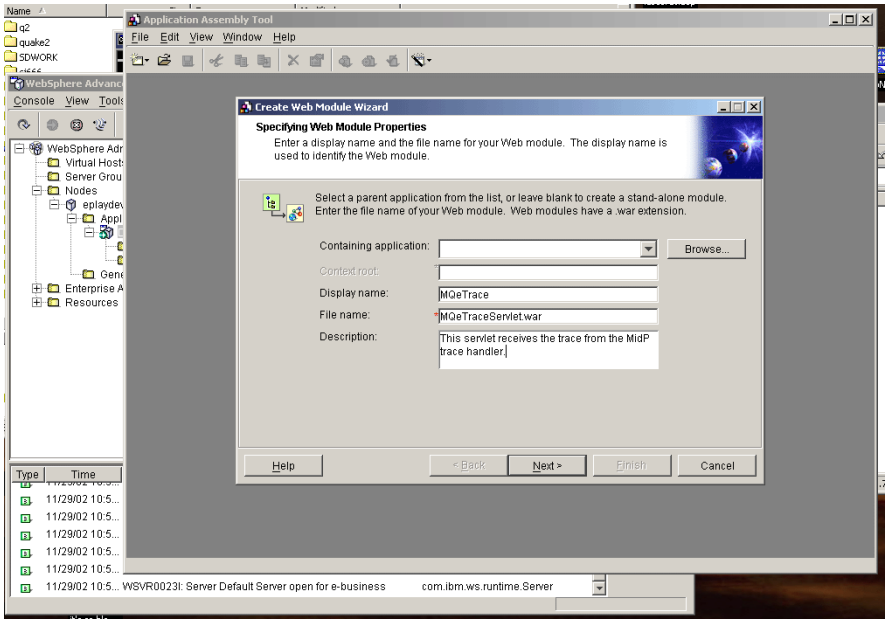


Figure 22. Specifying Web module properties

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.

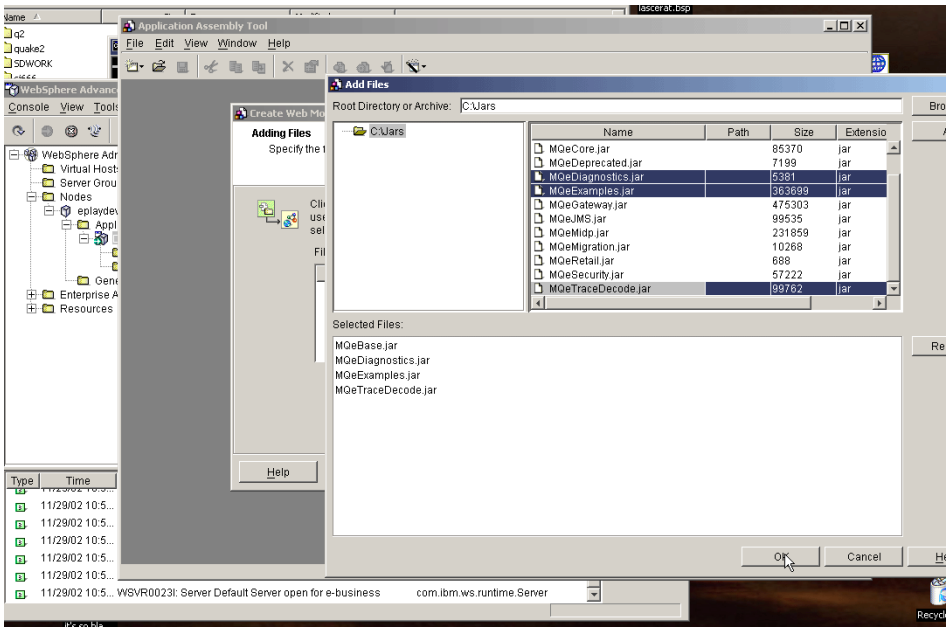


Figure 23. Adding files to the application

Adding web components

Next you have to specify the component properties.

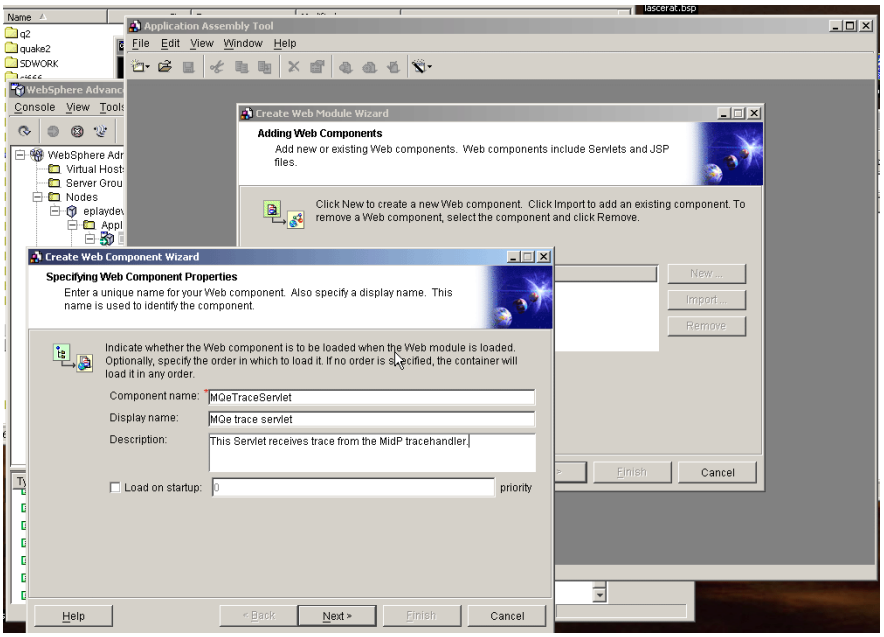


Figure 24. Adding web comopnents

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.

Specifying component type and class name

The next panel allows you to specify which class is the servlet to run.

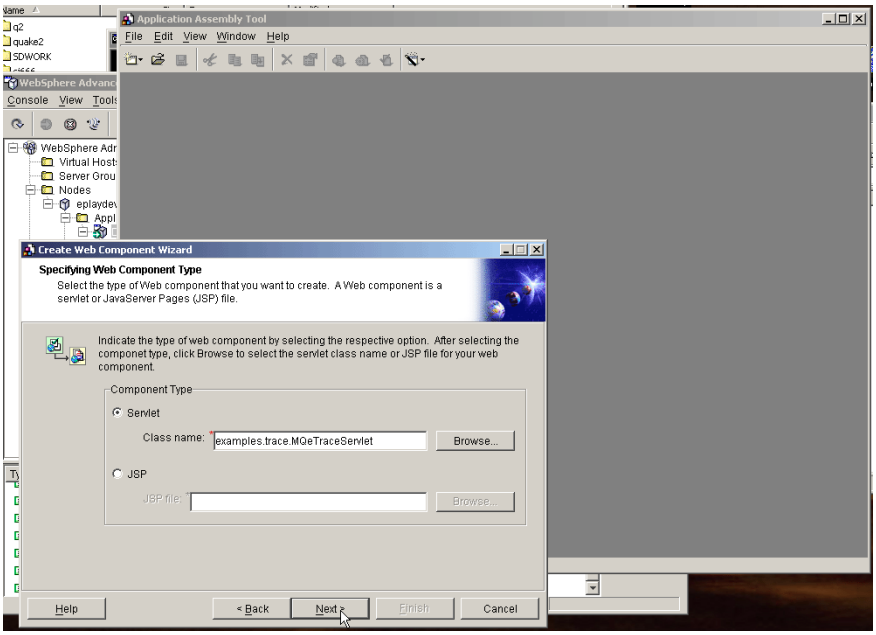


Figure 25. Specifying component type and class name

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

Specifying a URL to map to your servlet

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

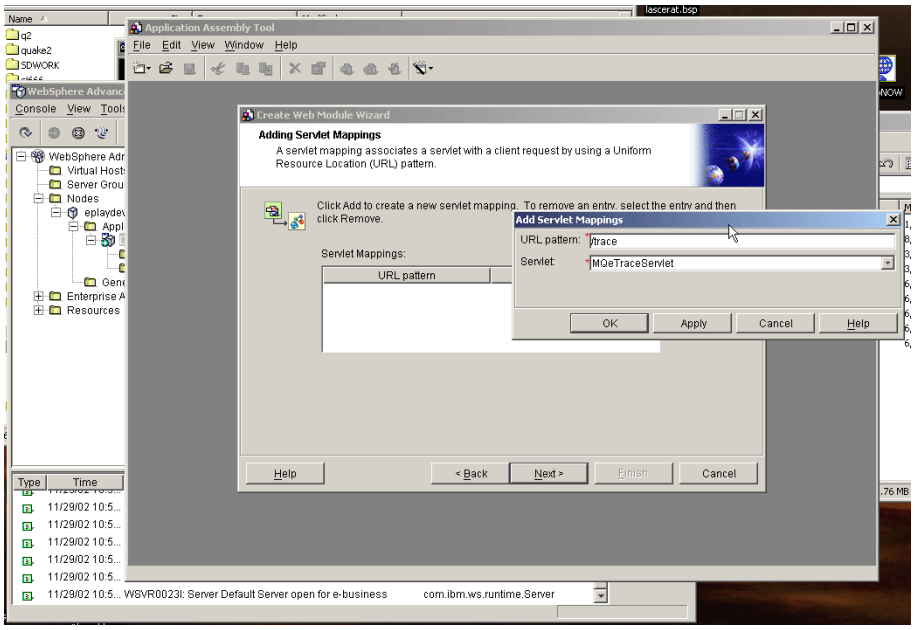


Figure 26. Specifying a URL to map to your servlet

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.

Finishing and saving the file

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.

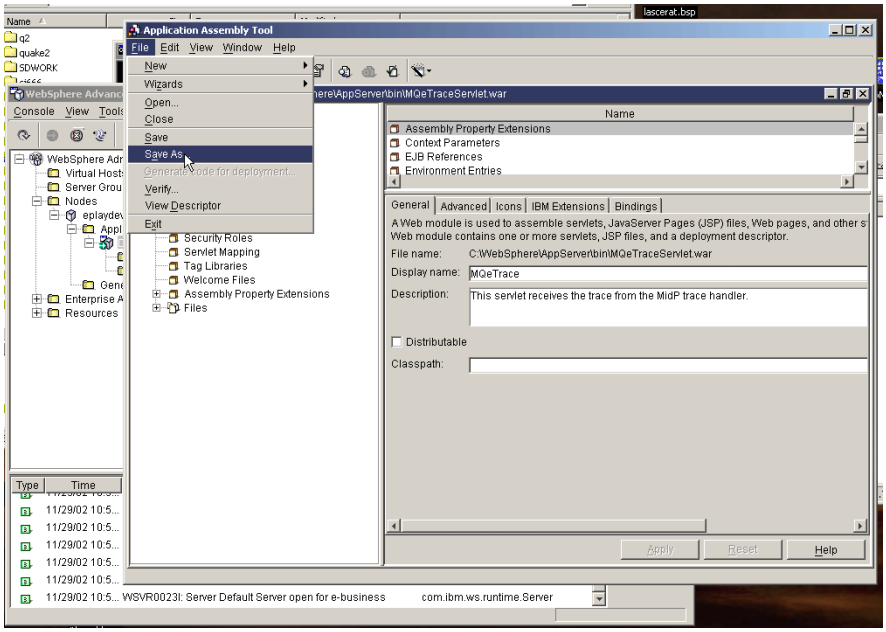


Figure 27. Saving the file

Next, this component needs to be imported and started. From the wizards button, select "Install Enterprise Application".

Install enterprise application

Next, this component needs to be imported and started. From the wizards button, select "Install Enterprise Application".

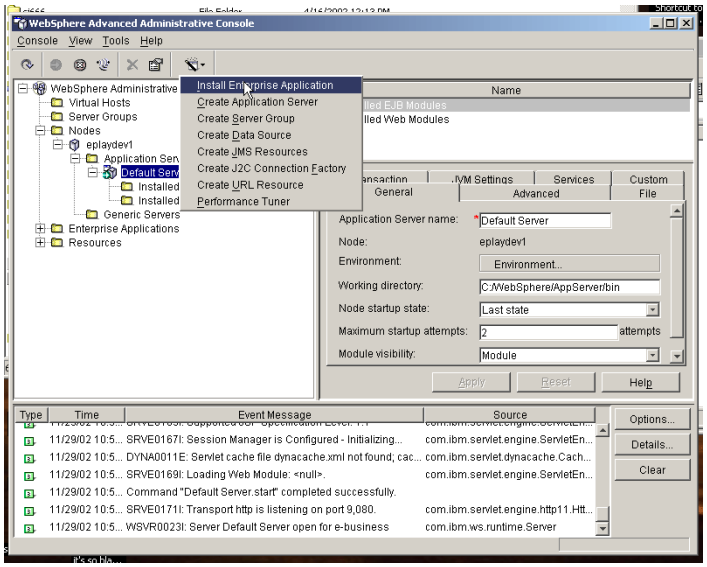


Figure 28. Install enterprise application

Install your component as a standalone module.

Installing your component as a standalone module

Install your component as a standalone module.

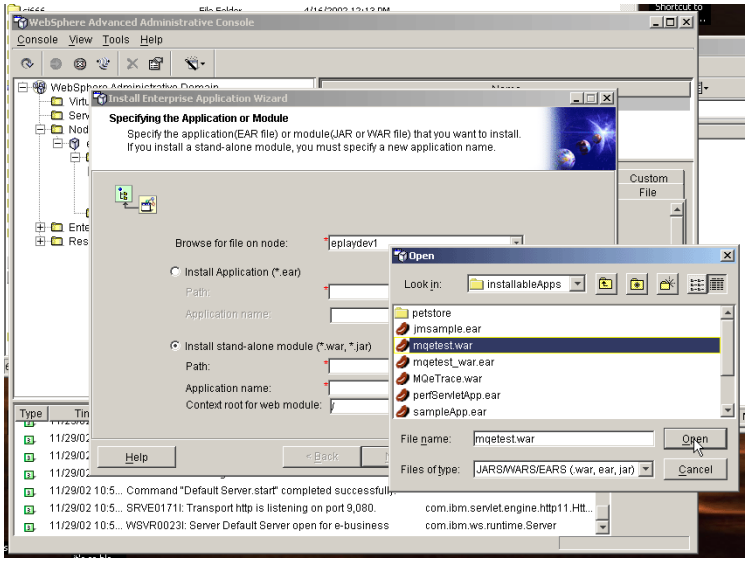


Figure 29. Installing your component as a standalone module

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 /

Specifying an application name

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 /

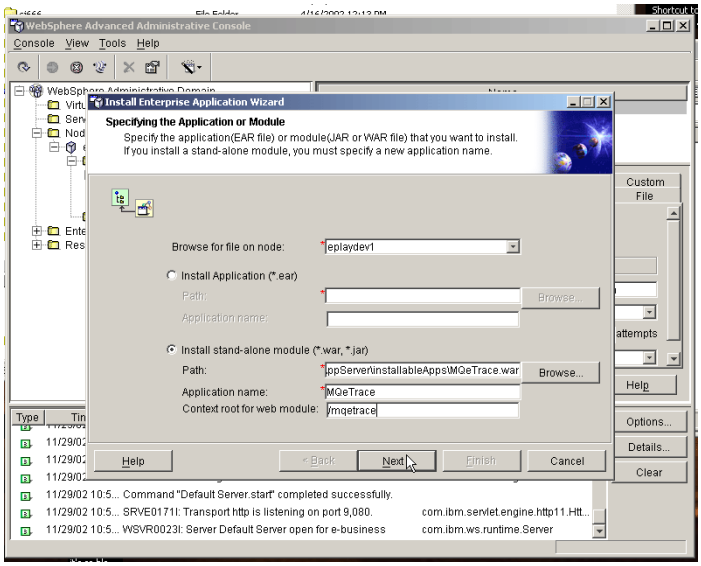


Figure 30. Specifying an application name

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.

Finishing the configuration

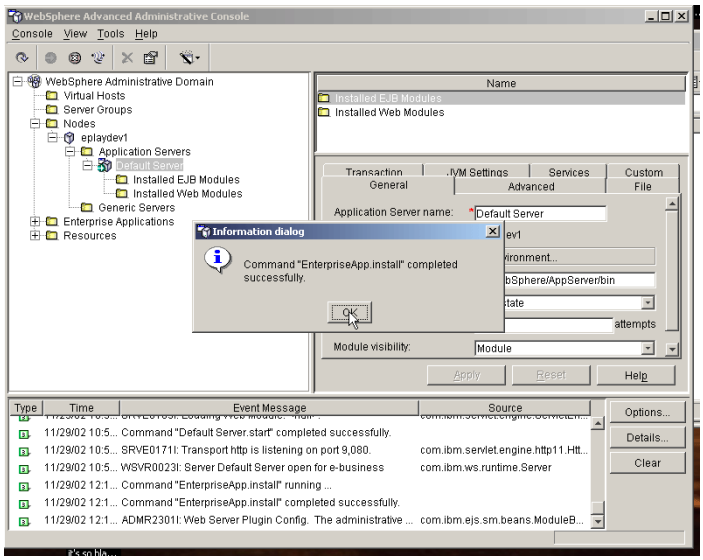


Figure 31. Information dialog

Starting the web module

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.

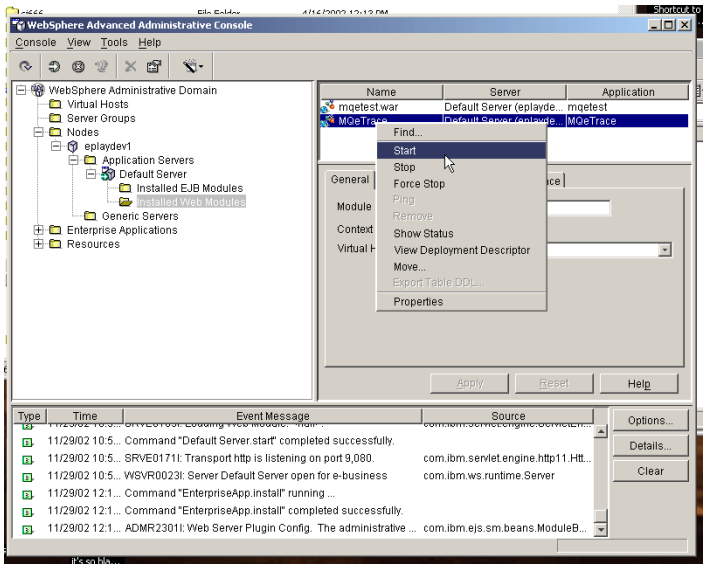


Figure 32. Starting the web module

Start succeeded

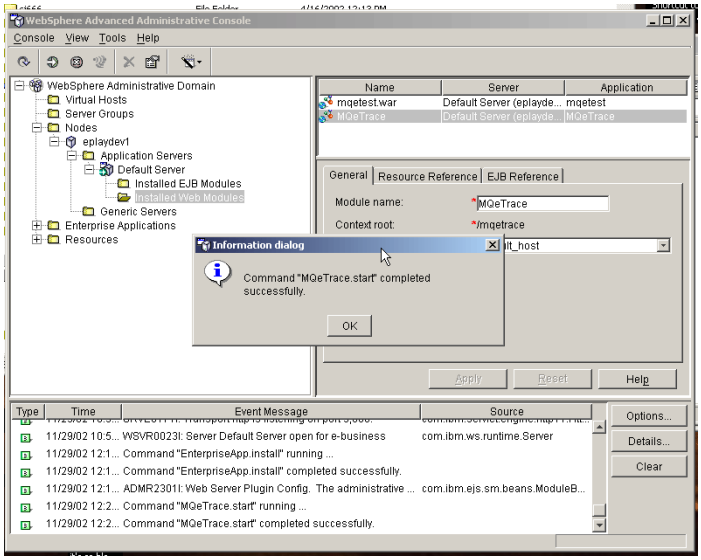


Figure 33. Information dialog success message

Using the servlet

If everything went well, it should now be available for use from the com.ibm.mqe.trace.MQeTraceToBinaryMidp.

Because 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.

JMS (Java Message Service) configuration

JMS Object naming changes from V2.0.1

The following naming changes apply starting from MQe V2.0.1 (the old names will still work for backward compatibility).

Old name	New name
QueueConnection	Connection
MQeQueueConnection	MQeConnection
MQeQueueConnectionFactory	MQeConnectionFactory
QueueConnectionFactory	ConnectionFactory

Introduction to JMS

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 MQe JMS these correspond to two classes:

- `MQeConnectionFactory` must be configured so that it can obtain a reference to an MQe queue manager.
- `MQeJMSQueue` can be configured with details of an MQe queue.

Note:

These classes are typically placed in a JNDI namespace by an administrator.

However, because on small devices access to a JNDI namespace may be impractical or may represent an unnecessary overhead, these classes do not include the necessary methods to allow them to be bound by JNDI.

Instead, two subclasses, `MQeJNDIConnectionFactory` and `MQeJMSJNDIQueue` extend these classes to allow them to be stored using JNDI.

Configuring MQeConnectionFactory

MQeConnectionFactory is the MQe implementation of the javax.jms.ConnectionFactory interface. It is used to generate instances of Connection classes, which for MQe must have a reference to an active queue manager. The ConnectionFactory must be able to create a reference to an active queue manager in order to pass it on to the Connection classes that it generates. The MQeConnectionFactory 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 MQeConnectionFactory class can be extended to provide the required behavior, see “Extending MQeConnectionFactory” on page 125.

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:

```
(MQeConnectionFactory(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 Connection it starts the client queue manager using the initialization file and passes a reference to the active queue manager to the Connection. If it generates more Connection classes, it passes them a reference to the same active queue manager. When the last Connection 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 MQeConnectionFactory class is created, and it can also be set explicitly using the setIniFileName() method:

```
(MQeConnectionFactory(factory)).setIniFileName(null);
```

In this case, when the connection factory generates a Connection, it looks for a queue manager already running in the JVM and passes the Connection a reference to it. An exception is thrown if no queue

manager is running. If it generates more Connection classes, it passes them a reference to the same queue manager. When an external queue manager is used, the connection factory does not close the queue manager when the last Connection is closed.

Note: A JVM can run only one MQe 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 MQe implementation of the Queue class. It is used to represent MQe 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 MQe queue manager which owns the queue
- *mqeQueueName* is the name of the MQe 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 MQe 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: MQe 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 MQe administration tool for JMS

The administration tool provides a simple way for administrators to define and edit the properties of MQe JMS administered objects.

This tool is based on the administration tool shipped with JMS for MQ, differing only in the properties that can be applied to JMS administered objects.

The JMS administration tool is included in `MQeJMSAdmin.jar`.

Configuring the JMS administration tool

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.)

An example configuration file is included in `examples/jms/MQeJMSAdmin.config`.

Starting the JMS administration 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.

JMS 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 25. 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 the following table

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 the following table:

Table 27. JMS administered objects

Object type	Keyword	Description
MQeJNDIQueueConnectionFactory	QCF	The MQe implementation of the JMS ConnectionFactory interface. This represents a factory object for creating connections in the JMS 1.02b Point-to-Point messaging domain.
MQeJNDIConnectionFactory	CF	The MQe implementation of the JMS ConnectionFactory interface. This represents a factory object for creating connections in the JMS 1.1 unified messaging domain.
MQeJMSJNDIQueue	Q	The MQe implementation of the JMS Queue interface. This represents a message Destination.

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. The following table summarizes their use. Substitute TYPE with the keyword that represents the required administered object, as listed in the table above in "Administering JMS objects" above.

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.
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 JMS 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 of JMS objects

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 in "LDAP schema definition for Java object storage" on page 126.

JMS object properties

A property consists of a name-value pair in the format:

```
PROPERTY_NAME(property_value)
```

Names and values are not case sensitive, but are restricted to a set of recognized names shown in the following table:.

Table 29. Property names and valid values

Property	Short form	Valid values
AUTHENTICATOR	AUTH	Any String
CLIENTID	CID	Any String
DESCRIPTION	DESC	Any String
DUPSOKCOUNT	DOC	Any positive integer
INIFILE	INI	Any String
ISMQNATIVE	ISMQ	″True″ or ″False″
JMXENABLED	JMSX	″True″ or ″False″
QUEUE	QU	Any String
QMANAGER	QMGR	Any String
SHUTDOWN	SHUT	Any positive integer

Most of these properties apply only to specific object types, but note that ConnectionFactory properties apply also to QueueConnectionFactory properties. The properties and the types they apply to are listed in the following table, together with a short description.

Two columns indicate the properties that apply to **QCF/CF** (QueueConnectionFactory or ConnectionFactory) and **Q** (Queue).

Table 30. Property names and descriptions

Property	QCF/CF	Q	Description
AUTHENTICATOR	Y		Fully-qualified class name implementing com.ibm.mqe.jms.MQeJMSAuthenticator interface.
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 an MQe Queue Manager
ISMQNATIVE		Y	The destination is a non-JMS, MQ, queue.
JMSXENABLED	Y		Enable JMSX properties.
QUEUE		Y	The name of an MQe queue
QMANAGER		Y	The name of an MQe queue manager

Table 30. Property names and descriptions (continued)

Property	QCF/CF	Q	Description
SHUTDOWN	Y		Delay before connection shutdown, in milliseconds.

Extending MQeConnectionFactory

By default MQeConnectionFactory 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 MQeConnectionFactory 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 an MQe queue manager when a Connection is first created, while the second shuts it down cleanly when the final Connection 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 MQeConnectionFactory to start its own queue manager without the need for an initialization file:

```
import javax.jms.*;
import examples.config.*;
import com.ibm.mqe.jms.MQeConnectionFactory;
import com.ibm.mqe.MQeQueueManager;
import java.io.File;

// type on one line:
public class MQeExtendedConnectionFactory
         extends MQeConnectionFactory {

  // Queue Manager Name -
  private static final String queueManagerName = "ExampleQM";
  // Location of the registry -
  private static final String registryLocation = ".\\ExampleQM";
  // Queue store -
  private static final String queueStore = "MsgLog:"
                                         + registryLocation
                                         + File.separator
                                         + "Queues";
  // the MQe Queue Manager -
  private static MQeQueueManager queueManager = null;

  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 Java object storage

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 MQe 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

Table 33. Attribute settings for javaClassNames (continued)

Attribute	Value
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

Table 36. Attribute settings for javaSerializedData (continued)

Attribute	Value
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

JMX (Java Management Extensions) interface

This section describes the **MQe Java Management Extensions interface**. The name is shortened in the text to **MQe JMX**.

The MQe JMX interface provides a JMX instrumentation level for MQe resources (queue managers, queues, and so on). This JMX instrumentation level facilitates the local and remote configuration and

administration of MQe queue managers and their associated objects, such as queues, connections, listeners and bridge objects. In order to validate the operation of the network, test messages can be sent to queues within the MQe network.

This feature facilitates the management of MQe resources through JMX from all platforms that are supported for JMX. For further information on platform support for JMX, see the JMX specification V1.2 at

`http://jcp.org/aboutJava/communityprocess/final/jsr003/index3.html`

.

(Note that queue managers and their resources on platforms which are not supported by JMX can still be remotely administered via JMX.)

For those who are familiar with either the MQe_Explorer tool (formerly packaged in IBM® SupportPac ES02) or the MQe_Script tool (formerly packaged in IBM SupportPac ES04), the MQe JMX interface facilitates the equivalent configurative and administrative functionality. Objects created using MQe_Script, MQe_Explorer (version 2 or later) and the MQe JMX interface can interact together. These SupportPacs are now bundled together as ES06 MQe Server Support SupportPac.

The MQe JMX interface includes full support for the configuration and management of gateway queue managers, that is, those MQe queue managers that can bridge to MQ queue managers and queues. The interface does not support the configuration of the MQ queue managers themselves, as these should be configured using the various MQ management tools and protocols.

Introduction to MQe JMX

The Java Management Extensions (also called the JMX specification) define an architecture, the design patterns, the APIs, and the services for application and network management in the Java programming language. The JMX specification provides Java developers across all industries with the means to instrument Java code, create smart Java agents, implement distributed management middleware and managers, and smoothly integrate these solutions into existing management systems.

The JMX architecture provides the following benefits:

- Enables Java applications to be managed without heavy investment

  A Java application simply needs to embed a managed object server and make some of its functionality available as one or several Manageable Beans registered in the object server; that is all it takes to benefit from the management infrastructure.
- Provides a scalable management architecture

  Every JMX agent service is an independent module that can be plugged into the management agent, depending on the requirements. This component-based approach means that JMX solutions can scale from small footprint devices to large telecommunications switches and beyond.
- Integrates existing management solutions

  JMX smart agents are capable of being managed through HTML browsers or by various management protocols such as SNMP and WBEM. The JMX APIs are open interfaces that any management system vendor can leverage.
- Can leverage future management concepts

  The APIs of the JMX specification can implement flexible and dynamic management solutions through the Java programming language which can leverage emerging technologies.

  The goal of the JMX API for MQe is to provide a JMX instrumentation level for MQe resources (queue managers, queues, and so on). The instrumentation is designed to have a small footprint, and to be flexible, easy to use and JMX compliant.
- Small footprint

The JMX API for MQe and the JMX implementation minimize resource demands in terms of size and memory requirements.

- Flexible

  The implementation is modular so that MQe developers can choose to use the API when manageability is desired and the overhead of the JMX implementation is within the capabilities of their target platforms. Or they can choose to leave it out without incurring any memory or performance penalty.

- Easy to use

  The API is simple and easy to use: it is possible to enable an existing MQe application for JMX with only a few lines of code.

- JMX compliant

  JMX compliance is virtually guaranteed by using the reference implementation developed by Sun Microsystems. From JMX v.1.2 onwards, open MBeans are a mandatory part of any JMX implementation, so we adhere to the data types required for open MBean instrumentation. Thus, all operation parameters and attributes are of the following data types:

  – Simple data types:
    - java.lang.Void
    - java.lang.Boolean
    - java.lang.Byte
    - java.lang.Character
    - java.lang.String
    - java.lang.Short
    - java.lang.Integer
    - java.lang.Long
    - java.lang.Float
    - java.lang.Double

  – Arrays of the above types:
    - javax.management.ObjectName
    - javax.management.openmbean.CompositeData
    - javax.management.openmbean.TabularData

JMX architecture

The JMX architecture is multilayered, as shown in the following diagram:

<table>
<tr><td>Distributed Services level</td><td>Web browser</td><td>Other</td><td>JMX-compliant management applications</td><td>Proprietary management applications</td></tr>
<tr><td rowspan="4">Agent level</td><td colspan="2" rowspan="2">Protocol adapters</td><td rowspan="2">Connectors..</td><td>JMX manager</td></tr>
<tr><td>..</td></tr>
<tr><td colspan="2" rowspan="2">MBeanServer...</td><td>...</td></tr>
<tr><td>Agent Services</td></tr>
<tr><td rowspan="2">Instrumentation level</td><td colspan="4">Instrumentation strategy (MQe JMX implementation)</td></tr>
<tr><td colspan="4">Application Resources</td></tr>
</table>

The scope of the MQe JMX implementation is limited to the *Instrumentation level* and **Instrumentation strategy (MQe JMX implementation)**.

The following are explanations of some of the terms used in the diagram:

Distributed services level
The distributed services level of the JMX architecture contains the middleware that connects agents to management applications.

Agent level
The agent level of the JMX architecture provides a registry for handling the manageable resources, called the MBeanServer, as well as several agent services which are themselves MBeans.

JMX agent
A JMX agent is a combination of an instance of the MBeanServer, its registered MBeans and any agent services within a single JVM.

Managed Beans (MBeans)
Resources instrumented according to the rules of the JMX specification. There are two main categories of MBeans:

- *Standard MBeans* implement their own interface, and are static.
- *Dynamic MBeans* (of which there are several sub-categories) implement a JMX interface called DynamicMBean. This interface contains methods that allow the management interface of the managed resource to be discovered at run-time.

Instrumentation level
The instrumentation level of the JMX architecture is the level at which resources to be managed are instrumented for JMX management. To make this possible, the resources must be instrumented as MBeans.

Resource
Any entity that needs to be monitored or controlled by a management application. In the context of this implementation, MQe queue managers, queues, and so on, are all resources.

Instrumenting your MQe resources as JMX MBeans

In your application, load or create your queue manager and activate it as usual. Now, create JMX MBeans for your queue manager and its resources as follows:

```
MQeQueueManagerJmx.createMQeMBeans(mbServer);
```

where mbServer is your instance of MBeanServer.

This method creates and registers MBeans for all of your queue manager resources: queues, bridge objects, connections and listeners.

Note: Whenever MQe resources are created or removed from this point, corresponding MBeans are also registered with or deregistered from the MBeanServer instance. It is strongly recommended that you do not create and register or deregister MQe MBeans independently of using the MQe interface, otherwise the MQe MBean representations may not be consistent and may not function as intended. For example, you should not use the register/unregister facilities offered by various adapters. Using the MQe JMX interface to create and delete methods ensures that MBeans are registered and unregistered in the approved manner. However, the AdminBean is an exception to this rule – see “ObjectName” on page 133.

You need to create one or more connectors or adaptors to allow JMX management clients to connect to and manage your MQe applications. Both the Sun and Tivoli® JMX Reference Implementations provide adaptors which allow you to manage your MQe application through a web browser. Please refer to the reference implementations for documentation and examples.

In addition to the HtmlAdaptorServer, the Sun JDMK provides the HttpConnectorServer, HttpsConnectorServer, the RmiConnectorServer, and the SnmpAdaptorServer. These allow JMX management clients to connect to and manage JMX manageable resources using the HTTP, HTTPS, RMI, and SNMP protocols. Refer to the JDMK for documentation and examples.

Once you have your connector(s) or adaptor(s), or both, you are in a position to access the MQe MBeans as specified in the JMX specification. You need to have all of the following queues set up:

- An admin queue on your local queue manager for local administration. The default assumes that this queue is named AdminQ but you can re-set this using the Admin MBean.
- An admin reply queue called AdminReplyQ.
- Queues named AdminQ and AdminReplyQ on any remote queue managers that you wish to manage via the JMX interface. If either of these queues does not exist (or the relevant connection definitions and listeners for remote two-way admin-adminReply communication do not exist), you may experience problems when performing remote administration.

When you have closed your MQe queue manager at the end of your application, you must invoke the following static method to ensure that all MQe JMX resources are cleaned up:

```
MQeQueueManagerJmx.endMQeJMXSession()
```

It is important that this method is called after the queue manager has been closed.

Typographical conventions in this JMX documentation

Text enclosed in angle brackets and italicized, for example <*QMName*>, represents a symbolic name, the value of which should be substituted with a value provided by the user as the command is typed.

Text written using a monospaced font, for example `getMBeanInfo`, represents user input, code in files, or text entered into a browser textbox.

Setting up the MQe JMX interface

The MQe JMX interface executes as an application running in a Java Virtual Machine (JVM). All it requires is an activated local queue manager. Given this, the interface can then manage the instrumented local queue manager, and the queue manager's resources. It can also manage any remotely activated MQe queue managers (and their resources) for which the local queue manager is able to connect directly to the MQe network.

Note: Before you can use JMX you must make sure certain properties files exist on your classpath. See "Translation" on page 147 for details.

MQe JMX

- A JVM version 1.2 or later
- A compliant implementation of the JMX specification.

The jar files provided by the JMX specification implementation must be added to the CLASSPATH before you attempt to use the MQe JMX interface APIs. The JMX Reference Implementation provided by Sun is freely available and redistributable (`http://java.sun.com/products/JavaManagement/`). To install it:

1. Download the Reference Implementation binary code, which comes in a ZIP file
2. Extract the contents to a directory
3. Copy lib/jmxri.jar and lib/jmxtools.jar into the extension directory of your Java runtime environment, or make sure they are in your classpath.

The JMX API for MQe has been developed in compliance with the JMX specification v.1.2.

The Tivoli Implementation of the JMX specification is also freely available (`http://www.alphaworks.ibm.com/tech/TMX4J`). To install it:

1. Download the binary code, which comes in a ZIP file.
2. Extract the contents to a directory
3. Copy the relevant jars for your platform into the extension directory of your Java runtime environment, or make sure they are in your classpath.

Enabling MQe applications for JMX managment

To enable your MQe applications for JMX management:

1. Obtain a compliant implementation of the JMX specification and set up your Java development environment so that the JMX and MQe JMX class libraries are accessible (as described in "Setting up the MQe JMX interface" on page 132).
2. Create your MQe application as usual, ensuring that your queue manager is loaded, then call the following static method, passing it your MBeanServer instance, to create MBeans for all of the queue manager's resources:

   ```
   MQeQueueManagerJmx.createMQeMBeans(MBeanServer mbServer);
   ```

   Each of the MBeans is registered with your MBeanServer instance. For further details see "Instrumenting your MQe resources as JMX MBeans" on page 131.
3. Create one or more connectors or adaptors to allow JMX management clients to connect to and manage your MQe applications. For further details on creating connectors and adaptors, see "Instrumenting your MQe resources as JMX MBeans" on page 131. You are now in a position to manage all of your MQe resources via JMX using the interface corresponding to your chosen connector or adaptor. It is also very important that you invoke the following static method at the end of your application to ensure that all MQe JMX resources are cleaned up:

   ```
   MQeQueueManagerJmx.endMQeJMXSession()
   ```

Accessing MQe MBeans via the MBeanServer

The MQe JMX interface provides the instrumentation level of the JMX architecture. We are not providing an implementation of the agent level, which is made up of the MBeanServer and the JMX agent services. Since the role of the MBean server is to act as a registry for MBeans, the user's instance of MBean server has to be passed to the instrumentation code via the MQeQueueManagerJmx.createMQeMBeans() method described in this section. All instrumented MQe MBeans are then registered with this MBeanServer instance.

An instance of an MBeanServer is created using one of two static methods of the MBeanServerFactory class: `createMBeanServer()` or `newMBeanServer()`. Once this instance has been created, MBeanServer methods can be used to access and manipulate the MBeans registered with the MBeanServer. In particular, the following methods allow the user to retrieve and set MQe MBean attributes and invoke operations. (The method names below assume that `mbeanServer` is the instance of MBeanServer.)

```
mbeanServer.getAttribute(ObjectName objName, String attributeName);
mbeanServer.getAttributes(ObjectName objName, String[] attributeNames);
mbeanServer.setAttribute(ObjectName objName, Attribute attribute);
mbeanServer.setAttributes(ObjectName objName, AttributeList attributes);
mbeanServer.invoke( ObjectName name, String operationName,
                    Object params[], String signature[]  );
```

For further details on the parameter types *Attribute* and *AttributeList*, see "Related information on JMX" on page 148. The concept of an MBean's ObjectName is central to the MQe JMX interface and is discussed in the following section.

ObjectName

An MBeanServer instance interacts with the MBeans registered with it via their ObjectNames. When an MBean is registered with an MBeanServer, both the MBean object instance and the corresponding MBean ObjectName instance are passed as parameters to the registration method. From this point onwards, the ObjectName is passed to all MBeanServer methods pertaining to this MBean.

ObjectNames are also returned from query methods on the MBeanServer instance which are designed to inquire upon the MBeans registered with the MBeanServer. A form of pattern matching can be used in these methods. Therefore the ObjectName hierarchy corresponding to MQe instrumented resources has been designed to facilitate queries on types of MQe resource such as Application Queue and Indirect Connection.

An object name consists of a string made up of two components: the domain name and the key property list. It has the format:

```
Domain-name:key1=value1[,key2=value2,...keyX=valueX]
```

A domain name corresponds to a namespacing prefix which identifies a group of MQe resources.

The following table provides a full description of the MQe JMX object naming conventions.

Table 41. MQe JMX Object Naming Conventions

MQe Resource	ObjectName
Local Queue Manager	com.ibm.MQe_LocalQueueManager:name = *<QMName>*
Remote Queue Manager	com.ibm.MQe_RemoteQueueManagers:name = *<QMName>*
Local Queue Manager Alias	com.ibm.MQe_LocalQueueManager:name = *<QMAliasName>*, type = alias, resourceName = *<QMName>*
Remote Queue Manager Alias	com.ibm.MQe_RemoteQueueManagers:name = *<QMAliasName>*, type = alias, resourceName = *<QMName>*
Application Queue	com.ibm.MQe_*<OwningQMName>*_ApplicationQueues:name = *<QName>*
Application Queue Alias	com.ibm.MQe_*<OwningQMName>*_ApplicationQueues:name = *<QAlias>*, type = alias, resourceName = *<QName@OwningQMName>*
Sync Proxy Queue	com.ibm.MQe_*<OwningQMName>*_SyncProxyQueues:name = *<QName>*, DestinationQMgr = *<DestinationQMgrName>*
Sync Proxy Queue Alias	com.ibm.MQe_*<OwningQMName>*_ SyncProxyQueues:name = *<QAlias>*, type = alias, resourceName = *<QName@DestinationQMgrName>*
Async Proxy Queue	com.ibm.MQe_*<OwningQMName>*_ AsyncProxyQueues:name = *<QName>*, DestinationQMgr = *<DestinationQMgrName>*
Async Proxy Queue Alias	com.ibm.MQe_*<OwningQMName>*_ AsyncProxyQueues:name = *<QAlias>*, type = alias, resourceName = *<QName@DestinationQMgrName>*
Admin Queue	com.ibm.MQe_*<OwningQMName>*_AdminQueues:name = *<QName>*
Admin Queue Alias	com.ibm.MQe_*<OwningQMName>*_ AdminQueues:name = *<QAlias>*, type = alias, resourceName = *<QName@OwningQMName>*
Home Server Queue	com.ibm.MQe_*<OwningQMName>*_HomeServerQueues:name = *<QName>*, GetFromQMgr = *<GetFromQMgr>*

Table 41. MQe JMX Object Naming Conventions (continued)

MQe Resource	ObjectName
Store Queue	com.ibm.MQe_<*OwningQMName*>_StoreQueues:name = <*QName*>
Forward Queue	com.ibm.MQe_<*OwningQMName*>_ForwardQueues:name = <*QName*>, ForwardToQMgr = <*ForwardToQMgrName*>
MQeMQBridge Queue	com.ibm.MQe_<*OwningQMName*>_BridgeQueues:name = <*QName*>, DestinationQMgr = <*DestinationQMgrName*>
MQeMQBridge Queue Alias	com.ibm.MQe_<*OwningQMName*>_BridgeQueues:name = <*QAlias*>, type = alias, resourceName = <*QName@DestinationQMgrName*>
Alias Connection	com.ibm.MQe_<*OwningQMName*>_MQConnections:name = <*ConnectionName*>
Alias Connection Alias	com.ibm.MQe_<*OwningQMName*>_MQConnections:name = <*AliasName*>, type = alias, resourceName = <*ConnectionName*>
Direct Connection	com.ibm.MQe_<*OwningQMName*>_DirectConnections:name = <*ConnectionName*>
Direct Connection Alias	com.ibm.MQe_<*OwningQMName*>_DirectConnections:name = <*AliasName*>, type = alias, resourceName = <*ConnectionName*>
Indirect Connection	com.ibm.MQe_<*OwningQMName*>_IndirectConnections:name = <*ConnectionName*>
Indirect Connection Alias	com.ibm.MQe_<*OwningQMName*>_IndirectConnections:name = <*AliasName*>, type = alias, resourceName = <*ConnectionName*>
MQ Connection	com.ibm.MQe_<*OwningQMName*>_MQConnections:name = <*ConnectionName*>
MQ Connection Alias	com.ibm.MQe_<*OwningQMName*>_MQConnections:name = <*AliasName*>, type = alias, resourceName = <*ConnectionName*>
Communications Listener	com.ibm.MQe_<*OwningQMName*>_CommunicationsListeners:name = <*ListenerName*>
MQ Bridge	com.ibm.MQe_<*OwningQMName*>_Bridges:name = <*BridgeName*>
MQ QMgrProxy	com.ibm.MQe_<*OwningQMName*>_MQQueueManagerProxies:name = <*ProxyName*>, bridge = <*BridgeName*>
MQ Client Connection	com.ibm.MQe_<*OwningQMName*>_MQClientConnections:name = <*ClientConnName*>, bridge = <*BridgeName*>, qmgrProxy = <*ProxyName*>

Table 41. MQe JMX Object Naming Conventions (continued)

MQe Resource	ObjectName
MQ Listener	com.ibm.MQe_*<OwningQMName>*_MQListeners:name = *<ListenerName>*, bridge = *<BridgeName>*, qmgrProxy = *<ProxyName>*, clientConnection = *<ClientConnName>*
MQe Admin bean	com.ibm.MQe_Admin:name = AdminBean

From the application, using this schema, queries may be done on the MBeans using the name or type fields or wildcards in the string preceding the colon (this string is known as the Domain). Thus it is easy to search for all application queues, all proxy queues, all connection aliases, and so on.

There are some important points to note about the use of ObjectNames for MQe resources:

- There are no spaces in these names since these make queries more difficult. Therefore, you must ensure that no spaces are accidentally inserted into object names used as parameters to methods, or exceptions will arise due to the resource not being found.
- Object names are case sensitive.
- When a queue has an alias, the resourceName property key in the alias MBean object name has a value which is composed of a string of the form queueName@queueManagerName.

The MQe JMX interface provides a helper method to instrument a queue manager's resources as MBeans (The MQeQueueManagerJmx method createMQeMBeans()). When this method is used, all of the MBeans have object names following the pattern specified above. However, it would also be possible for an application to instantiate instances of MQe MBeans by calling the appropriate constructor and it would then be possible to register the resultant MBean with the MBeanServer with an object name chosen by the application.

It is strongly recommended that you adhere to the naming conventions described above and do not register or unregister MQe MBeans independently of the MQe JMX interface. The MQe JMX interface helper method MQeQueueManagerJmx.createMQeMBeans() should always be used to instrument your MQe resources as MBeans.

The only instance where calling the constructor to create a MQe MBean is supported is to change defaults of the Admin MBean. This can be seen in the example code provided in the `examples\jmx` directory. For example, the register/unregister facilities offered by various adaptors should not be used - going via the MQe JMX interface create and delete methods ensures that MBeans are registered and unregistered in the approved manner (the AdminBean is an exception to this rule).

These conventions are used within the MQe code and the processing of MBeans may not be consistent if a different naming pattern is used. Using the helper method ensures a consistency of behavior, for example, when a refresh occurs on remote queue manager MBeans due to a resource having been added or removed by some method other than via JMX.

The object name patterns described above have been selected with a view to facilitating queries on the MBeanServer instance for its registered MBeans. Such queries allow for pattern-matching based on the object names. (See the JMX resource documentation listed in the preface for descriptions of how MBeanServer queries work). For example, to get a subset of registered MBeans corresponding to a local queue manager's application queues, the following query could be made:

```
// set up a filter for retrieving MyLocalQM's Application Queue MBeans
ObjectName scope =
   new ObjectName("com.ibm.MQe_MyLocalQM_ApplicationQueues:*");

// use the JMX MBeanServer API to make the query
```

```
Set results = mbeanServer.queryNames(scope,null);

// iterate through the results
Iterator iter = results.iterator();
ObjectName objName = null;
while(iter.hasNext()) {
objName = (ObjectName)iter.next();
  // process each result
    ...
}
```

The following example shows how to find out which resources actually represent queue aliases:

```
// set up a filter for retrieving all aliases for queues
ObjectName scope = new ObjectName("*Queues:*,type=alias");

// use the JMX MBeanServer API to make the query
Set results = mbeanServer.queryNames(scope,null);

// etc.
```

Useful MBeanServer methods

Once you have called the helper method `MQeQueueManagerJmx.createMQeMBeans()`, which instruments all of your queue manager resources as MBeans, you are in a position to manipulate those resources using the standard MBeanServer API.

In general terms, this manipulation involves either the setting or getting of MBean resource attributes or the invocation of MBean resource operations.

All attribute and operation manipulation at this level is done via the following agent-layer API.

getMBeanInfo:

```
public MBeanInfo getMBeanInfo(ObjectName objName)
  throws InstanceNotFoundException,
         IntrospectionException,
         ReflectionException;
```

In order to use some of the other MBeanServer methods described in this section, such as the `invoke()` method, you need information about the relevant input parameters. For example, you may need to know what operations can be invoked upon a given resource, what input parameters are required for a particular operation, what the type of each of these parameters is and what the return value type is.

There are two ways in which you can obtain this information. Firstly, you can use the list of attributes and operations for each MQe JMX-instrumented resource. You can look up the information that you need and hard-code it into an application as required.

Alternatively, you can use the getMBeanInfo() MBeanServer method that the agent layer provides to retrieve the input parameter information. This method takes as its sole parameter the ObjectName instance that corresponds to the equivalent MQe resource. The method returns a complex structure that contains information on the following properties of an MBean: class name, description, attributes, constructors, operations, notifications.

The information that the getMBeanInfo() method returns on attributes, constructors, operations and notifications consists of further structures of types MBeanAttributeInfo, MBeanConstructorInfo, MBeanOperationInfo and MBeanNotificationInfo. The method can also retrieve an MBeanParameterInfo instance that corresponds to each MBeanOperationInfo instance, and so on.

Below is one example of how to use the getMBeanInfo() method. Given the complexity of the MBeanInfo object, you will also find it helpful to refer to the JMX information sources listed in the Related Material section.

Suppose you know that an instrumented MQe application queue MBean has an addAlias method but you want to check the return type of this method. To do this, you would use the getMBeanInfo() method as follows:

```
/* call the getMBeanInfo() method on              */
/*  the MBeanServer instance for the queue MBean  */
MBeanInfo beanInfo = mbeanServer.getMBeanInfo(queueObjName);

/* retrieve information on operations for that MBean */
MBeanOperationInfo[] beanOps = beanInfo.getOperations();

/* loop through the operations until we find the one we want */
for(int i = 0; i < beanOps.length; i++) {
    if(beanOps[i].getName().equals("addAlias")) {
      /* get the return type for that operation */
      String retval = beanOps[i].getReturnType();
      System.out.println(retval);
      break;
    }
```

A very useful aspect of these information structures is the description parameter. Instances of MBeanAttributeInfo, MBeanParameterInfo, MBeanConstructorInfo and MBeanOperationInfo all have a getDescription() method, which you can use to return a text description of the item in question.

getAttribute:

```
public Object getAttribute(ObjectName objName, String attrname)
  throws MBeanException,
         AttributeNotFoundException,
         InstanceNotFoundException,
         ReflectionException;
```

This API allows the agent layer to retrieve the value of an MBean attribute. (See the JMX documentation on the Attribute class for further details.) From the point of view of the MQe JMX interface, the most important properties of this class are name and value. The `getAttribute()` method takes two parameters: an `ObjectName` corresponding to the resource in question (a JMX-instrumented queue, for example) and a `String` parameter corresponding to the Attribute name. The method returns an Object which must be cast to the expected type of the Attribute value.

So, for example, if a MQe queue MBean has an attribute named Description of type java.lang.String, the value for that attribute would be retrieved at the agent layer as follows (assuming that the ObjectName for the queue in question has been retrieved from a query):

```
String queueDesc = (String)mbeanServer.getAttribute( queueObjName,
                                                     "Description");
```

This method throws exceptions of type: `AttributeNotFoundException`, `MBeanException`, or `ReflectionException`. MQe Exceptions are returned wrapped in MBeanExceptions. See “Error handling” on page 143.

Note: For the sake of convenience, the try/catch blocks needed to catch exceptions thrown by these MBeanServer methods are omitted in the examples in these sections. See “Error handling” on page 143.

Some adapters such as the Sun HtmlAdaptorServer invoke the getAttribute() and setAttribute() methods recursively when getting or setting several attributes rather than invoking getAttributes() or setAttributes(). This may result in a high overhead. In this case, it is advisable to increase the `cacheInterval` attribute in the Admin MBean. Caching attribute values decreases the amount of work being done by the adaptor.

getAttributes:

```
public AttributeList getAttributes( ObjectName name,
                                    String[] attributes)
  throws InstanceNotFoundException,
         ReflectionException;
```

The attributes parameter consists of an array of the names of attributes to be retrieved. The return value is of type javax.management.AttributeList extends java.util.ArrayList and provides methods for adding Attribute objects to an AttributeList. Attributes are retrieved from an AttributeList using an instance of Iterator and the Attribute class methods getName() and getValue().

```
String[] attributeNames = {"Description","Expiry"};
AttributeList myAttrs =
  mbeanServer.getAttributes(queueObjName,attributeNames);
Iterator myIter = myAttrs.iterator();
while(myIter.hasNext()) {
  Attribute attribute = (Attribute)myIter.next();
  System.out.println("Attribute name: " + attribute.getName());
  System.out.println("Attribute value: " + attribute.getValue());
}
```

The corresponding method for getAttributes() at the instrumentation level cannot throw user exceptions. This limits the usefulness of getAttributes() at the agent layer as, for example, MQe exceptions cannot be retrieved. Instead of using getAttributes(), it may be more useful to loop through the String array of attribute names (attributeNames), calling getAttribute() for each, though this increases the overhead. The same applies to setAttributes().

setAttribute:

```
public void setAttribute( ObjectName name,
                          Attribute attribute)
  throws InstanceNotFoundException,
         AttributeNotFoundException,
         InvalidAttributeValueException,
         MBeanException,
         ReflectionException;
```

This method is used to set the name and value of a new Attribute or to update the current value of an Attribute. The following example shows how to use the JMX-instrument MQe queue MBean known by object name queueObjName to set the Description attribute at the agent level:

```
Attribute descAttr =
  new Attribute("Description","A description for my queue");
mbeanServer.setAttribute(queueObjName, descAttr);
```

Some adapters such as the Sun HtmlAdaptorServer invoke the getAttribute() and setAttribute() methods recursively when getting or setting several attributes rather than invoking getAttributes() or setAttributes(). This may result in a high overhead. In this case, it would be a good idea to increase the cacheInterval attribute in the Admin MBean. Caching attribute values will decrease the amount of work being done by the adaptor.

setAttributes:

```
public AttributeList setAttributes( ObjectName name,
                                    AttributeList attribute)
  throws InstanceNotFoundException,
         ReflectionException;
```

This method is used to set or update several Attributes at once. The following example shows how to use the JMX-instrumented MQe queue MBean known by object name queueObjName, to set the Description and Expiry attributes at the agent level:

```
/*create the attributes to update */
Attribute descAttr =
  new Attribute("Description","A new description for my queue");
Attribute expiryAttr =
```

```
  new Attribute("Expiry", new Long(1000));

/*create the input parameter AttributeList   */
/* and add our Attributes to the List        */
AttributeList toUpdate = new AttributeList();
toUpdate.add(descAttr);
toUpdate.add(expiryAttr);

/* call setAttributes() and check results if required */
AttributeList updates = mbeanServer.setAttributes(queueObjName, toUpdate);
/* can now process updates as shown in getAttributes() */
```

Note: The same limitations apply to error handling for setAttributes as those described earlier for getAttributes().

invoke:

```
public Object invoke( ObjectName name,
                      String operationName,
                      Object[] params,
                      String[] signature)
  throws InstanceNotFoundException,
         MBeanException,
         ReflectionException;
```

This method is used to invoke JMX-wrappered MQe operations on MQe JMX-instrumented resources.

The input parameters are:

name
the ObjectName corresponding to the MQe resource to be administered.

operationName
the name of the operation to be invoked, for example: `addAlias`.

params
an array representing the input parameters to the operation.

signature
an array representing the data types corresponding to each parameter.

Note: The indices for entries in params and signature must correspond: the entry at index j in signature must represent the data type of the entry at index j in params.

Suppose you want to invoke the addAlias() method on a MQe queue represented by object name queueObjName where there is one input parameter of type String, which represents the alias name. The following example shows how to do this:

```
Object[] params = {new String("myAlias")};
String[] signature = {new String("java.lang.String")};

mbeanServer.invoke(queueObjName, "addAlias", params, signature);
```

In this case, there is no return value to worry about. However, although this is a relatively simple example, it illustrates the principles which apply to all operations invoked using this method.

Data types

In the example for the MBeanServer invoke() method in the previous subsection, the input parameter String[] signature represents the data types of all input parameters to the method being invoked.

In order to ensure compliance with the OpenMBean model, we only use the approved data types for our attribute getter() and setter() methods and for operation parameters. The data types specified when invoke() is called are therefore always limited to a set of approved types as follows:

- Simple data types:
  - java.lang.Void
  - java.lang.Boolean
  - java.lang.Byte
  - java.lang.Character
  - java.lang.String
  - java.lang.Short
  - java.lang.Integer
  - java.lang.Long
  - java.lang.Float
  - java.lang.Double
- Arrays of the above types:
  - javax.management.ObjectName
  - javax.management.openmbean.CompositeData
  - javax.management.openmbean.TabularData

The class name literals for each type have a specific format as follows:

- The simple data types listed above are returned as described, for example ″java.lang.Byte″
- For arrays of these types, the situation is more complex. For the purpose of the MQe JMX types, the only array types are of types java.lang.String and java.lang.Short. These array types are defined as follows:

Table 42. Data Types and Class Name Literal Strings

Data Type	Class Name Literal String
String[]	″[Ljava.lang.String;″
Short[]	″[Ljava.lang.Short;″

Note: Notice the semi-colon as the end of the class name. In all contexts where a data type has to be specified throughout the JMX instrumentation and agent layers, the class name literal format must be used.

Divergence from MQe Administration Interface

This section describes those aspects of the MQe JMX interface that differ in their implementation from the MQe administrative interface.

Messaging operations

The MQe JMX interface is intended as an administrative resource to assist in the configuration and management of MQe resources via JMX. Messaging operations do not therefore fall into its brief and such messaging operations as are provided are intended only for test purposes. putMessage() and deleteMessage() operations are provided for permitted queue types. These messaging operations provide a very limited scope for testing a network's connectivity and configuration. As the messaging operations are minimal it is not possible to test the operation of a network using store or forward queues.

The `putMessage()` method takes a single java.lang.String parameter representing the text of a message body. The user can only provide this single String – no further customization of the test message can take place. This message is put to the queue represented by the MBean upon which the method is invoked.

The MBean representing the queue in question also has an attribute called Messages (to qualify this statement, only MBeans representing queues of a type on which browse is permitted have this attribute). This attribute is of type [Ljava.lang.String; and can be retrieved using `getAttribute()`or `getAttributes()`. Each item in the String array represents the text of the message body put to the queue using the

putMessage() method. If messages are put to the queue in question by any means other than the JMX interface putMessage() method, the text body will not be readable by the JMX interface and the value returned for Messages will reflect this.

The index of a message body in the Messages array can be used to delete that message from the queue using the deleteMessage() method. The index is passed in as the only parameter to the deleteMessage() method.

Note that both the putMessage() and the deleteMessage() methods should only be invoked via the MBeanServer invoke() method. This is true for all operations listed in this section.

Store and Forward queues

In MQe, there is a single queue class, MQeStoreAndForwardQueue, which encompasses the functionality of both Store Queues and Forward Queues in the MQe JMX interface. This type of queue has the capacity to do both of the following:

- Forward messages either to the target queue manager (which MQe JMX calls ForwardToQMgr), 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. Using this approach messages are pulled from the store-and-forward queue. The target queue manager, in this case, is included in what MQe JMX calls the DestinationQMgrList.

MQeStoreAndForwardQueues have a property identifying their set of target queue managers (Queue_QMgrNameList).

In the case of the *Store Queue* MBean, there is no ForwardToQMgr. The sole purpose of this queue is to store messages for the queues in its DestinationQMgrList.

The *Forward Queue* MBean instance, by contrast, has a ForwardToQMgr as well as a DestinationQMgrList. Thus it has both the *forward* and *store* capabilities of the MQeStoreAndForwardQueue while the *Store Queue* just has the *store* capability.

This division of functionality between queue MBean representations is intended to simplify the roles of the queues in question. The Store Queue is, in effect, a ″storing″ queue without the ″forwarding″ capacity of the Forward Queue.

Programmatic interface versus user interface terminology

Queue references: When programming in MQe, you refer to different types of queues by the references used in this documentation.

When a queue is displayed in the JMX user interface, however, it is given a different reference. The following table shows the relationship between the user interface references and the programmatic references.

Table 43. Queue reference mapping

User interface queue reference	Programming interface queue reference	MQeAdminMessage class
Admin	Admin	MQeAdminQueueAdminMsg
Application	Local	MQeQueueAdminMsg
Async Proxy	Remote (where the mode is asynchronous)	MQeRemoteQueueAdminMsg
Bridge	Bridge	MQeBridgeQueueAdminMsg

Table 43. Queue reference mapping (continued)

User interface queue reference	Programming interface queue reference	MQeAdminMessage class
Forward	Store and Forward (different owning queue manager name from local queue manager)	MQeStoreAndForwardQueueAdminMsg
Home Server	Home Server	MQeHomeServerQueueAdminMsg
Proxy	Remote (incorporates both queue modes)	MQeRemoteQueueAdminMsg
Store	Store and Forward (same owning queue manager name as local queue manager)	MQeStoreAndForwardQueueAdminMsg
Sync Proxy	Remote (where the mode is synchronous)	MQeRemoteQueueAdminMsg

Queue queue manager references: When programming in MQe, you use the term *queueqm* — most queues have an associated queue manager and this is the *queueqm*. You refer to these different types of queue managers by the references used in this documentation.

However, when a *queueqm* is displayed in the JMX user interface it is given a different reference.

The following table shows how the *queueqm* is referred to for each type of queue, using the user interface references for both.

Table 44. Queueqm reference mapping

User interface queue reference	User interface queueqm reference	Description
Admin	none	Same as the local queue manager name
Application	none	Same as the local queue manager name
Async Proxy	DestinationQMgr	The name of the queue manager that holds the corresponding application queue
Bridge	DestinationQMgr	The name of the MQ queue manager that holds the corresponding MQ queue
Forward	ForwardToQMgr	The name of the queue manager that messages arriving on this queue will be forwarded to
Home Server	GetFromQMgr	The name of the queue manager that messages on store or forward queues will be pulled from
Store	none	Same as the local queue manager name
Sync Proxy	DestinationQMgr	The name of the queue manager that holds the corresponding application queue

Error handling

The MBeanException class is defined in the *Sun JMX Specification (1.2)* as follows:

This class represents ″user defined″ exceptions thrown by MBean methods in the agent. It ″wraps″ the actual ″user defined″ exception thrown. This exception will be built by the MBeanServer when a call to an MBean method results in an unknown exception.

There are methods in the MBeanException class that will return the original exception class and any message that was inside the exception:

```
public Exception getTargetException();
public Throwable getCause();
```

Therefore, an application can retrieve and handle any MQe (or other) exceptions.

However, it may be the case that exceptions caught at the agent layer may not be adequately displayed via the adapter or connector used. For example, the Sun RI HtmlAdaptorServer does not retrieve and display exceptions wrappered in MBeanExceptions.

This impacts the usefulness of using the HtmlAdaptorServer to get back MQeExceptions when setting attributes. For example, MQe throws an exception if you attempt to set a queue priority outside the range 0-9. All that the HtmlAdaptorServer shows is that the Priority attribute value has not been set. This is an unfortunate limitation to this specific adapter.

If a null value is entered for a required parameter on an operation a NullPointerException will be inside the wrapped MBeanException.

Any exception returning from MQe after an operation or an attempt to set an attribute is made will be of type MQeException inside the MBeanException.

Notifications

The JMX specification provides a notification mechanism which has been implemented in the MQe JMX interface.

This interface implements the JMX NotificationBroadcaster class and can therefore send notifications to any applications which implement the corresponding JMX NotificationListener class.

A notification in this context is a message sent by a notification broadcaster to a notification listener via the JMX infrastructure.

Notifications of two classes which subclass the JMX Notification class are sent from the MQe JMX interface. These classes are:

- com.ibm.mqe.jmx.MQeAliasNotification;
- javax.management.AttributeChangeNotification.

MQeAliasNotification

This class extends javax.management.Notification and provides the following notification types:

- mqe.connection.alias.added: when an alias is added to a connection.
- mqe.connection.alias.removed: when an alias is removed from a connection.
- mqe.queue.alias.added: when an alias is added to a queue.
- mqe.queue.alias.removed: when an alias is removed from a queue.
- mqe.queuemanager.alias.added: when an alias is added to a queue manager.
- mqe.queuemanager.alias.removed: when an alias is removed from a queue manager.

AttributeChangeNotification

This class is used to notify interested JMX listeners when the value of an MBean attribute changes. It provides the following notification type:

- jmx.attribute.changed: when the value of an attribute changes.

If an exception occurs when an attempt is made to change an attribute, the text of the exception will be passed back to the user via the notification 'message'. The getMessage() method can thus be used to

retrieve the exception text. The AttributeChangeNotification class also provides getOldValue() and getNewValue() methods to return the original attribute value and the value to which it is being changed. In the event of an error, getNewValue() will not return the actual attribute value (since the attempt to change the attribute has not succeeded) – in this case, getOldValue() returns the actual attribute value at the point of notification.

Using notifications

In order to receive notifications, a user (at the Agent layer) implements the JMX NotificationListener interface and then invokes the addNotificationListener on the MBeanServer instance:

```
public void addNotificationListener( ObjectName objName,
                                     NotificationListener listener,
                                     NotificationFilter filter,
                                     Object handback);
```

where

- `objectName` represents the MBean from which notifications are to be received
- `listener` is the user's instance of NotificationListener
- `filter` is an optional filter used if only a subset of possible notifications is required (may be null)
- `handback` is an object which can be used to hold private data that the handler of the received notification wants to access (may be null)

  Note: There is an alternative `addNotificationListener()` method on the MBeanServer which passes the ObjectName for the listener rather than the actual NotificationListener instance. This can be used if the NotificationListener instance is itself a registered MBean.

If you have any resources with aliases, and you add listeners for notifications from both a resource and its aliases, you will receive multiple identical notifications. It is a good idea to ensure that object names passed as parameters to the addNotificationListener() method do not contain the property key-value pair ″type=alias″.

Having called this API, the user's listener will now be added to the broadcaster's table of listeners.

In order to handle received notifications, the user also has to implement the following method:

```
public void handleNotification ( Notification notification,
                                 Object handback);
```

where:

- `notification` is the Notification instance sent by the NotificationBroadcaster object
- `handback` is an object which can be used to hold private data which the handler of the received notification wishes to access (may be null)

This method is where the received notifications are processed. The Notification class provides several useful methods which may be used to extract information about the notification:

```
public String getType();     // returns the notification type
public Object getSource();   // returns the source of the notification
public long   getSequence(); // returns the sequence number of
                             //  the notification [1]
public String getMessage();  // returns a text message associated with
                             //  the notification
public Object getUserData(); // returns the handback object
```

Note:

1. The sequence number provides information on the occurrence of the notification but is not set in this MQe JMX implementation so will always have a value of 0.

The following example shows how the agent could set up listeners for certain MQe MBeans. In this example, the user is only interested in receiving notifications from MBeans representing application queues belonging to the local queue manager TestQueueManager:

```
 /* find all the mbeans and set up listeners for them */
ObjectName scope = new
ObjectName("com.ibm.MQe_TestQueueManager_ApplicationQueues:*");
Set results = mbeanServer.queryNames(scope,null);
Iterator iter = results.iterator();
while(iter.hasNext()) {
   /* for each bean, check that it is not an alias MBean –
    * these beans have type=alias in the ObjectName        */
   ObjectName objName = (ObjectName)iter.next();
   String type = objName.getKeyProperty("type");

   if(type == null || !type.equals("alias")) {
      /*  add a listener */
      mbeanServer.addNotificationListener(objName,this,null,null);
      }
 }
```

Other Issues

Setting attributes of array type

It is possible for attributes of array type (for example, `[Ljava.lang.String;`) to be written to as well as read from. So, it is possible, for instance, to update a queue's array of aliases using the MQe JMX interface.

However, there are limitations to the manner in which some adapters allow the user to make such updates. For example, the Sun RI HtmlAdaptorServer adapter will only provide an array for the update which is of the same dimensions as the current array. Thus, if a queue has no existing aliases, the array for update will be of size zero, and hence no new alias can be added for the queue using the queue alias attribute. However, in this case, an alias can be added using the `addAlias()` operation.

If a queue has two existing aliases, then the array provided for the alias attribute update is of size two. One or both of the two aliases can be changed using the writable array. However, some adapters do not allow the user to clear the contents of the array cells and pass back an array containing empty string(s). This will cause an exception. Hence these adapters will only allow an update which keeps the number of existing aliases constant.

Since these are limitations of specific adapters only, we have decided to allow such array attributes to be updated where appropriate. The alternative would be to force users to use operations for adding aliases rather than using the attribute update potential.

This specific example may also be extended to situations where the capability of the adapter or connector does not match the capability of a programmatic interface. We are not in a situation to predict such limitations in advance and hence there may be features of our implementation which are not ideally suited to some adaptors and connectors. We have decided against constricting the functionality of our instrumentation layer to match the capabilities of specific adapters.

Alias MBeans

Certain MQe resources – queues, queue managers and connections – can have aliases, other names by which they can be known. In order to facilitate administration, the MQe JMX interface re-registers MBeans which have aliases under an object name corresponding to the alias. Thus for example if the JMX interface is used to add an alias `myAlias` to an application queue `myQueue`, the queue MBean is actually registered twice,

once with object name

`com.ibm.MQe_<OwningQMName>_ApplicationQueues:name=myQueue`

and once with object name

```
com.ibm.MQe_<OwningQMName>_ApplicationQueues:name=myAlias,
type=alias,
resourceName=myQueue@<OwningQMName>.
```

This means that the administrator does not have to be aware of the real name of the resource in order to administer it via JMX.

Likewise, when aliases are removed from resources, the corresponding ObjectName is de-registered.

One side-effect of this practice is that if a user chooses to create and register some MQe MBeans without using the helper method `createMQeMBeans()`, this may result in an inconsistent picture where some resources are registered with alias names while others are not. This enforces the argument for using the helper method to create and register all MQe MBeans.

Translation

Description attributes for MQe resources are available in all the different MQe supported languages, in translated properties files. The language used for the descriptions will be selected according to the default locale of your machine, using the normal Java convention, as described briefly below.

You must provide appropriate properties files on your classpath for all languages, **including US English**.

The properties files can be found in the folder `Java/com/ibm/mqe/properties` below the folder in which MQe was installed. This directory contains two required properties files in all the supported languages:

- `AdminDescBundle`
- `JMXDescBundle`

Either all the properties files can be added to the classpath or just the files required for your language.

There are two default files:

- `AdminDescBundle.properties`
- `JMXDescBundle.properties`

that contain the English descriptions. These files will be used if there is no matching translated file for your locale.

All other files have a country code XX appended to the first part of the filename to make `JMXDescBundle_`*`XX`*`.properties` where *XX* is one of the following country codes:

Code	Language
de	German
es	Spanish
fr	French
it	Italian
ja	Japanese
ko	Korean
pt_BR	Brazilian Portuguese
zh	Chinese
zh_TW	Traditional Chinese

The Java system for selecting the language is as follows:

1. determine the computer's default locale, for example `fr_FR`
2. Search for that locale through the files provided, checking the language code on each:
   - If a fully-qualified language code file, for example `file_fr_FR` is **not** found, then it will use the semi-qualified code file, if it exists — in that example `file_fr`
   - When a file with a fully-qualified code **is** found, only the fully matching locale will select it, for example locale `pt_BR` will use the file `file_pt_BR`, but locale `pt_PT` will not, and will hence default to English.

Note: The JTC recommend that when a fully-qualified language code file exists, the semi-qualified code file must also exist, even if it is empty.

For full information on this, see these websites:

```
http://oss.software.ibm.com/icu/userguide/design.html
http://java.sun.com/products/jdk/1.2/docs/api/java/util/ResourceBundle.html
```

Related information on JMX

This section lists various sources of information that you might find useful.

Books

- Perry, J. Steven, Java Management Extensions: Managing Java Applications with JMX (O'Reilly & Associates, Inc: 2002)
- Jasnowski, Mike. JMX Programming (Wiley Publishing, Inc: 2002)

Articles

- http://java.sun.com/products/JavaManagement/ (Sun articles, downloads, resources for JMX)
- http://mx4j.sourceforge.net/ (Open Source JMX http://mx4j.sourceforge.net/)
- http://www-106.ibm.com/developerworks/library/j-jmx1/
- http://www-106.ibm.com/developerworks/library/j-jmx2/
- http://www-106.ibm.com/developerworks/library/j-jmx3/ (Article: From black boxes to enterprises Parts 1-3)
- http://www.informit.com/content/index.asp?session_id={61F6D302-4F4F-4D68-96D8-AD545B00CA28}&product_id={583CB44A-AC47-4959-9C01-FA8DF0884EEE} (Article: Managing Complex systems with JMX)
- http://jcp.org/aboutJava/communityprocess/maintenance/jsr003/jmx1.2-change-log.txt (Changes between JMX 1.1 and 1.2)

Other resources

- http://www.alphaworks.ibm.com/tech/TMX4J (IBM Tivoli Tivoli's implementation of the JMX Specification)
- http://www.jguru.com/forums/JMX (JGuru JMX forum).

Index