IBM

WebSphere MQ Everyplace V2.0.2

Contents

Designing your real application

Messaging

Overview of MQe messaging

The MQe programming model uses several entities, for example messages, queues, and queue managers, that work together as a flexible toolkit. Each entity has a specific purpose and works together with other entities to provide solutions for message topologies.

What are MQe messages?

Introduction to the use of MQe messages

Messages are collections of data sent by one application and intended for another application. MQe messages contain application-defined content. When stored, they are held in a queue and such messages may be moved across an MQe network.

MQe messages are a special type of MQeFields items, as described in “MQeFields” on page 6. Therefore, you can use methods that are applicable to MQeFields with messages.

Therefore, messages are Fields objects with the addition of some special fields. Java™ provides a subclass of MQeFields, MQeMsgObject which provides methods to manage these fields. The C code base does not provide such a subclass. Instead, there are a number of mqeFieldsHelper_operation functions. The following fields form the `Unique ID` of an MQe message:

- In Java, the timestamp, generated when the message is first created or, in C, when the message is first put to a queue
- The name of the queue manager, to which the message is first put.

The `Unique ID` identifies a message within an MQe network provided all queue managers within the MQe network are named uniquely. However, MQe does not check or enforce the uniqueness of queue manager names.

In Java, the message is created when an instance of MQeMsgObject is created. In C, the Message is "created", that is UniqueID fields are added, when the message is put to a queue.

The mqeMsg_getMsgUIDFields()method or mqeFieldsHelpers_getMsgUidFields() function accesses the `UniqueID` of a message, for example:

Java code

C code

```
        rc = mqeFieldsHelpers_getMsgUidFields(hMgsObj,
                        &exceptBlock,&hUIDFields);
```

MQe adds property related information to a message (and subsequently removes it) in order to implement messaging and queuing operations. When sending a message between queue managers, you can add resend information to indicate that data is being retransmitted.

Typical application-based messages have additional properties in accordance with their purpose. Some of these additional properties are generic and common to many applications, such as the name of the reply-to queue manager.

Message properties

Table of MQe message properties

MQe supports the following message properties:

Table 1. Message properties

Property name	**Java type**	**C type**	**Description**
Action	int	MQEINT32	Used by administration to indicate actions such as inquire, create, and delete
Correlation ID	byte[]	MQEBYTE[]	Byte string typically used to correlate a reply with the original message
Errors	MQeFields	MQeFieldsHndl	Used by administration to return error information
Expire time	int or long	MQEINT32 or MQEINT64	Time after which the message can be deleted (even if it is not delivered)
Lock ID	long	MQEINT64	The key necessary to unlock a message
Message ID	byte[]	MQEBYTE[]	A unique identifier for a message
Originating queue manager	string	MQeStringHndl	The name of the queue manager that sent the message
Parameters	MQeFields	MQeFieldsHndl	Used by administration to pass administration details
Priority	byte	MQEBYTE	Relative order of priority for message transmission
Reason	string	MQeStringHndl	Used by administration to return error information
Reply-to queue	string	MQeStringHndl	Name of the queue to which a message reply should be addressed
Reply-to queue manager	string	MQeStringHndl	Name of the queue manager to which a message reply should be addressed
Resend	boolean	MQEBOOL	Indicates that the message is a resend of a previous message
Return code	byte	MQEBYTE	Used by administration to return the status of an administration operation
Style	byte	MQEBYTE	Distinguishes commands from request/reply for example
Wrap message	byte[]	MQEBYTE[]	Message wrapped to ensure data protection

Symbolic names:

Table of symbolic names corresponding to MQe message properties

The following table lists the symbolic names corresponding to the MQe message properties:

Table 2. Symbolic names that correspond to message property names

Property name	**Java constant**	**C constant**
Action	MQeAdminMsg.Admin_Action	MQE_ADMIN_ACTION
Correlation ID	MQe.Msg_CorrelID	MQE_MSG_CORRELID
Errors	MQeAdminMsg.Admin_Errors	MQE_ADMIN_ERRORS
Expire time	MQe.Msg_ExpireTime	MQE_MSG_EXPIRETIME
Lock ID	MQe.Msg_LockID	MQE_MSG_LOCKID
Message ID	MQe.Msg_MsgID	MQE_MSG_MSGID

Table 2. Symbolic names that correspond to message property names (continued)

Property name	Java constant	C constant
Originating queue manager	MQe.Msg_OriginQMgr	MQE_MSG_ORIGIN_QMGR
Parameters	MQeAdminMsg.Admin_Params	MQE_ADMIN_PARAMS
Priority	MQe.Priority	MQE_MSG_PRIORITY
Reason	MQeAdminMsg.Admin_Reason	MQE_ADMIN_REASON
Reply-to-queue	MQe.Msg_ReplyToQ	MQE_MSG_REPLYTO_Q
Reply-to queue manager	MQe.Msg_ReplyToQMgr	MQE_MSG_REPLYTO_QMGR
Resend	MQe.Msg_Resend	MQE_MSG_RESEND
Return code	MQeAdminMsg.Admin_RC	MQE_ADMIN_RC
Style	MQe.Msg_Style	MQE_MSG_STYLE
Wrap message	MQe.Msg_WrapMsg	MQE_MSG_WRAPMSG

Examples:

Message Properties - Examples

In all cases, a defined constant allows the property name to be carried in a single byte. For example, priority (if present) affects the order in which messages are transmitted, correlation ID triggers indexing of a queue for fast retrieval of information, expire time triggers the expiry of the message, and so on. Also, the default message dump command minimizes the size of the generated byte string for more efficient message storage and transmission.

The MQe `Message ID` and `Correlation ID` allow the application to provide an identity for a message. These are also used in interactions with the rest of the MQ family:

Java

```
MQeMsgObject msgObj = new MQeMsgObject;
msgObj.putArrayOfByte( MQe.Msg_ID, MQe.asciiToByte( "1234" ));
```

C

```
rc = mqeFields_putArrayOfByte(hMsg,&exceptBlock,
          MQE_MSG_MSGID,pByteArray,sizeByteArray);
```

`Priority` contains message priority values. Message priority is defined as in other members of the MQ family. It ranges from 9 (highest) to 0 (lowest):

Java

```
MQeMsgObject msgObj = new MQeMsgObject();
msgObj.putByte( MQe.Msg_Priority, (byte)8 );
```

C

```
rc = mqeFields_putByte(hsg,&exceptBlock, MQE_MSG_PRIORITY, (MQEBYTE)8);
```

Applications can create fields for their own data within messages:

Java

```
MQeMsgObject msgObj = new MQeMsgObject();
msgObj.putAscii( "PartNo", "Z301" );
msgObj.putAscii( "Colour", "Blue" );
msgObj.putInt( "Size", 350 );
```

C

```
        MQeFieldsHndl hPartMsg;
        MQeStringHndl hSize_FieldLabel;
        rc = mqeFields_new(&exceptBlock,&hPartMsg);
        rc = mqeString_newUtf8(&exceptBlock,
                        &hSize_FieldLabel,"Size");

        rc = mqeFields_putInt32(hPartMsg,
                      &exceptBlock,hSize_FieldLabel,350);
```

The priority of the message is used, in part, to control the order in which messages are removed from the queue. If the message does not specify any, then the queue default priority is used . This, unless changed, is 4. However, the application must interpret the different levels of priority.

In Java, you can extend the MQeMsgObject to include some methods that assist in creating messages, as shown in the following example:

```
package messages.order;
import com.ibm.mqe.*;

/*** This class defines the Order Request format */
public class OrderRequestMsg extends MQeMsgObject
{

  public OrderRequestMsg() throws Exception
  {
  }

 /*** This method sets the client number */
  public void setClientNo(long aClientNo) throws Exception
  {
    putLong("ClientNo", aClientNo);
  }

 /*** This method returns the client number */
  public long getClientNo() throws Exception
  {
    return getLong("ClientNo");
  }
```

To find out the length of a message, you can enumerate on the message as each data type has methods for getting its length.

Message filters

Introduction to MQe message filters

Filters allow MQe to perform powerful message searches. Most of the major queue manager operations support the use of filters. You can create filters using MQeFields.

Using a filter, for example in a getMessage() call, causes an application to return the first available message that contains the same fields and values as the filter. The following examples create a filter that obtains the first message with a message id of ″1234″:

Java

```
        MQeFields filter = new MQeFields();
        filter.putArrayOfByte( MQe.Msg_MsgID,
                MQe.AsciiToByte( "1234" ) );
```

C rc = mqeFields_putArrayOfByte(hMsg, &exceptBlock, MQE_MSG_MSGID, pByteArray, sizeByteArray);

You can use this filter as an input parameter to various API calls, for example `getMessage`.

Message expiry

Overview of the expiry of messages in queues

Queues can be defined with an expiry interval. If a message has remained on a queue for a period of time longer than this interval then the message is automatically deleted. When a message is deleted, a queue rule is called. This rule cannot affect the deletion of the message, but it does provide an opportunity to create a copy of the message.

Messages can also have an expiry interval that overrides the queue expiry interval. You can define this by adding a C `MQE_MSG_EXPIRETIME` or Java `MQe.Msg_ExpireTime` field to the message. The expiry time is either relative (expire 2 days after the message was created), or absolute (expire on November 25th 2000, at 08:00 hours). Relative expiry times are fields of type Int or MQEINT32, and absolute expiry times are fields of type Long or MQEINT64.

In the example below, the message expires 60 seconds after it is created (60000 milliseconds = 60 seconds).

```
/* create a new message         */
MQeMsgObject msgObj = new MQeMsgObject();
msgObj.putAscii( "MsgData", getMsgData() );
/* expiry time of sixty seconds after message was created    */
msgObj.putInt( MQe.Msg_ExpireTime, 60000 );
```

In the example below, the message expires on 15th May 2001, at 15:25 hours.

```
/* create a new message         */
MQeMsgObject msgObj = new MQeMsgObject();
msgObj.putAscii( "MsgData", getMsgData() );
/* create a Date object for 15th May 2001, 15:25 hours      */
Calendar calendar = Calendar.getInstance();
calendar.set( 2001, 04, 15, 15, 25 );
Date expiryTime = calendar.getTime();
/* add expiry time to message       */
msgObj.putLong( MQe.Msg_ExpireTime, expiryTime.getTime() );
/* put message onto queue       */
qmgr.putMessage( null, "MyQueue", msgObj, null, 0 );
```

To set a relative expiry time use the following on a message handle:

```
mqeFields_putInt32(pErrorBlock, hMsg, relativeTime);
```

To set an absolute expiry time use:

```
mqeFields_putInt64(pErrorBlock,hMsg, absoluteTime);
```

All Times are in milliseconds

Checking for expired messages:

Explanation of when MQe checks for expired messages

A message is checked for expiry when:

It is added to a queue
Expiry can occur when a message is added from the local API, pulled down via a Home Server Queue, or pushed to a queue.

It is removed from a queue
Expiry can occur when a message can be removed from the local API, or when a message is pulled remotely.

A queue is activated
When a queue is activated, a reference to the queue is created in memory. Any message that has expired is removed. The state of the message is irrelevant to this operation.

A queue is deleted
If an admin message arrives to delete a queue, the queue must be empty first. Therefore, before this check is done, any expired messages are removed from the queue. The state of the message is irrelevant to this operation.

A queue is checked for size
If an admin message arrives to inquire on the size of a queue, the queue is first purged of admin messages.

You can add a queue rule to notify you when messages expire. However, in a certain situation between two queue managers, a message may seem to expire twice. This is not because the message has been duplicated, but is outlined in the following paragraph.

Assume that an asynchronous queue has a message on it due to expire at 10:00 1st Jan 2005. All messages on such queues are transmitted using a 2 stage process. This process is equivalent to a putMessage and confirmPutMessage pair of operations. Suppose that the first transmission stage occurs at 09:55. A reference to the message appears on the remote queue manager. However, it is not yet available to an application on that queue manager. Then, if the network fails until 10:05, the expiry time of the message is missed. Therefore, the message expires on the remote queue and the queue expiry rule gets fired. Also, in due course, the queue expiry rule gets fired on the destination queue manager.

Assurance of expiry:

Explains how to ensure message expiry

The expiry time can be calculated to the millisecond. For correct operation the clocks of the machines running the queue managers must be accurately aligned. Failure to do this within accuracy determined by your choice of expiry times causes messages to appear active on one queue manager, while they have expired on others. Ensure that you use the correct field type for the expiry value. An int (32 bit) field is used for relative expiry times, and a long (64 bit) field is used for absolute times. The field name is the same in both cases.

MQeFields

Overview of the MQeFields container structure

MQeFields is a container data structure widely used in MQe. You can put various types of data into the container. It is particularly useful for representing data that needs to be transported, such as messages. The following code creates an MQeFields structure:

Java code
```
  /* create an MQeFields  object      */
  MQeFields fields = new MQeFields( );
```

C code
```
MQeFieldsHndl hFields;
  rc = mqeFields_new(&exceptBlock, &hFields);
```

MQeFields contains a collection of orderless fields. Each field consists of a triplet of entry name, entry value, and entry value type. MQeFields forms the basis of all MQe messages.

Use the entity name to retrieve and update values. It is good practice to keep names short, because the names are included with the data when the MQeFields item is transmitted.

The name must:

- Be at least 1 character long
- Conform to the ASCII character set (characters with values 20 < value < 128)
- Exclude any of the characters { } [ ] # ( ) : ; , ′ ″ =
- Be unique within MQeFields

Storage and retrieval of values in MQeFields

Examples of storing values in an MQeFields item, and retrieving values from an MQeFields item

The following example shows how to store values in an MQeFields item:

Java code

```
/* Store integer values into a fields object    */
  fields.putInt( "Int1", 1234 );
  fields.putInt( "Int2", 5678 );
  fields.putInt( "Int3",    0 );
```

C code

```
MQeStringHndl hFieldName;
  rc = mqeString_newChar8(&errStruct,  &hFieldName, "A Field Name");
  rc = mqeFields_putInt32(hNewFields,&errStruct,hFieldName,1234);
```

The following example shows how to retrieve values from an MQeFields item:

Java code

```
/* Retrieve an integer value from a fields object    */
  int Int2 = fields.getInt( "Int2" );
```

C code

```
MQEINT32 value;
  rc = mqeFields_getInt32(hNewFields, &errStruct, &value, hFieldName);
```

MQe provides methods for storing and retrieving the following data types:

- A fixed length array is handled using the putArrayOf*type* and getArrayOf*type* methods, where *type* can be Byte, Short, Int, Long, Float, or Double.
- The ability to store variable length arrays is possible, but has been deprecated in this release. You can access these arrays using the Java put*type*Array and get*type***Array** calls or the C put*type* calls.
- The Java code base has a slightly special form of operations for Float and Double types. This provides compatibility with the MicroEdition. Floats are put using an Int representation and Doubles are put using a Long representation. Use the `Float.floatToIntBits()` and `Double.doubleToLongBits()` to perform the conversion. However, this is not required on the C API.

Embedding MQeFields items

Description of how to embed an MQeFields item within another MQeFields item

An MQeFields item can be embedded within another MQeFields item by using the putFields and getFields methods.

The contents of an MQeFields item can be dumped in one of the following forms:

binary Binary form is normally used to send an MQeFields or MQeMsgObject object through the network. The dump method converts the data to binary. This method returns a binary byte array containing an encoded form of the contents of the item.

Note: This is not Java serialization.

When a fixed length array is dumped and the array does not contain any elements (its length is zero), its value is restored as null.

encoded string (Java only)

The string form uses the dumpToString method of the MQeFields item. It requires two parameters, a template and a title. The template is a pattern string showing how the MQeFields item data should be translated, as shown in the following example:

```
"(#0)#1=#2\r\n"
```

where

#0	is the data type (ascii or short, for example)
#1	is the field name
#2	is the string representation of the value

Any other characters are copied unchanged to the output string. The method successfully dumps embedded MQeFields objects to a string, but due to restrictions, the embedded MQeFields data may not be restored using the restoreFromString method.

Queues

Overview of MQe queues

What are MQe queues?

Introduction to MQe queues

MQe queues store messages. The queues are not directly visible to an application and all interactions with the queues take place through queue managers. Each queue manager can have queues that it manages and owns. These queues are known as *local* queues. MQe also allows applications to access messages on queues that belong to another queue manager. These queues are known as *remote* queues. Similar sets of operations are available on both local and remote queues, with the exception of defining message listeners. Refer to "Message listeners" on page 44 for more information. The Queue types section provides more information on the different types of queue you can have.

Messages are held in the queue's persistent store. A queue accesses its persistent store through a *queue store adapter*. These adapters are interfaces between MQe and hardware devices, such as disks or networks, or software stores such as a database. Adapters are designed to be pluggable components, allowing the protocols available to talk to the device to be easily changed.

Queues may have characteristics, such as authentication, compression and encryption. These characteristics are used when a message object is stored on a queue.

Queue names

Constraints of MQe queue names

MQe queue names can contain the following characters:
- Numerics 0 to 9
- Lower case a to z
- Upper case A to Z
- Underscore _
- Period .
- Percent %

There are no inherent name length limitations in MQe.

Queues are configured using administration messages.

Queue properties

Table of MQe queue properties

Queue properties are shown in the following table. Not all the properties shown apply to all the queue types:

Field Name provided as a static string	C Static String - MQe_Queue_Constants.h	Explanation	Class to be used for MQeField value	Static string value for field name
Admin_Class		Queue class	String	admtype
Admin_Name		ASCII queue name	String	admname
Queue_Active	MQE_QUEUE_ACTIVE	Queue in active/inactive state	boolean	qact
Queue_AttRule		Rule class controlling security operations	String	qar
Queue _Authenticator	MQE_QUEUE_AUTHENTICATOR	Authenticator class	String	qau
Queue_BridgeName		Owning MQ bridge name - bridge only	String	q-mq-bridge
Queue_Client-Connection		Client connection name - bridge only	String	q-mq-client-con
Queue_CloseIdle		Close the connection to the remote queue manager once all messages have been transmitted	boolean	qcwi
Queue_CreationDate	MQE_QUEUE_CREATIONDATE	Date the queue was created	long	qcd
Queue_Compressor	MQE_QUEUE_COMPRESSOR	Compressor class	String	qco
Queue_Cryptor	MQE_QUEUE_CRYPTOR	Cryptor class	String	qcr
Queue_CurrentSize	MQE_QUEUE_CURRENTSIZE	Number of messages on the queue	int	qcs
Queue_Description	MQE_QUEUE_DESCRIPTION	Unicode description	String	qd
Queue_Expiry	MQE_QUEUE_EXPIRY	Expiry time for messages		qe
Queue_FileDesc	MQE_QUEUE_FILEDESC	File descriptor, specifies the type of message store	String	qfd
Queue_MaxIdletime		Maximum time to keep a connection idle - bridge only	int	q-mq-max-idle-time
Queue_MaxMsgSize	MQE_QUEUE_MAXMSGSIZE MQE_QUEUE_NOLIMIT	Maximum length of messages allowed on the queue	int	qms
Queue_MaxQSize	MQE_QUEUE_MAXQSIZE MQE_QUEUE_NOLIMIT	Maximum number of messages allowed	int	qmqs
Queue_Mode	MQE_QUEUE_MODE MQE_QUEUE_SYNCHRONOUS MQE_QUEUE_ASYNCHRONOUS	Synchronous or asynchronous	byte Queue_Synchron-ous Queue_Asynchro-nous	qm

Field Name provided as a static string	C Static String - MQe_Queue_Constants.h	Explanation	Class to be used for MQeField value	Static string value for field name
Queue_MQQMgr		MQ queue manager proxy - bridge only	String	q-mq-q-mgr
Queue_Priority	MQE_QUEUE_PRIORITY	Priority to be used for messages (unless overridden by a message value)	byte	qp
Queue_QAlias-NameList	MQE_QUEUE_QALIASNAMELIST	Alterantive names for the queue	String[]	qanl
Queue_QMgrName	MQE_QUEUE_QMGRNAME	Queue manager owning the real queue	String	qqmn
Queue_QMgr-NameList	MQE_QUEUE_QMGRNAMELIST - for admin only, C does not support store queues	Queue manager targets - used in store queues	String[] -	qqmnl
Queue_Remote-QName		Remote MQ field name - bridge only	String	q-mq-remote-q
Queue_Rule		Rule class for queue properties	String	qr
Queue_QTimer-Interval		Delay before processing pending messages on Home Server Queue - use Rule for trigger transmission instead *	long	qti
Queue_Target-Registry	MQE_QUEUE_TARGETREGISTRY	Target registry tupe	String[] possible values: Queue_Registry-None Queue_Registry-QMgr Queue_Registry-Queue	qtr
Queue_Transporter	MQE_QUEUE_TRANSPORTER MQE_QUEUE_DEFAULT-TRANSPORTER	Transporter class	String - use: Queue_Default-Transporter	qtc
Queue_Transporter-XOR		Transporter to use XOR compression	boolean	qtxor
Queue_Transformer		Transformer class	String	q-mq-transformer

* If a timer interval is used on the HomeServer queue if an error occurs, the application never knows the thread has stopped, and therefore cannot do anything about it. Instead, the timer interval should be set to zero and a rule on the queue manager used to loop and explicitly call the triggerTransmission(). It is wise not to set the loop too tight but to set the timer on the loop to a sensible value so messages are still sent/retrieved without extraneous CPU being used.

Queue types

Introduction to MQe queue types

There are several different types of *queues* that you can use in an MQe environment.

Local queue

The simplest type of queue is a local queue. This type of queue is local to, and owned by, a specific queue manager. It is the final destination for all messages. Applications on the owning queue manager can interact directly with the queue to store messages in a safe and secure way, excluding hardware failures or loss of the device.

You can use local queues either online or offline, either connected or not connected to a network. Queues can also have security attributes set, in a very similar manner to protecting messages with attributes.

Access to messages on local queues is always synchronous, which means that the application waits until MQe returns after completing the operation, for example a put, get, or browse operation.

The queue owns access and security and may allow a remote queue manager to use these characteristics, when connected to a network. This allows others to send or receive messages to the queue.

Remote queue

A remote queue is a local queue belonging to another queue manager. This remote queue definition exchanges messages with the remote local queue.

MQe can establish remote queues automatically. If you attempt to access a queue on another queue manager, for example to send a message to that queue, MQe looks for a remote queue definition. If one exists it is used. If not, *queue discovery* occurs.

Note: The concept of queue discovery does not apply to the C code base.

MQe discovers the authentication, cryptography, and compression characteristics of the real queue and creates a remote queue definition. Such queue discovery depends upon the target being accessible. If the target is not accessible, a remote definition must be supplied in some other way. When queue discovery occurs, MQe sets the access mode to synchronous, because the queue is now known to be synchronously available.

Synchronous remote queues are queues that can be accessed only when connected to a network that communicates with the owning queue manager. If the network is not established, the operations return an error. 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, because, in this case, MQe is no longer responsible for once and once-only assured delivery.

Asynchronous remote queues are queues used to send messages to remote queues and can store messages pending transmission. They cannot remotely retrieve messages. If the network connection is established, messages are sent to the owning queue manager and queue. However, if the network is not connected, messages are stored locally until there is a network connection and then the messages are transmitted. This allows applications to operate on the queue when the device is offline. As a result, these queues temporarily store messages at the sending queue manager while awaiting transmission.

Store-and-forward queue

Note: Store-and-forward queues are not implemented in the C code base.

A store-and-forward queue stores messages on behalf of one or more remote queue managers until they are ready to receive them. This can be configured to perform either of the following:

- Push messages either to the target queue manager or to another queue manager between the sending and the target queue managers.
- Wait for the target queue manager to pull messages destined for it.

A store-and-forward queue stores messages associated with one or more target queue manager destinations. Messages addressed to a specific or target queue manager are placed on the relevant store-and-forward queue. The store-and-forward queue can optionally have a forwarding queue manager name set. If this name is set, the queue attempts to send all its messages to that named queue manager. If the name is not set, the queue just holds the messages.

Note: A store-and-forward queue and a *home server queue* should not have the same target queue manager. A store-and-forward queue with a queue QueueManagerName that is not the same as its host QueueManagerName, attempts to push messages to the remote queue manager. If that remote queue manager has a home server queue, it may attempt to pull the same message simultaneously, causing the message to lock.

Store-and-forward queues can hold messages for many target queue managers, or there may be one store-and-forward queue for each target queue manager.

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

Dead-letter queue

MQe has a similar dead-letter queue concept to MQ. Such queues store messages that cannot be delivered. However, there are important differences in the manner in which they are used.

- In MQ, if a message is being moved from queue manager `A` to queue manager `B`, then if the target queue on queue manager `B` cannot be found, the message can be placed on the *receiving queue manager's* (`B`'s) dead-letter queue.
- In MQe, if home-server queue on a client pulls a message from a server and is not able to deliver the message to a local queue and the client has a dead letter queue, the message will be placed on the client's dead letter queue.

  Note: In C, the Dead letter queue is just a local queue with a specific name.

  The use of dead-letter queues with an MQ bridge needs special consideration. For more information, see "MQ bridge queue" on page 13.

Administration queue

The administration queue is a specialized queue that processes administration messages.

Messages put to the administration queue are processed internally. Because of this applications cannot get messages directly from the administration queue. Only one message is processed at a time, other messages that arrive while a message is being processed are queued up and processed in the sequence in which they arrive.

Home-server queue

This type of queue usually resides on a client and points to a store-and-forward queue on a server known as the *home-server*. The home-server queue pulls messages from the home-server store-and-forward queue when the client connects on the network.

In Java, home-server queues normally have a polling interval that causes them to check for any pending messages on the server while the network is connected.

When this queue pulls a message from the server, it uses assured message delivery to put the message to the local queue manager. The message is then stored on the target queue.

Home-server queues have an important role in enabling clients to receive messages over client-server connections.

MQ bridge queue

Note: The C code base does not support MQ bridge queues.

This type of queue is always defined on an MQe gateway queue manager and provides a path from the MQe environment to the MQ environment. The MQ bridge queue is a remote queue definition that refers to a queue residing on an MQ queue manager.

Applications can use **put**, **get**, and **browse** operations on this type of queue, as if it were a local MQe queue.

Queue persistent storage

Overview of MQe message stores

Local queues and asynchronous remote queues store messages and therefore have properties to determine how and where the messages are stored.

The message store determines how the messages are mapped to the storage medium. The C and Java versions of MQe support a default message store, allowing long file names. The Java version of MQe has two additional message stores, `MQeShortFilenameMessageStore` that ensures the file name does not exceed eight characters, and the `MQe4690ShortFilenameMessageStore` that supports the default file system on a 4690. A storage adapter provides the message store access to the storage medium, the Java and C versions of MQe provide disk adapters with the Java version also providinges a case insensitive adapter and a memory adapter.

The backing store used by a queue can be changed using an MQe administration message. Changing the backing store is not allowed while the queue is active or contains messages. If the backing store used by the queue allows the messages to be recovered in the event of a system failure, then this allows MQe to assure the delivery of messages.

Using queue aliases

Introduces the use of queue aliases

Aliases can be assigned for MQe queues to provide a level of indirection between the application and the real queues. Hence the attributes of a queue that an alias relates to can be changed without the application needing to change. For instance, a queue can be given a number of aliases and messages sent to any of these names will be accepted by the queue.

Examples of queue aliasing

Illustrates some of the ways in which aliasing can be used with queues

The following examples illustrate some of the ways in which aliasing can be used with queues:

Merging applications:

Using queue aliasing to merge applications

Suppose you have the following configuration:
- A client application that puts data to queue Q1
- A server application that takes data from Q1 for processing
- A client application that puts data to queue Q2
- A server application which takes data from Q2 for processing

Some time later the two server applications are merged into one application supporting requests from both the client applications. It may now be appropriate for the two queues to be changed to one queue. For example, you may delete Q2, and add an alias of the Q1 queue, calling it Q2. Messages from the client application that previously used Q2 are automatically sent to Q1.

Upgrading applications:

Using queue aliasing to upgrade applications

Suppose you have the following configuration:
- A queue Q1
- An application that gets messages from Q1
- An application that puts messages to Q1

You then develop a new version of the application that gets the messages. You can make the new application work with a queue called Q2. You can define a queue called Q2 and use it to exercise the new application. When you want it to go live, you let the old version clear all traffic off the Q1 queue, and then create an alias of Q2 called Q1. The application that puts to Q1 will still work, but the messages will end up on Q2.

Using different transfer modes to a single queue:

Using different transfer modes to a single queue, using queue aliasing

Suppose you have a queue MY_Q_ASYNC on queue manager MQE1. Messages are passed to MY_Q_ASYNC by a different queue manager MQE2, using a remote queue definition that is defined as an asynchronous queue. Now suppose your application periodically wants to get messages in a synchronous manner from the MY_Q_ASYNC queue.

The recommended way to achieve this is to add an alias to the MY_Q_ASYNC queue, perhaps called MY_Q_SYNC. Then define a remote queue definition on your MQE2 queue manager, that references the MY_Q_SYNC queue. This provides you with two remote queue definitions. If you use the MY_Q_ASYNC definition, the messages are transported asynchronously. If you use the MY_Q_SYNC definition, synchronous message transfer is used.

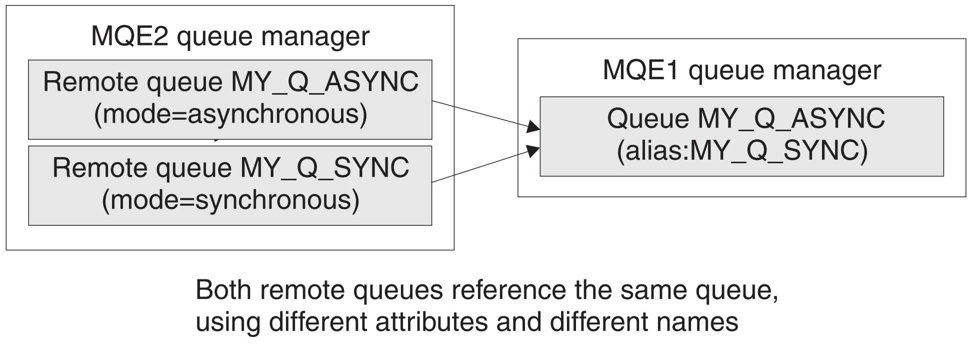

MQE2 queue manager
Remote queue MY_Q_ASYNC (mode=asynchronous)
Remote queue MY_Q_SYNC (mode=synchronous)
MQE1 queue manager
Queue MY_Q_ASYNC (alias:MY_Q_SYNC)
Both remote queues reference the same queue, using different attributes and different names

Figure 1. Two modes of transfer to a single queue

MQe connection definitions

Explains how logical connections between queue managers are established

MQe supports a method of establishing logical connections between queue managers, in order to send or receive data.

MQe clients and servers communicate over connections called *client/server channels*.

Client/server channels have the following attributes:
- They are *dynamic*, that is created on demand. This differentiates them from MQ connections which have to be explicitly created.
- You can only establish the connection from the client-side.
- A client can connect to many servers, with each connection using a separate channel.
- The server-side queue manager can accept many connections simultaneously, from a multitude of different clients, using a listener for each protocol.
- They work through a Firewall, if the server-side of the connection is behind the Firewall. However, this depends on the configuration of the Firewall.
- They are *unidirectional* and support the full range of functions provided by MQe, including both synchronous and asynchronous messaging.

  Note: Unidirectional means that the client can send data to, or request data from the server, but the server-side cannot initiate requests of the client.

Standard connections, used for the client/server connection style, are unidirectional, but depend on a listener at the server, as servers cannot initiate data transfer. The client initiates the connection request and the server responds. A server can usually handle multiple incoming requests from clients. Over a standard connection, the client has access to resources on the server. If an application on the server needs synchronous access to resources on the client, a second connection is required where the roles are reversed. However, because standard connections are themselves bidirectional, messages destined for a client from its server's transmission queue, are delivered to it over the standard (client/server) connection that it initiated.

A client can be a client to multiple servers simultaneously. The client/server connection style is generally suited for use through Firewalls, because the target of the incoming connection is normally identified as being acceptable to the Firewall.

Note: Supposing there are two server queue managers, SQM1 and SQM2. SQM2 has listener address host 2: 8082. Also, suppose that SQM1 has a connection to SQM2 and a listener addresss, host 1:8081. If you create a connection definition on a client queue manager, named SQM2 with address host 1: 8081, this transports commands for SQM2 to SQM1, which then transports them to SQM2. Avoid this construct, as it is inefficient.

Because of the way channel security works, when a specific attribute rule is specified for a target queue, it forces the local queue manager to create an instance of the same attribute rule, `examples.rules.AttributeRule` and `com.ibm.mqe.MQeAttributeRule` are treated as the same rule. If this is not a desirable behaviour, you can specify a null rule for the target queue. In this case, com.ibm.mqe.`MQeAttributeDefaultRule` takes effect.

Connections can have various attributes or characteristics, such as authentication, cryptography, compression, or the transmission protocol to use. Different connections can use different characteristics. Each connection can have its own value set for each of the following attributes:

Authenticator
This attribute causes authentication to be performed. This is a security function that challenges the putting application environment or user to prove their identity. It has a value of either `NULL` or an *authenticator* that can perform user or connection authentication.

Cryptor
This attribute causes encryption and decryption to be performed on messages passing through the channel. This is a security function that encodes the messages during transit so that you cannot read them without the decoding information. Either null or a *cryptor* that can perform encryption and decryption.

The simplest type of cryptor is MQeXorCryptor, which encrypts the data being sent by performing an exclusive-OR of the data. This encryption is not secure, but it modifies the data so that it cannot be viewed. In contrast, MQe3DESCryptor implements triple DES, a symmetric-key encryption method.

Channel
The class providing the transport services.

Compressor
This attribute causes compression and decompression to be performed on messages passing through the channel. This attempts to reduce the size of messages while they are being transmitted and stored. Either null or a *compressor* that can perform data compression and decompression. The simplest type of compressor is the MQeRleCompressor, which compresses the data by replacing repeated characters with a count.

Destination
The server and port number for the connection. The target for this connection, for example SERVER.XYZ.COM

Typically, authentication only occurs when setting up the connection. All flows normally use compressors and cryptors.

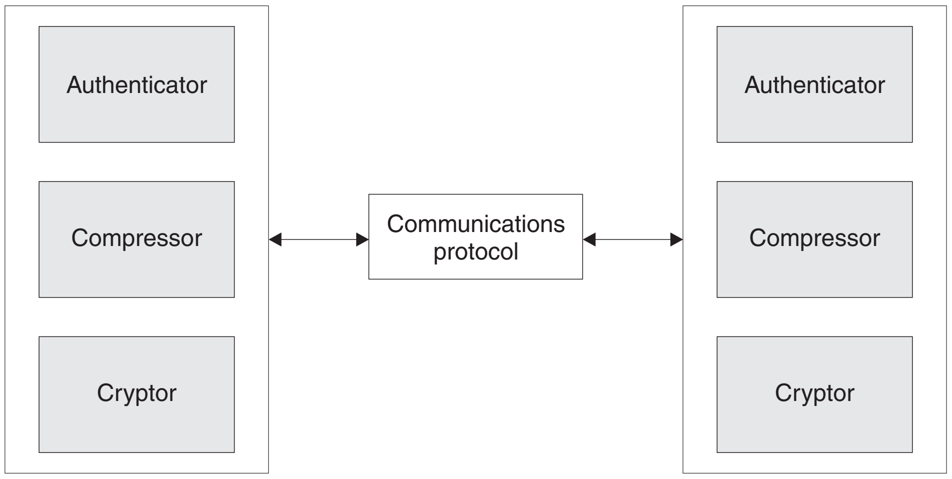


Figure 2. MQe connection

You can establish MQe connections using a variety of protocols allowing them to connect in a number of different ways, for example:

- Permanent connection, for example a LAN, or leased line
- Dial out connection, for example using a standard modem to connect to an Internet service provider (ISP)
- Dial out and answer connection, using a CellPhone, or ScreenPhone for example

MQe implements the communications protocols as a set of adapters, with one adapter for each of the supported protocols. This enables you to add new protocols.

Queue manager operations

Explanation of the messaging operations that you can perform on a queue manager

This topic explains in detail the messaging operations that you can perform on a queue manager. It describes the services, functions, and uses of queue managers under the following headings:

What is an MQe queue manager

Introduction to the function and use of queue managers

The MQe queue manager is the focal point of the MQe system. It provides:

- A central point of access to a messaging and queueing network for MQe applications
- Optional client-side queuing
- Optional administration functions
- Once and once-only assured delivery of messages
- Recovery from failure conditions
- Extendable rules-based behavior

Unlike base MQ, MQe has a single queue manager type. However, you can program MQe queue managers to act as traditional clients or servers. You can also customize queue manager behavior using rules. The MQe queue manager is embedded within user written programs and these programs can run on any MQe supported device or platform.

You can configure queue managers in a number of different ways, the main types being client, server, and gateway. You can also update the queue store of a queue manager using administration messages. For more information on administration messages, refer to the MQe Configuration Guide.

Communication with other queue managers on the MQe messaging network can be synchronous or asynchronous. If you want to use synchronous communications, the originator, and the target MQe queue managers must both be available on the network. Asynchronous communication allows an MQe application to send messages even when the remote queue manager is offline.

The queue manager life cycle

Overview of the life cycle of a queue manager

Typically, an application creates a new queue manager, configures it with a number of queues, and then frees the queue manager. An application also opens an existing queue manager, starts it, carries out messaging operations, and then stops. A further administration program can reopen the queue manager, remove all of its queues, and then stop. The following diagram displays this information:

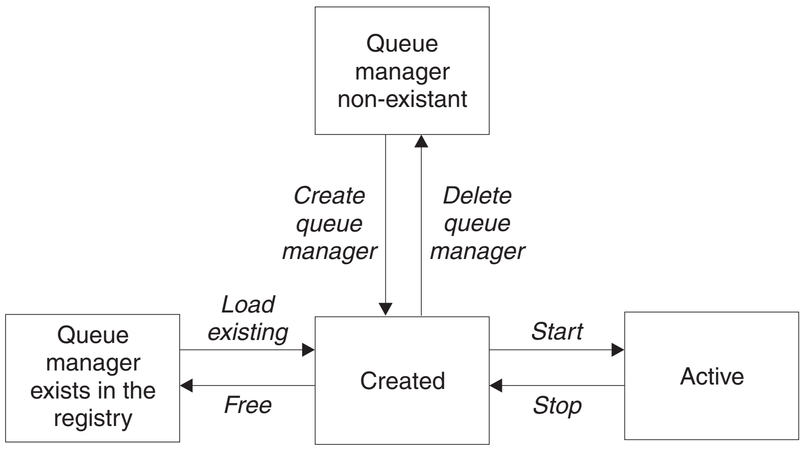


Figure 3. The queue manager life cycle

Creating queue managers

A queue manager requires at least the following:

- A registry
- A queue manager definition
- Local default queue definitions

Once these definitions are in place you can run the queue manager and use the administration interface to perform further configuration, such as adding more queues.

Methods to create these initial objects are supplied in the MQeQueueManagerConfigure class.

The example install programs `examples.install.SimpleCreateQM` and `examples.install.SimpleDeleteQM` use this class.

Queue manager names

MQe queue manager names can contain the following characters:

- Numerics 0 to 9
- Lower case a to z

- Upper case A to Z
- Underscore _
- Period .
- Percent %

There are no inherent name length limitations in MQe.

Creating a queue manager - step by step

The basic steps required to create a queue manager are:

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

Create and activate an instance of MQeQueueManagerConfigure:

You create the MQeQueueManagerConfigure object by calling the mqeQueueManagerConfigure_new method. Apart from the `ExceptionBlock` and the new MQeQueueManagerConfigure `Handle`, this method takes two additional parameters.

The method of operation depends on these parameters. ″NULL″ can be passed for these parameters, in which case mqeQueueManagerConfigure_activate is called immediately after mqeQueueManagerConfigure_new. Alternatively startup parameters can be passed.

You can activate the MQeQueueManagerConfigure class in either of the following ways:

1. Call the empty constructor followed by activate():

   ```
   try
   {
      MQeQueueManagerConfigure qmConfig;
      MQeFields parms = new MQeFields();
      // initialize the parameters

      qmConfig = new MQeQueueManagerConfigure( );
      qmConfig.activate( parms, "MsgLog:qmName\\Queues\\" );
   }
   catch (Exception e)
   { ... }
   ```

2. Call the constructor with parameters:

   ```
   try
   {
      MQeQueueManagerConfigure qmConfig;
      MQeFields parms = new MQeFields();
      // initialize the parameters

      qmConfig = new MQeQueueManagerConfigure( parms, "MsgLog:qmName\\Queues\\" );
   }
   catch (Exception e)
   { ... }
   ```

The first parameter is an MQeFields object that contains initialization parameters for the queue manager. These must contain at least the following:

- An embedded MQeFields object (`Name`) that contains the name of the queue manager.

- An embedded MQeFields object, that contains the location of the local queue store as the registry type (*LocalRegType*) and the registry directory name (*DirName*). If a base file registry is used these are the only parameters that are required. If a private registry is used, a *PIN* and *KeyRingPassword* are also required.

The directory name is stored as part of the queue manager definition and is used as a default value for the queue store in any future queue definitions. The directory does not have to exist and will be created when needed.

If you use an alias for any of the initialization parameters, or if you wish to use an alias to set the connection attribute rule name, the aliases should be defined before activating MQeQueueManagerConfigure .

```
import com.ibm.mqe.*;
import com.ibm.mqe.registry.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
   MQeQueueManagerConfigure qmConfig;
   MQeFields parms = new MQeFields();
   // initialize the parameters
   MQeFields qmgrFields  = new MQeFields();
   MQeFields regFields   = new MQeFields();

   // Queue manager name is needed
   qmgrFields.putAscii(MQeQueueManager.Name, "qmName");
   // Registry information
   regFields.putAscii(MQeRegistry.LocalRegType,
                     "com.ibm.mqe.registry.MQeFileSession");
   regFields.putAscii(MQeRegistry.DirName, "qmname\\Registry");

   // add the embedded MQeFields objects
   parms.putFields(MQeQueueManager.QueueManager, qmgrFields);
   parms.putFields(MQeQueueManager.Registry, regFields);
   // activate the configure object
   qmConfig = new MQeQueueManagerConfigure( parms, "MsgLog:qmName\\Queues\\" );
}
catch (Exception e)
{ ... }
```

The example code includes creating an instance of MQeQueueManagerConfigure.

Set queue manager properties:

When you have activated MQeQueueManagerConfigure, but before you create the queue manager definition, you can set some or all of the following queue manager properties:

- You can add a description to the queue manager with mqeQueueManagerConfigure_setDescription()
- You can set a connection time-out value with mqeQueueManagerConfigure_setChannelTimeout()
- You can set the name of the connection attribute rule with mqeQueueManagerConfigure_setChnlAttributeRuleName()

Call mqeQueueManagerConfigure_defineQueueManager( ) to create the queue manager definition. This creates a registry definition for the queue manager that includes any of the properties that you set previously.

```
import com.ibm.mqe.*;
import com.ibm.mqe.registry.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
   MQeQueueManagerConfigure qmConfig;
   MQeFields parms = new MQeFields();
```

```
      // initialize the parameters
      ...
      // activate the configure object
      qmConfig = new MQeQueueManagerConfigure( parms, "MsgLog:qmName\\Queues\\" );
      qmConfig.setDescription("a test queue manager");
      qmConfig.setChnlAttributeRuleName("ChannelAttrRules");
      qmConfig.defineQueueManager();
   }
   catch (Exception e)
   { ... }
```

At this point you can call `close()` and `free()` MQeQueueManagerConfigure and run the queue manager, however, it cannot do much because it has no queues. You cannot add queues using the administration interface, because the queue manager does not have an administration queue to service the administration messages.

The following sections show how to create queues and make the queue manager useful.

Create definitions for the default queues:

MQeQueueManagerConfigure allows you to define the following four standard queues for the queue manager:

defineDefaultAdminQueue()mqeQueueManagerConfigure_
: This administration queue is needed to allow the queue manager to respond to administration messages, for example to create new connection definitions and queues.

defineDefaultAdminReplyQueue()mqeQueueManagerConfigure_
: This administration reply queue is a local queue, used by connections as the destination of reply messages generated by administration.

defineDefaultDeadLetterQueue()mqeQueueManagerConfigure_
: This dead letter queue can be used, depending on the rules in force, to store messages that cannot be delivered to their correct destination.

defineDefaultSystemQueue()mqeQueueManagerConfigure_
: This default local queue, SYSTEM.DEFAULT.LOCAL.QUEUE, has no special significance within MQe itself, but it is useful when MQe is used with MQ messaging because it exists on every MQ messaging queue manager.

All methods throw an exceptionreturn an error if the queue already exists.

```
   import com.ibm.mqe.*;
   import com.ibm.mqe.registry.*;
   import examples.queuemanager.MQeQueueManagerUtils;
   try
   {
      MQeQueueManagerConfigure qmConfig;
      MQeFields parms = new MQeFields();
      // initialize the parameters
      ...
      qmConfig = new MQeQueueManagerConfigure( parms, "MsgLog:qmName\\Queues\\" );
      qmConfig.setDescription("a test queue manager");
      qmconfig.defineDefaultAdminQueue();
      qmconfig.defineDefaultAdminReplyQueue();
      qmconfig.defineDefaultDeadLetterQueue();
      qmconfig.defineDefaultSystemQueue();
   }
   catch (Exception e)
   { ... }
```

Close the MQeQueueManagerConfigure instance:

When you have defined the queue manager and the required queues, you can close() MQeQueueManagerConfigure and run the queue manager.

The complete example looks like this:

```
import com.ibm.mqe.*;
import com.ibm.mqe.registry.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
   MQeQueueManagerConfigure qmConfig;
   MQeFields parms = new MQeFields();
   // initialize the parameters
   MQeFields qmgrFields  = new MQeFields();
   MQeFields regFields   = new MQeFields();
   // Queue manager name is needed
   qmgrFields.putAscii(MQeQueueManager.Name, "qmName");
   // Registry information
   regFields.putAscii(MQeRegistry.LocalRegType,
                      "com.ibm.mqe.registry.MQeFileSession");
   regFields.putAscii(MQeRegistry.DirName, "qmname\\Registry");
   // add the embedded MQeFields objects
   parms.putFields(MQeQueueManager.QueueManager, qmgrFields);
   parms.putFields(MQeQueueManager.Registry, regFields);
   // activate the configure object
   qmConfig = new MQeQueueManagerConfigure( parms, "MsgLog:qmName\\Queues\\" );
   qmConfig.setDescription("a test queue manager");
   qmConfig.setChnlAttributeRuleName("ChannelAttrRules");
   qmConfig.defineQueueManager();
   qmconfig.defineDefaultAdminQueue();
   qmconfig.defineDefaultAdminReplyQueue();
   qmconfig.defineDefaultDeadLetterQueue();
   qmconfig.defineDefaultSystemQueue();
   qmconfig.close();
}
catch (Exception e)
{ ... }
```

The registry definitions for the queue manager and the required queues are created immediately. The queues are not created until they are activated.

Persistent configuration data

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

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

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

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

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

- Queue manager type.

Creating simple queue managers

The simplest MQe queue manager is a queue manager that uses a registry based upon the internal default values. The queue manager could be created without any queues, but its functionality would be severely limited. The example we create contains four standard queues:
- Admin queue - so that administration can be performed
- Admin reply queue - a standard place to store replies from administration actions
- System default queue - a useful general purpose local queue
- Dead letter queue - a place for undeliverable messages

The simplest queue manager has no security and has a registry stored in the local file system. The steps to achieve are:
- Create a registry on disk
- Create and start a queue manager using the registry
- Stop the queue manager

These actions are described for both the Java code base and the C code base, with example code for each. The example Java code is shipped as examples.config.CreateQueueManager. For C example code, refer to the HelloWorld compilation section and the transport-c file in the Broker example.

Creating a simple queue manager in Java:

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

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

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

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

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

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

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

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

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

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

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

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

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

Finally the MQeQueueManagerConfigure object can be closed.

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

Creating a simple queue manager in C:

Stage 1: Create the admin components

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

Stage 2: Create a QueueManager

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

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

```
if ( MQERETURN_OK == rc ) {

   MQeQueueManagerParms qmParams  = QMGR_INIT_VAL;
   MQeRegistryParms     regParams = REGISTRY_INIT_VAL;

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

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

}
```

Figure 4. Create queue manager C example

Starting queue managers

Queue managers need to be created before use. The creation step uses the QueueManagerConfigure Java class or the C administration API to create persistent queue manager data in a registry. The queue manager then uses the registry each time its starts.

Starting queue managers in Java

Normally, creating and starting a queue manager can require a large set of parameters. Therefore, the required parameters are supplied as an instance of MQeFields, storing the values as fields of correct type and name.

The parameters fall into two categories, queue manager parameters and registry parameters. Each of these categories is represented by its own MQeFields instance, and both are also enclosed in an MQeFields instance. The following Java example explains this concept, passing the queue managers name, ″ExampleQM″ and the location of a registry, ″C:\ExampleQM″:

```
/*create fields for queue manager parameters and place the queue manager name
MQeFields queueManagerParameters = new MQeFields();
queueManagerParameters.putAscii(MQeQueueManager.Name, "ExampleQM");

/*create fields for registry parameters and place the registry location
MQeFields registryParameters = new MQeFields();
registryParameters.putAscii(MQeRegistry.DirName, "C:\\ExampleQM\\registry");

/*create fields for combined parameters and place the two sub fields
MQeFields parameters = new MQeFields();
parameters.putFields(MQeQueueManager.Registry, queueManagerParameters);
parameters.putFields(MQeQueueManager.Registry, registryParameters);
```

Wherever you see ″`initialize the parameters`″ in code snippets, prepare a set of parameters as shown in the example, including the appropriate options. Only one queue manager name and one registry location are mandatory.

Starting a simple queue manager in Java:

To start the simplest queue manager, you only need to provide the queue manager name and registry location to the queue manager constructor. This starts and activates the queue manager, and when the constructor returns the queue manager is running.

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

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

Starting queue managers in C

The `mqeQueueManager_new` function loads a queue manager for an established registry. To do this, you need information supplied by a queue manager parameter structure and a registry parameter structure.

The following example shows how you can set these structures to their default values, supplying only the directories of the queue store and registry:

```
MQeQueueManagerHndl hQueueManager;
MQeRegistryParms regParms = REGISTRY_INIT_VAL;
MQeQueueManagerParms qmParms = QMGR_INIT_VAL;
regParms.hBaseLocationName = hRegistryDirectory;
qmParms.hQueueStore = hStore;
qmParms.opFlags = QMGR_Q_STORE_OP;
rc = mqeQueueManager_new(&exceptBlock,
                         &hQueueManager, hQMName,
                         &regParams, &qmParms);
```

This creates a queue manager and loads its persistent information from the registry and creates queues. However, you must start the queue manager to:

- Create messages
- Get and put messages
- Process administration messages, using the administration queue

Note: In C, the queues are activated on starting the queue manager.

To start the queue manager, use

```
rc = mqeQueueManager_start(&hQueueManager, &exceptBlock);
```

Once the queue manager is started, messaging operations can take place and any queues that have messages on them are loaded.

To stop the queue manager, use:

```
rc = mqeQueueManager_stop(&hQueueManager, &exceptBlock);
```

Once stopped, you can restart the queue manager as required.

At the end of the application, you must free the queue manager to release any resources it uses, for example memory. First, stop the queue manager and then use:

```
rc = mqeQueueManager_free(&hQueueManager, &exceptBlock);
```

Starting a simple queue manager in C:

This process involves two steps:

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

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

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

```
     if (MQERETURN_OK == rc) {

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

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

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

```
     /* Start the queue manager  */

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

Queue manager parameters

List of the parameter names that can be passed to the queue manager and the registry.

The following lists the parameter names that you can pass to the queue manager and the registry:

Queue manager Parameters

MQeQueueManager.Name(ascii)
This is the name of the queue manager being started.

Registry Parameters

MQeRegistry.LocalRegType(ascii)
This is the type of registry being opened. MQe currently supports:

file registry
Set this parameter to `com.ibm.mqe.registry.MQeFileSession`.

private registry
Set this parameter to `com.ibm.mqe.registry.MQePrivateSession`.

You also need a private registry for some security features.

MQeRegistry.DirName(ascii)
This is the name of the directory holding the registry files. You must pass this parameter for a file registry.

MQeRegistry.PIN(ascii)
You need this PIN for a private registry.

Note: For security reasons, MQe deletes the PIN and KeyRingPassword, if supplied, from the startup parameters as soon as the queue manager is activated.

MQeRegistry.CAIPAddrPort(ascii)
You need this address and port number of a mini-certificate server for auto-registration, so that the queue manager can obtain its credentials from the mini-certificate server.

MQeRegistry.CertReqPIN(ascii)
This is the certificate request number allocated by the mini-certificate administrator to allow the registry to obtain its credentials. You need this for auto-registration, so that the queue manager can obtain its credentials from the mini-certificate server.

MQeRegistry.Separator(ascii)
This is used to specify a non-default separator. A separator is the character used between the the components of an entry name, for example `<QueueManager><Separator><Queue>`. Although this parameter is specified as a string, it must contain a single character. If it contains more than one, only the first character is used. Use the same separator for each registry opened and do not change it once a registry is in use. If you do not specify this parameter, the separator defaults to "+".

MQeRegistry.RegistryAdapter(ascii)
This is the class, or an alias that resolves to a class, of the adapter that the registry uses to store its data. You must include this class if you want the registry to use an adapter other than the default MQeDiskFieldsAdapter. You can use any valid storage adapter class.

You always need the first two parameters. The last two are for auto-registration of the registry if it wishes to obtain credentials from the mini-certificate server.

`MQeRegistry.RegistryAdapter (ascii)`
The class, (or an alias that resolves to a class), of the adapter that the registry uses to store its data. This value should be included if you want the registry to use an adapter other than the default MQeDiskFieldsAdapter. Any valid adapter class can be used.

A queue manager can run:
- As a client
- As server
- In a servlet

The following sections describe the example client, servers and servlet that are provided in the examples.queuemanager package.refer extensively to the example code to illustrate how to start queue managers. All queue managers are constructed from the same base MQe components, with some additions that give each its unique properties. MQe provides an example class, `MQeQueueManagerUtils`, that encapsulates many of the common functions.

All the examples require parameters at startup. These parameters are stored in standard ini files. The ini files are read and the data is converted into an MQeFields object. The `loadConfigFile()` method in the MQeQueueManagerUtils class performs this function.

Registry parameters for a queue manager

Description of the queue manager-related data held in the registry

The registry is the primary store for queue manager-related information; one exists for each queue manager. Every queue manager uses the registry to hold its:

- Queue manager configuration data
- Communications listener resource definitions
- Queue definitions
- Remote queue definitions
- Remote queue manager definitions
- User data, including configuration-dependent security information
- Optional bridge resource definitions

Registry type

`MQE_REGISTRY_LOCAL_REG_TYPE`
: The type of registry being opened. *file registry* and *private registry* are currently supported. A private registry is required for some of the security features.

For a file registry this parameter should be set to:

```
com.ibm.mqe.registry.MQeFileSession
```

For a private registry it should be set to:

```
com.ibm.mqe.registry.MQePrivateSession
```

Aliases can be used to represent these values.

Client queue managers

A client typically runs on a device platform, and provides a queue manager that can be used by applications on the device. It can open many connections to other queue managers.

A server usually runs for long periods of time, but clients are started and stopped on demand by the application that use them. If multiple applications want to share a client , the applications must coordinate the starting and stopping of the client.

Example - starting a client queue manager:

Starting a client queue manager involves:

1. Ensuring that there is no client already running. (Only one client is allowed per Java Virtual Machine.)
2. Adding any aliases to the system
3. Enabling trace if required
4. Starting the queue manager

The following code fragment starts a client queue manager:

```
MQERETURN createQueueManager(MQeExceptBlock *pErrorBlock, MQeQueueManagerHndl *phQMgr,
                             MQeFieldsHndl hInitFields, MQeStringHndl hQStore)
{

   MQERETURN rc;
   MQeQueueManagerConfigureHndl hQMgrConfigure;

      /* Create instance of QueueManagerConfigure Class */
      rc = mqeQueueManagerConfigure_new(pErrorBlock,&hQMgrConfigure,
                            hInitFields,hQStore);

      if (MQERETURN_OK == rc) {
         /* define queue manager */
         rc = mqeQueueManagerConfigure_defineQueueManager(hQMgrConfigure, pErrorBlock);
         if (MQERETURN_OK == rc) {
            /* define system default queues */
            rc = mqeQueueManagerConfigure_defineDefaultSystemQueue(hQMgrConfigure,
                                                           pErrorBlock, NULL);
         }

         /* close mqeQueueManagerConfigure */
         (void)mqeQueueManagerConfigure_close(hQMgrConfigure, NULL);
         if (MQERETURN_OK == rc) {
            /* create queue manager */
            rc = mqeQueueManager_new(pErrorBlock, phQMgr);
            if (MQERETURN_OK == rc) {
               rc = mqeQueueManager_activate(*phQMgr, pErrorBlock, hInitFields);
            }
         }
         /* free mqeQueueManagerConfigure */
         (void)mqeQueueManagerConfigure_free(hQMgrConfigure, NULL);
      }

   return rc;
}
/*---------------------------------------*/
/* Init - first stage setup              */
/*---------------------------------------*/
public void init( MQeFields parms ) throws Exception
{
  if ( queueManager != null )
/* One queue manager at a time   */
  {
    throw new Exception( "Client already running" );
  }
  sections = parms;
/* Remember startup parms        */
  MQeQueueManagerUtils.processAlias( sections );
/* set any alias names       */

// Uncomment the following line to start trace
    before the queue manager is started
//  MQeQueueManagerUtils.traceOn("MQeClient Trace", null);
/* Turn trace on   */

  /* Display the startup parameters */
  System.out.println( sections.dumpToString("#1\t=\t#2\r\n"));

  /* Start the queue manage  */
  queueManager = MQeQueueManagerUtils.processQueueManager( sections, null);
}
```

Once you have started the client, you can obtain a reference to the queue manager object by using API call mqeQueueManager_getReference(queueManagerName)either from the static class variable `MQeClient.queueManager` or by using the static method `MQeQueueManager.getReference(queueManagerName)`.

The following code fragment loads aliases into the system:

```
public static void processAlias( MQeFields sections ) throws Exception
{
  if ( sections.contains( Section_Alias ) )
/* section present ?           */
  {
/* ... yes                     */
    MQeFields section = sections.getFields( Section_Alias );
    Enumeration keys  = section.fields( );
/* get all the keywords        */
    while ( keys.hasMoreElements() )
/* as long as there are keywords*/
    {
      String    key   = (String) keys.nextElement();
/* get the Keyword   */
      MQe.alias( key, section.getAscii( key ).trim( ) );
/* add               */
    }
  }
}
```

Use the processAlias method to add each alias to the system. MQe and applications can use the aliases once they have been loaded.

Starting a queue manager involves:

1. Instantiating a queue manager. The name of the queue manager class to load is specified in the alias QueueManager. Use the MQe class loader to load the class and call the null constructor.
2. Activate the queue manager. Use the activate method, passing the MQeFields object representation of the ini file. The queue manager only makes use of the [QueueManager] and [Registry] sections from the startup parameters.

The following code fragment starts a queue manager:

```
public static MQeQueueManager processQueueManager( MQeFields sections,
     Hashtable ght ) throws Exception
{
/*                             */
  MQeQueueManager queueManager = null;
/* work variable               */
  if ( sections.contains( Section_QueueManager) )
/* section present ?    */
  {
/* ... yes                     */
   queueManager = (MQeQueueManager) MQe.loader.loadObject(Section_QueueManager);
    if ( queueManager != null )
/* is there a Q manager ?      */
    {
      queueManager.setGlobalHashTable( ght );
      queueManager.activate( sections );
/* ... yes, activate           */
    }
  }
  return( queueManager );
/* return the alloated mgr     */
}
```

Example - MQePrivateClient:

MQePrivateClient is an extension of MQeClient with the addition that it configures the queue manager and registry to allow for secure queues. For a secure client, the [Registry] section of the startup parameters is extended as follows:

```
(ascii)LocalRegType=PrivateRegistry

   Location of the registry

(ascii)DirName=.\ExampleQM\PrivateRegistry
   Adapter on which registry sits
(ascii)Adapter=RegistryAdapter
Network address of certificate authority

(ascii)CAIPAddrPort=9.20.7.219:8082
```

For MQePrivateClient and MQePrivateServer to work, the startup parameters must *not* contain `CertReqPIN`, *`KeyRingPassword`* and `CAIPAddrPort`.

Server queue managers

A server usually runs on a server platform. A server can run server-side applications but can also run client-side applications. As with clients, a server can open connections to many other queue managers on both servers and clients. One of the main characteristics that differentiate a server from a client is that it can handle many concurrent incoming requests. A server often acts as an entry point for many clients into an MQe network . MQe provides the following server examples:

MQeServer
: A console based server.

MQePrivateServer
: A console based server with enhanced security.

AwtMQeServer
: A graphical front end to MQeServer.

MQBridgeServer
: In addition to the normal MQe server functions, this server can send and receive messages to and from other members of the MQ family. This server is in package examples.mqbridge.queuemanager.

Example - MQeServer:

MQeServer is the simplest server implementation.

```
qm_server server_QMgr_name [-p private_reg_PIN]
```

You must supply the *`-p`* parameter if the queue manager uses a private registry. Otherwise, the queue manager's registry is treated as a file registry. The program activates the queue manager (including a listener listening on port 8081) and goes into an indefinite sleep.

Use `ctrl-C` to shut down the server.

To delete the constructed queue manager, use the example qm_delete.

When two queue managers communicate with each other, MQe opens a connection between the two queue managers. The connection is a logical entity that is used as a queue manager to queue manager pipe. Multiple connections may be open at any time.

Server queue managers, unlike client queue managers, can have one or more listeners. A listener waits for communications from other queue managers, and processes incoming requests, usually by forwarding them to its owning queue manager. Each listener has a specified adapter that defines the protocol of incoming communications, and also specifies any extra data required.

You create listeners on the local queue manager using administration messages, remotely and locally. However, a remote queue manager must have a listener in order to receive a message.

A listener that has just been created by sending administration messages to the queue manager does not then start. To start it you can send an administration message explicitly to start the listener, or you can restart the queue manager. (However, listeners are persistent in the registry. This means that, once created, listeners that exist at queue manager startup are started automatically).

This example shows how to create and start a listener using administration messages:

```
String  listenerName = "MyListener";
   String  listenAdapter = "com.ibm.mqe.adapters.MQeTcpipHttpAdapter";
   int     listenPort = 1881;
   int     channelTimeout = 300000;
   int     maxChannels = 0;

   MQeCommunicationsListenerAdminMsg msg = new MQeCommunicationsListenerAdminMsg();

     msg.setName(listenerName);
     msg.create(listenAdapter, listenPort, channelTimeout, maxChannels);

     .
     .
     .

     //In order to start the listener use the start action

   MQeCommunicationsListenerAdminMsg msg = new MQeCommunicationsListenerAdminMsg();

   msg.setName(listenerName);
   msg.start();

   .
   .
```

When the listener is started, the server is ready to accept network requests.

When the server is deactivated:
1. The listener is stopped, preventing any new incoming requests
2. The queue manager is closed

Example - MQePrivateServer:

MQePrivateServer is an extension of MQeServer with the addition that it configures the queue manager and registry to allow for secure queues.

Environment relationship

This topic describes some requirements for running Java and C implementations of MQe.

Java code

The java queue manager runs inside an instance of a JVM. You can have only one queue manager per JVM. However, you can invoke multiple instances of the JVM.

Each of these queue managers must have a unique name. Java applications run inside the same JVM as the queue manager they use.

C code

You can run only one queue manager within a native C process. You need multiple processes for multiple queue managers. Each of these queue managers must have a unique name.

Stopping queue managers

Overview of stopping queue managers in Java and C

Stopping a queue manager in Java

There are 2 ways to close down a QueueManager, and one of the close methods should be called by MQe applications when they have finished using the queue manager:

- closeQuiese
- closeImmediate

closeQuiesce:

Stopping a queue manager using the closeQuiesce method

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

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

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

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

closeImmediate:

Stopping a queue manager using the closeImmediate method

This closes Queue Manager immediately.

After this method has been called, no more event notifications are dispatched to message listeners. Messages might complete their arrival after this method has been called, and before it finishes. Such messages are not notified, and therefore message arrival does not generate a message event.

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

Stopping a queue manager in C

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

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

    /* Lets do some clean up */
    (void)mqeString_free(hFieldLabel,&exceptBlock);
    (void)mqeString_free(hLocalQMName,&exceptBlock);
    (void)mqeString_free(hLocalQueueName,&exceptBlock);
    (void)mqeString_free(hQueueStore,&exceptBlock);
    (void)mqeString_free(hRegistryDir,&exceptBlock);


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

Deleting queue managers

This section details how to delete a queue manager in Java and C.

Java

Steps required to delete queue managers in Java

The basic steps required to delete a queue manager are:

1. Use the administration interface to delete any definitions
2. Create and activate an instance of `MQeQueueManagerConfigure`
3. Delete the standard queue and queue manager definitions
4. Close the `MQeQueueManagerConfigure` instance

When these steps are complete, the queue manager is deleted and can no longer be run. The queue definitions are deleted, but the queues themselves are not deleted. Any messages remaining on the queues are inaccessible.

Note: If there are messages on the queues they are not automatically deleted. Your application programs should include code to check for, and handle, remaining messages before deleting the queue manager.

1. Delete any definitions

You can use MQeQueueManagerConfigure to delete the standard queues that you created with it. Use the administration interface to delete any other queues before you call MQeQueueManagerConfigure.

2. Create and activate an instance of MQeQueueManagerConfigure

This process is the same as when creating a queue manager. See "Creating queue managers" on page 18.

3. Delete the standard queue and queue manager definitions

Delete the default queues by calling:

- mqeQueueManagerConfigure_deleteAdminQueueDefinition() to delete the administration queue
- mqeQueueManagerConfigure_deleteAdminReplyQueueDefinition() to delete the administration reply queue
- mqeQueueManagerConfigure_deleteDeadLetterQueueDefinition() to delete the dead letter queue
- mqeQueueManagerConfigure_deleteSystemQueueDefinition() to delete the default local queue

These methods work successfully even if the queues do not exist.

Delete the queue manager definition by calling mqeQueueManagerConfigure_deleteQueueManagerDefinition()

```
import com.ibm.mqe.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialize the parameters
 ...
 // Establish any aliases defined by the .ini file
 MQeQueueManagerUtils.processAlias(parms);
qmConfig = new MQeQueueManagerConfigure( parms );
 qmConfig.deleteAdminQueueDefinition();
 qmConfig.deleteAdminReplyQueueDefinition();
 qmConfig.deleteDeadLetterQueueDefinition();
 qmConfig.deleteSystemQueueDefinition();
 qmConfig.deleteQueueManagerDefinition();
 qmconfig.close();
}
catch (Exception e)
{ ... }
```

You can delete the default queue and queue manager definitions together by calling mqeQueueManagerConfigure_deleteStandardQMDefinitions(). This method is provided for convenience and is equivalent to:

```
deleteDeadLetterQueueDefinition();
deleteSystemQueueDefinition();
deleteAdminQueueDefinition();
deleteAdminReplyQueueDefinition();
deleteQueueManagerDefinition();
```

4. Close the MQeQueueManagerConfigure instance

When you have deleted the queue and queue manager definitions, you can close the MQeQueueManagerConfigure instance.

The complete example looks like this:

```
import com.ibm.mqe.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialize the parameters
 ...
 // Establish any aliases defined by the .ini file
 MQeQueueManagerUtils.processAlias(parms);
 qmConfig = new MQeQueueManagerConfigure( parms );
 qmConfig.deleteStandardQMDefinitions();
 qmconfig.close();
}
catch (Exception e)
{ ... }
```

C

Steps required to delete queue managers in C

The steps in deleting a queue manager are:

1. Remove all Connection Definitions.
2. Remove all Queues, including any ″system″ queues, for example the dead letter queue. Ensure all queues are empty.
3. Remove the queue manager.

You require an administrator to perform these functions. We also recommend stopping the queue manager first.

Note: Deleting the queue mananger will free the queue manager handle for you.

MQeAdministratorHndl hAdmin:

```
/* Create the new administrator based on the exisitng QM Handle */
rc = mqeAdministrator_new(&exceptBlock,
                  &hAdmin,hQueueManager);
if (MQERETURN_OK == rc) {

    if (MQERETURN_OK == rc) {
        /* delete any conncetion definitins for example :*/
        rc = mqeAdministrator_Connection_delete(hAdmin,
                                &exceptBlock,
                                hRemoteQM);
    }

    /* delete all the local queues here - remember to do "special*/
   /*queues" for example ... */
    if (MQERETURN_OK == rc) {
        rc = mqeAdministrator_LocalQueue_delete(hAdmin,
                                &exceptBlock,
                                MQE_DEADLETTER_QUEUE_NAME,
                                hLocalQMName);
    }

    /* Finally delete the queue manager */
    if (MQERETURN_OK == rc) {
        rc = mqeAdministrator_QueueManager_delete(hAdmin,
                                  &exceptBlock);
    }



    /* free of the amdinsitrator */
    (void)mqeAdministrator_free(hAdmin, &exceptBlock);
}
```

Messaging life cycle

Description of the series of states through which a message progresses when it is put to a queue

When a message is put to a queue it progresses through a series of states. This section describes these states and related commands or events under the following headings:

Message states

Most queue types hold messages in a persistent store, for example a hard disk. While in the store, the state of the message varies as it is transferred into and out of the store. As shown in Figure 5 on page 37:

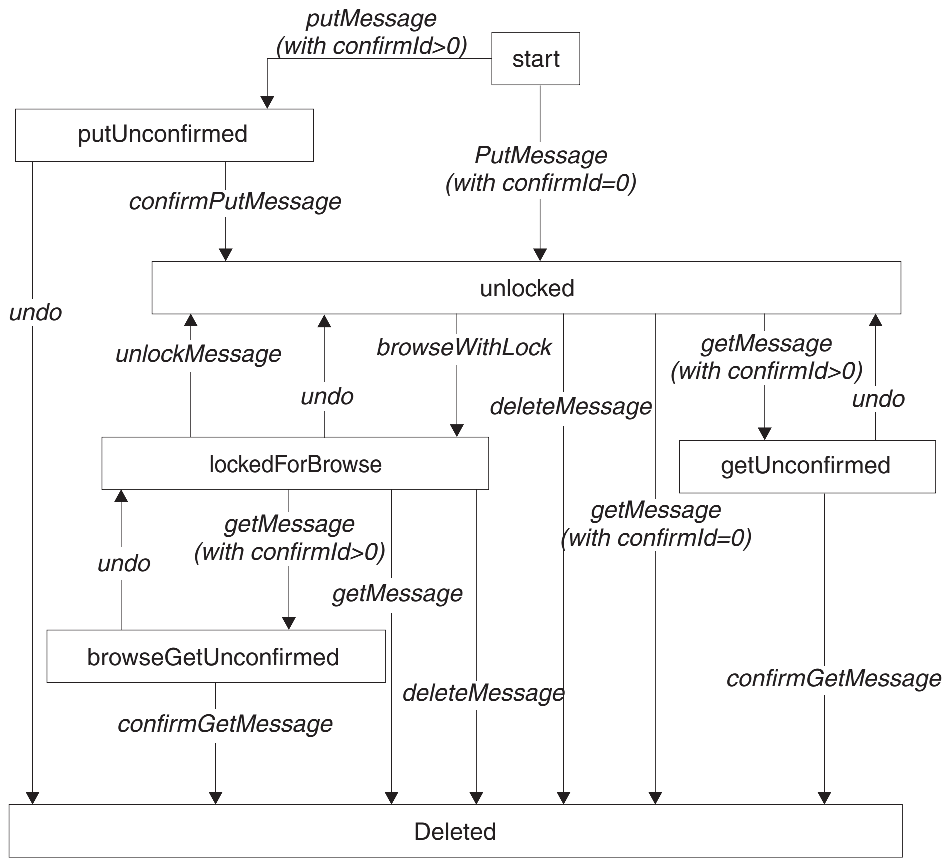


Figure 5. Stored message state flow

In this diagram, ″start″ and ″deleted″ are not actual message states. They are the entry and exit points of the state model. The message states are:

Put unConfirmed
: A message is put to the message store of a queue with a `confirmID`. The message is effectively hidden from all actions except `confirmPutMessage` or `undo`.

Unlocked
: A message has been put to a queue and is available to all operations.

Locked for Browse
: A browse with lock retrieves messages. Messages are hidden from all queries except `getMessage`, `unlock`, `delete`, `undo`, and `unlockMessage`. A `lockID` is returned from the browse operation. You must supply this `lockID` to all other operations.

Get Unconfirmed
: A `getMessage` call has been made with a `confirmID`, but the get has not been confirmed. The message is invisible to all queries except `confirmGetMessage`, `confirm`, and `undo`. Each of these actions requires the inclusion of the matching `confirmID` to confirm the get.

Browse Get Unconfirmed
: A message got while it is locked for browse. You can do this only by passing the correct `lockID` to the `getMessage` function.

On an asynchronous remote queue, other states exist where a message is being transmitted to another machine. These states are entered as ″unlocked″, that is only confirmed messages are transmitted.

Message events

Messages pass from one state to another as a result of an event. These events are typically generated by an API call. The possible message events, as shown in Figure 5 on page 37, are:

`putMessage`
: Places a message on a queue. This does not require a `confirmID`.

`getMessage`
: Retrieves a message from a queue. This does not require a `confirmID`.

`putMessage with confirmId>0`
: Places a message on a queue. This requires a `confirmID`. However, messages do not arrive at the receiving end in the order of sending, but in the order of confirmation.

`confirmPutMessage`
: A confirm for an earlier `putMessage` with a `confirmID`>0.

`getMessage with confirmId>0`
: Retrieves message from a queue. This requires a `confirmID`.

`confirmGetMessage`
: A confirm for an earlier `getMessage` with a `confirmID`>0.

`browseWithLock`
: Browses messages and lock those that match. Prevents messages from changing while browse is in operation.

unlockMessage
: Unlocks a message locked with a `browsewithLock` command.

undo Unlocks a message locked with a browse, undoes a `getMessage` with a `confirmID`>0, or undoes a putMessage with a `confirmID`>0.

deleteMessage
: Removes a message from a queue.

Message index fields

Due to memory size constraints, complete messages are not held in memory, but, to enable faster message searching, MQe holds specific fields from each message in a *message index*. The fields that are held in the index are:

Java In Java, the following fields are held in the index:

`UniqueID`
: `MQe.Msg_OriginQMgr + MQe.Msg_Time`

`MessageID`
: `MQe.Msg_ID`

`CorrelationID`
: `MQe.Msg_CorrelID`

`Priority`
: `MQe.Msg_Priority`

C In C, the following fields are held in the index:

`UniqueID`
: MQE_MSG_ORIGIN_QMGR + MQE_MSG_TIME

`MessageID`
: MQE_MSG_MSGID

`CorrelationID`
MQE_MSG_CORRELID

`Priority`
MQE_MSG_PRIORITY

Providing these fields in a filter makes searching more efficient, since MQe may not have to load all the available messages into memory.

Messaging operations

The following table shows which types of messaging operations are valid on local queues, synchronous remote queues, and asynchronous remote queues. Note that the `Listen` and `Wait` operations are supported in Java only.

Table 3. Messaging operations on MQe queues

Operation	Local queue	Synchronous remote queue	Asynchronous remote queue
"Put"	Yes	Yes	Yes
"Get" on page 40	Yes	Yes	No
"Browse" on page 40	Yes	Yes	No
"confirmPut" on page 41	Yes	Yes	Yes
"confirmGet" on page 41	Yes	Yes	No
"Delete" on page 40	Yes	Yes	No
"Listen" on page 41	Yes	No	No
"Wait" on page 41	Yes	Yes	No

Note:
1. The synchronous remote wait operation is implemented through a poll of the remote queue, so the actual wait time is a multiple of the poll time
2. The MQ bridge supplied with MQe only supports an assured or unassured put, unassured get, and unassured browse (without lock).

Put

This operation places specified messages on a specified queue. The queue can belong to a local or remote queue manager. Puts to remote queues can occur immediately, or at a later time, depending on how the remote queue is defined on the local queue manager.

If a remote queue is defined as synchronous, message transmission occurs immediately. If a remote queue is defined as asynchronous, the message is stored within the local queue manager. The message remains there until it is transmitted. The put message call may finish before the message is put. Refer to "Message delivery" on page 49 for more information.

Note: In Java, if the local queue manager does not hold a definition of the remote queue then it attempts to contact the queue synchronously. This does not apply to the C code base.

Assured delivery depends on the value of the `confirmID` parameter. Passing a non-zero value transmits the message as normal, but the message is locked on the target queue until a subsequent confirm is received. Passing a value of zero transmits the message without the need for a subsequent confirm. However, message delivery is not assured. Refer to "Message delivery" on page 49, for more information on assured and non-assured message delivery.

You can protect a message using message-level security.

Get

This operation returns an available message from a specified queue and removes the message from the queue. The queue can belong to a local or remote MQe queue manager, but cannot be an asynchronous remote queue.

If you do not specify a filter, the first available message is returned. If you do specify a filter, the first available message that matches the filter is returned. Including a valid `lockID` in the message filter allows you to get messages that have been locked by a previous browse operation. If no message is available, the get operation returns an error.

Using assured message delivery depends on the value of the `confirmID` parameter. Passing a non-zero value returns the message as normal. However, the message is locked and is not removed from the target queue until it receives a subsequent confirm. You can issue a confirm using the `confirmGetMessage()` method. However, message delivery is not assured. Refer to "Message delivery" on page 49, for more information on assured and non-assured message delivery.

Browse

You can browse queues for messages using a filter, for example `message ID` or `priority` . Browsing retrieves all the messages that match the filter, but leaves them on the queue. The queue can belong to a local or remote queue manager.

MQe also supports *Browsing under lock*. This allows you to lock the matching messages on the queue. You can lock messages individually, or in groups identified through a filter, and the locking operation returns a `lockID`. Use the `lockID` to get or delete messages. An option on browse allows you to return either the full messages, or only the UniqueIDs.

```
  MQeVectorHndl hListMsgs;

  rc = mqeQueueManager_browseMessages(hQueueManager,
                                      &exceptBlock,
                                      &hListMsgs,
                                      hQMName,
                                      hQueueName,
                                      hFilter,
                                     NULL,MQE_FALSE);
if (MQERETURN_OK == rc) {
    /* process list using mqeVector_* apis */

    /* free off the vector */
    rc = mqeVector_free(hListMsgs,&exceptBlock);
}
```

Returning an entire collection of messsages can be expensive in terms of system resources. Setting the `justUID` parameter to true and returns the `uniqueID` of each message that matches the filter only.

The messages returned in the collection are still visible to other MQe APIs. Therefore, when performing subsequent operations on the messages contained in the enumeration, the application must be aware that another application can process these messages once the collection is returned. To prevent other applications from processing messages, use the `browseMessagesAndLock` method to lock messages contained in the enumeration.

Delete

This method deletes a message from a queue. It does not return the message to the application that called it. You must specify the UniqueID and you can delete only one message per operation.

The queue can belong to a local or synchronous remote MQe queue manager. Including a valid `lockID` in the message filter allows you to delete messages that have been locked by a previous operation, for example browse. If a message is not available, the application returns an error.

```
/* Example for deleting a message */
MQeFieldsHndl hMsg,hFilter;

/* create the new message */
rc = mqeFields_new(&exceptBlock, &hMsg);
if (MQERETURN_OK == rc) {

    /* add application fields here */
    /* ... */


    /* put message to a queue */
    rc = mqeQueueManager_putMessage(hQueueManager,
                        &exceptBlock,
                        hQMName,
                        hQueueName, hMsg,
                        NULL,0);
    if (MQERETURN_OK == rc) {
        /* Delete requires a filter -
        this can most easily be*/
      /*  found from the UID fields of the message*/
        rc = mqeFieldsHelper_getMsgUidFields(hMsg,
                              &exceptBlock,
                              &hFilter);
    }

}


/* some time later want to delete the message  -
    use the esatblished filter */
rc = mqeQueueManager_deleteMessage(hQueueManager,
                                 &exceptBlock,
                                  hQMName,
                                  hQueueName,
                                  hFilter);
```

confirmPut

This method performs the confirmation of a previously successful `putMessage()` operation.

confirmGet

This method confirms the successful receipt of a message retrieved from a queue manager by a previous `getMessage()` operation. The message remains locked on the target queue until it receives a confirm flow.

Listen

Applications can listen for MQe message events, again with an optional filter. However, in order to do this, you must add a listener to a queue manager. Listeners are notified when messages arrive on a queue.

Wait

This method implements message polling. It allows you to specify a time for messages to arrive on a queue. Java implements a helper function for this. The C code base, as it is non-threaded, must implement a function in application layer code. The following example demonstrates the `Wait` method:

Java Message polling uses the `waitForMessage()` method. This command issues a `getMessage()`

command to the remote queue at regular intervals. As soon as a message that matches the supplied filter becomes available, it is returned to the calling application:

```
qmgr.waitForMessage("RemoteQMgr",
              "RemoteQueue",
              filter,
              null,
              0,
              60000);
```

The `waitForMessage()` method polls the remote queue for the length of time specified in its final parameter. The time is specified in milliseconds. Therefore, in the example, polling lasts for 6 seconds. This blocks the thread on which the command is running for 6 seconds, unless a message is returned earlier. Message polling works on both local and remote queues.

Note: Using this technique sends multiple requests over the network.

Queue ordering

Overview of the ordering of messages on a queue

The order of messages on a queue is primarily determined by their priority. Message priority ranges from 9 (highest) to 0 (lowest). Messages with the same priority value are ordered by the time at which they arrive on the queue, with messages that have been on the queue for the longest being at the head of the priority group.

Reading messages on a queue

If you issue a `getMessage` command when a queue is empty, the queue throws a Java code base `Except_Q_NoMatchingMsg` exception or returns a C code base `MQERETURN_QUEUE_ERROR, MQEREASON_NO_MATCHING_MSG`. This allows you to create an application that reads all the available messages on a queue.

Java

Encasing the `getMessage()` call inside a `try..catch` block allows you to test the code of the resulting exception. This is done using the `code()` method of the MQeException class. You can compare the result from the `code()` method with a list of exception constants published by the MQe class. If the exception is not of type `Except_Q_NoMatchingMsg`, throw the exception again.

The following code shows this technique:

```
try
{
  while(true)
    { /* keep getting messages until
      an exception is thrown    */
    MQeMsgObject msg = qmgr.getMessage( "myQMgr", "myQueue",
                            null, null, 0 );
    processMessage(msg);
    }
}
catch (Exception e)
{
    if ( e.code() != MQe.Except_Q_NoMatchingMsg )
    throw e;
}
```

Therefore, you can read all messages from a queue by iteratively getting messages until `MQe.Except_Q_NoMatchingMsg` is returned.

C

You can read all messages from a queue by looping, until the return code is MQERETURN_QUEUE_WARNING and the reason code is MQEREASON_NO_MATCHING_MSG.

Browse and Lock

Performing BrowseAndLock on a group of messages allows an application to ensure that no other application is able to process messages when they are locked. The messages remain locked until that application unlocks them. No other application can unlock the messages. Any messages that arrive on the queue after the BrowseAndLock operation are not locked.

An application can perform either a get or a delete operation on the messages to remove them from the queue. To do this, the application must supply the lockID that is returned with the enumeration of messages.

Specifying the lockID allows applications to work with locked messages without having to unlock them first.

Instead of removing the messages from the queue, it is also possible just to unlock them. This makes them visible once again to all MQe applications. You can achieve this by using the unlockMessage method.

Note: See the MQe Configuration Guide for special considerations with MQ bridge queues.

Example - Java:

Example of BrowseAndLock (Java)

The MQeMessageEnumerationMQeEnumeration object contains all the messages that match the filter supplied to the browse. MQeEnumeration can be used in the same manner as the standard Java Enumeration. You can enumerate all the browsed messages as follows:

Note: You must supply a confirmID, in case the action of locating messages fails. It must be possible to undo the location, and this action requires the confirmID.

```
long confirmID = MQe.uniqueValue();
MQeEnumeration msgEnum = qmgr.browseMessagesAndLock( null,
                    "MyQueue",
                    null, null,
                             confirmID, false);

while( msgEnum.hasMoreElements() )
{
    MQeMsgObject msg = (MQeMsgObject)msgEnum.nextElement();
    System.out.println( "Message from  queue manager: " +
                        msg.getAscii( MQe.Msg_OriginQMgr ) );
}
```

The following code performs a delete on all the messages returned in the enumeration. The message's UniqueID and lockID are used as the filter on the delete operation:

```
while(msgEnum.hasMoreElements())
{
    MQeMsgObject msg = (MQeMsgObject)
                msgEnum.getNextMessage(null,0);

    processMessage(msg);

    MQeFields filter = msg.getMsgUIDFields();
    filter.putLong(MQe.Msg_LockID,
              msgEnum.getLockId());

    qmgr.deleteMessage(null, "MyQueue", filter);
}
```

Example - C:

Example of BrowseAndLock (C)

The C code base example gets the actual message. Note the additional parameters, a `confirmID` in case the operation needs undoing, and the `lockID`.

```
 MQeVectorHndl hMessages;
 MQEINT64 lockID, confirmID=42;
 rc = mqeQueueManager_browseAndLock(hQueueManager,
                            &exceptBlock,
                            &hmessages,
                            &lockID,
                            hQueueManagerName,
                            hQueueName,
                            hFilter,
                            NULL,      /*No Attribute*/
                            confirmID,
                            MQE_TRUE);    /*Just UIDs*/
 /*process vector*/
 MQeFieldsHndl hGetFilter;
 rc = mqeFields_new(&exceptBlock, &hGetFilter);
 if (MQERETURN_OK == rc){
   rc = mqeFields_putInt64(&hGetFilter,
                    &exceptBlock,
                    MQE_MSG_LOCKID,
                    lockID);
     if (MQERETURN_OK == rc){
       rc = mqeQueueManager_getMessage(&hQueueManager,
                             &exceptBlock,
                             hQueueManagerName,
                             hQueueName,
                            hGetFilter,
                            &hMsg);
 }
```

Message listeners

Note: This section does not apply to the C code base.

MQe allows an application to *listen* for events occurring on queues. The application is able to specify message filters to identify the messages in which it is interested, as shown in the following Java example:

```
/* Create a filter for "Order" messages of priority 7  */
MQeFields filter = new MQeFields();
filter.putAscii( "MsgType", "Order" );
filter.putByte( MQe.Msg_Priority, (byte)7 );
/* activate a listener on "MyQueue"        */
qmgr.addMessageListener( this, "MyQueue", filter );
```

The following parameters are passed to the addMessageListener() method:

- The name of the queue on which to listen for message operations
- A *`callback`* object that implements MQeMessageListenerInterface
- An MQeFields object containing a message filter

When a message arrives on a queue with a listener attached, the queue manager calls the `callback` object that it was given when the message listener was created.

The following is an example of the way in which an application would normally handle message events in Java:

```
public void messageArrived(MQeMessageEvent msgEvent)
  {
    String queueName =msgEvent.getQueueName();
    if (queueName.equals("MyQueue"))
    {
           try
           {
      /*get message from queue */
      MQeMsgObject msg =qmgr.getMessage(null,queueName,
            msgEvent.getMsgFields(),null,0);

      processMessage(msg );
           }
           catch (MQeException  e)
          {
           ...
          }
    }
  }
```

messageArrived() is a method implemented in MQeMessageListenerInterface. The `msgEvent` parameter contains information about the message, including:

- The name of the queue on which the message arrived
- The `UID` of the message
- The `messageID`
- The `correlationID`
- Message `priority`

Message filters only work on local queues. A separate technique known as polling allows messages to be obtained as soon as they arrive on remote queues.

Message polling

Note: This section does not apply to the C code base.

Message polling uses the mqeQueueManager_waitForMessage() method. This command issues a mqeQueueManager_getMessage() command to the remote queue at regular intervals. As soon as a message that matches the supplied filter becomes available, it is returned to the calling application.

A wait for message call typically looks like this:

```
  qmgr.waitForMessage( "RemoteQMgr", "RemoteQueue",
                filter, null, 0, 60000 );
```

The mqeQueueManager_waitForMessage() method polls the remote queue for the length of time specified in its final parameter. The time is specified in milliseconds, so in the example above, the polling lasts for 60 seconds. The thread on which the command is executing is blocked for this length of time, unless a message is returned earlier.

Message polling works on both local and remote queues.

Note: Use of this technique results in multiple requests being sent over the network.

Trigger transmission

This method attempts to transmit pending messages. Only unlocked messages are transmitted.

Asynchronous remote queues and home server queues respond to trigger transmission processing. Put messages with no `confirmID` or put messages and confirm them before calling this method. Only messages that are fully 'put' can be transmitted.

Trigger transmission rules

There are a number of rules, which can control the trigger transmission processing, if processing occurs. See the Rules topic for more information.

```
rc = mqeQueueManager_triggerTransmission(hQueueManager,&exceptBlock);
```

Servlet

Overview of servlet queue managers, which run inside a Web server

As well as running as a standalone server, a queue manager can be encapsulated in a servlet to run inside a Web server . A servlet queue manager has nearly the same capabilities as a server queue manager. MQeServlet provides an example implementation of a servlet. As with the server, servlets use ini files to hold start up parameters. A servlet uses many of the same MQe components as the server.

The main component not required in a servlet is the connection listener, this function is handled by the Web server itself. Web servers only handle http data streams so any MQe client that wishes to communicate with an MQe servlet must use the http adapter (com.ibm.mqe.adapters.MQeTcpipHttpAdaper). When you configure connections to queue managers running in servlets, you must specify the name of the servlet in the parameters field of the connection.

Example - configuring a connection on a servlet

The following definitions configure a connection on servlet /servlet/MQe with queue manager `PayrollQM`:

Connection name
    `PayrollQM`

Channel
    `com.ibm.mqe.communications.MQeChannel`

    Note: The com.ibm.mqe.MQeChannel class has been moved and is now known as com.ibm.mqe.communications.MQeChannel. Any references to the old class name in administration messages is replaced automatically with the new class name.

Channel Adapter
    `com.ibm.mqe.adapters.MQeTcpipAdapter:192.168.0.10:80`

Parameters
    `/servlet/MQe`

Options

Example - configuring a connection on a servlet using aliases

If the relevant aliases have been set up, you can configure the connection as follows:

Connection name
    `PayrollQM`

Channel
    `DefaultChannel`

Adapter
    `Network:192.168.0.10:80`

Parameters
    /servlet/MQe

Options

Differences between server and servlet startup

The main differences compared to a server startup are:

- The servlet overrides the init method of the superclass. This method is called by the Web server to start the servlet. Typically this occurs when the first request for the servlet arrives.
- The name of the startup ini file cannot be passed in from the command line. The example expects to obtain the name using the servlet method getInitParameter() which takes the name of a parameter and returns a value. The MQe servlet uses a *Startup* parameter that it expects to contain an ini file name. The mechanism for configuring parameters in a Web server is Web server dependant.
- A listener is not started as the Web server handles all network requests on behalf of the servlet.
- As there is no listener a mechanism is required to time-out connections that have been inactive for longer than the time-out period. A simple timer class MQeChannelTimer is instantiated to perform this function. The *TimeInterval* value is the only parameter used from the [Listener] section of the ini file.

Example - starting a servlet

The MQe servlet extends C servlet namejavax.servlet.http.HttpServlet and overrides methods for starting, stopping and handling new requests. The following code fragment starts a servlet:

C example

```
/**
 * Servlet initialization......
 */
public void init(ServletConfig sc) throws ServletException
{
  // Ensure supers constructor is called.
  super.init(sc);

  try
  {
    // Get the the server startup ini file
    String startupIni;
    if ((startupIni = getInitParameter("Startup")) == null)
      startupIni = defaultStartupInifile;

    // Load it
    MQeFields sections = MQeQueueManagerUtils.loadConfigFile(startupIni);

    // assign any class aliases
    MQeQueueManagerUtils.processAlias(sections);

    // Uncomment the following line to start trace before the queue
    // manager is started
    //     MQeQueueManagerUtils.traceOn("MQeServlet Trace", null);

    // Start connection manager
    channelManager = MQeQueueManagerUtils.processChannelManager(sections);

    // check for any pre-loaded classes
    loadTable = MQeQueueManagerUtils.processPreLoad(sections);

    // setup and activate the queue manager
    queueManager = MQeQueueManagerUtils.processQueueManager(sections,
     channelManager.getGlobalHashtable( ));

    // Start ChannelTimer  (convert time-out from secs to millisecs)
    int tI =
      sections.getFields(MQeQueueManagerUtils.Section_Listener).getInt
```

```
                                    ("TimeInterval");
    long timeInterval = 1000 * tI;
    channelTimer = new MQeChannelTimer(channelManager, timeInterval);

    // Servlet initialization complete
    mqe.trace(1300, null);
  }
  catch (Exception e)
  {
    mqe.trace(1301, e.toString());
    throw new ServletException(e.toString());
  }
}
```

Example - handling incoming requests

A servlet relies on the Web server for accepting and handling incoming requests. Once the Web server has decided that the request is for an MQe servlet, it passes the request to MQe using the doPost() method. The following code handles this request:

```
C example
```

```
/**
 * Handle POST......
 */
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
              throws IOException
{
  // any request to process ?
  if (request == null)
    throw new IOException("Invalid request");
  try
  {
    int max_length_of_data = request.getContentLength();
    // data length
    byte[] httpInData = new byte[max_length_of_data];
    // allocate data area
    ServletOutputStream httpOut = response.getOutputStream();
    // output stream
    ServletInputStream  httpIn  = request.getInputStream();
    // input stream

    // get the request
    read( httpIn, httpInData, max_length_of_data);

    // process the request
    byte[] httpOutData = channelManager.process(null, httpInData);

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

    // Pass back the response
    httpOut.write(httpOutData);
  }
  catch (Exception e)
  {
    // pass it on ...
    throw new IOException( "Request failed" + e );
  }
}
```

This method:

1. Reads the http input data stream into a `byte array`. The input data stream may be buffered so the read() method is used to ensure that the entire data stream is read before continuing.

   Note: MQe only handles requests with the doPost() method, it does not accept requests using the doGet() method
2. The request is passed to MQe through a connection manager. From this point, all processing of the request is handled by core MQe classes such as the queue manager.
3. Once MQe has completed processing the request, it returns the result wrapped in http headers as a byte array. The byte array is passed to the Web server and is transmitted back to the client that originated the request.

Running multiple servlets on a web server

Web servers can run multiple servlets. It is possible to run multiple different MQe servlets within a Web server, with the following restrictions:
- Each servlet must have a unique name
- Only one queue manager is allowed per servlet
- Each MQe servlet must run in a different Java Virtual Machine (JVM)

Message delivery

Details of the different types of message delivery process

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

Asynchronous message delivery

An asynchronous put to a remote queue places the message on the backing store associated with the local definition of that queue, along with its destination queue manager name, queue name, and the compressor, authenticator, and cryptor characteristics that match the target destination of the message. The message's dump method is called as it is saved to persistent storage in a secure format that is defined by its destination queue. The queue manager controls message delivery. It identifies or establishes a connection with appropriate characteristics to the queue manager for the next hop, then creates or reuses a transporter to the target queue manager. The transporter dumps the message and transmits the resulting byte string. The target queue manager and queue name are not part of that message flow.

If appropriate, the message is encrypted and compressed over the connection. If it has reached its destination queue manager, it is decrypted and decompressed. A new message is created, using the restore method, and the resultant message is placed on the destination queue. If the message has not reached its destination queue manager, it is decrypted and decompressed. It is then re-encrypted, compressed, and placed on a store-and-forward queue for onward transmission, if a store-and-forward queue exists. In both cases it is held on its respective queue in a secure format, as defined by its destination queue.

A characteristic of asynchronous message delivery is that messages are passed to the queue manager at intermediate hops, being queued for onward transmission. Messages are taken off the intermediate queues first in order of priority, then in order of arrival on the queue. Duplicate messages, created when you resend a message, are also taken off the intermediate queues in the order of their arrival on the queue.

Synchronous message delivery

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

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

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

Assured and non-assured message delivery

Message delivery using synchronous message transmission can be assured or non-assured.

Assured message delivery

Asynchronous transmission introduces the concept of *assured message delivery*. When delivering messages asynchronously, MQe delivers each message once, and once-only, to its destination queue. However, this assurance is only valid if the definition of the remote queue and remote queue manager match the current characteristics of the remote queue and remote queue manager. If a remote queue definition and the remote queue do not match, then it is possible that a message may become undeliverable. In this case the message is not lost, but remains stored on the local queue manager.

Non-assured message delivery

Non-assured delivery of a message takes place in a single network flow. The queue manager sending the message creates or reuses a channel to the destination queue manager.

The message to be sent is dumped to create a byte-stream, and this byte stream is given to the channel for transmission. Once program control has returned from the channel the sender queue manager knows that the message has been successfully given to the target queue manager, that the target has logged the message on a queue, and that the message has been made visible to MQe applications.

However, a problem can occur if the sender receives an exception over the channel from the target. The sender has no way of knowing if the exception occurred before or after the message was logged and made visible. If the exception occurred before the message was made visible it is safe for the sender to send the message again. However, if the exception occurred after the message was made visible, there is a danger of introducing duplicate messages into the system since an MQe application could have processed the message before it was sent the second time.

The solution to this problem involves transmitting an additional confirmation flow. If the sender application receives a successful response to this flow, then it knows that the message has been delivered once and once-only.

Synchronous assured message delivery

You can perform assured message delivery using synchronous message transmission.

Put message - assured put

You can perform assured message delivery using synchronous message transmission, but the application must take responsibility for error handling.

The `confirmID` parameter of the `putMessage` method dictates whether a confirm flow is expected or not. A value of `zero` means that message transmission occurs in one flow, while a value of greater than zero means that a confirm flow is expected. The target queue manager logs the message to the destination queue as usual, but the message is locked and invisible to MQe applications, until a confirm flow is received. When you put messages with the `confirmID`, the messages are ordered by confirm time, not arrival time.

an MQe application can issue a `put` message confirmation using the mqeQueueManager_`confirmPutMessage` method. Once the target queue manager receives the flow generated by this command, it unlocks the message, and makes it visible to MQe applications. You can confirm only one message at a time. It is not possible to confirm a batch of messages.

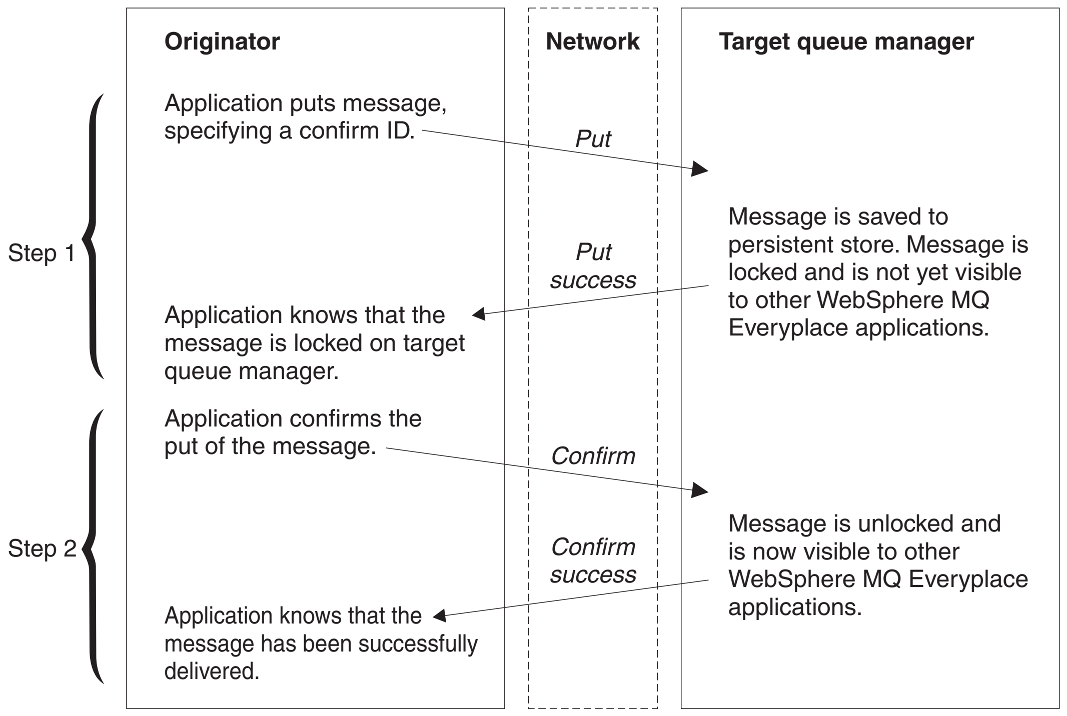

Originator
Network
Target queue manager
Step 1
Application puts message, specifying a confirm ID.
Put
Message is saved to persistent store. Message is locked and is not yet visible to other WebSphere MQ Everyplace applications.
Put success
Application knows that the message is locked on target queue manager.
Step 2
Application confirms the put of the message.
Confirm
Message is unlocked and is now visible to other WebSphere MQ Everyplace applications.
Confirm success
Application knows that the message has been successfully delivered.

Figure 6. Assured put of synchronous messages

The mqeQueueManager_`confirmPutMessage()` method requires you to specify the `UniqueID` of the message, not the `confirmID` used in the prior put message command. The `confirmID` is used to restore messages that remain locked after a transmission failure.

Example (Java) - assured put:

A skeleton version of the code required for an assured `put` is shown below:

```
long confirmId = MQe.uniqueValue();

try
```

```
{
  qmgr.putMessage( "RemoteQMgr", "RemoteQueue",
                   msg, null, confirmId );
}
catch( Exception e )
{
  /* handle any exceptions*/
}

try
{
    qmgr.confirmPutMessage( "RemoteQMgr", "RemoteQueue",
                            msg.getMsgUIDFields() );
}
catch ( Exception e )
{
  /* handle any exceptions    */
}
```

Example (C) - assured put:

A skeleton version of the code required for an assured put is shown below:

```
/* generate confirm Id */
MQEINT64 confirmId;
rc = mqe_uniqueValue(&exceptBlock,
                     &confirmId);

/* put message to queue using this confirm Id */
if(MQERETURN_OK == rc) {
    rc = mqeQueueManager_putMessage(hQMgr,
                         &exceptBlock,
                         hQMgrName, hQName,
                         hMsg, NULL, confirmId);
    /* now confirm the message put */
    if(MQERETURN_OK == rc) {
        /* first get the message uid fields */
           MQeFieldsHndl hFilter;
           rc = mqeFieldsHelper_getMsgUidFields(hMsg,
                                 &exceptBlock,
                                &hFilter);
          if(MQERETURN_OK == rc) {
            rc = mqeQueueManager_confirmPutMessage(hQMgr,
                                 &exceptBlock,
                                 hQMgrName,
                                 hQName, hFilter);
          }
    }
}
```

Exception handling - put message:

If a failure occurs during step 1 in "Put message - assured put" on page 51, the application should retransmit the message. There is no danger of introducing duplicate messages into the MQe network since the message at the target queue manager is not made visible to applications until the confirm flow has been successfully processed.

If the MQe application retransmits the message, it should also inform the target queue manager that this is happening. The target queue manager deletes any duplicate copy of the message that it already has. The application sets the `MQE_MSG_RESENDMQe.Msg_Resend` field to do this.

If a failure occurs during step 2 in "Put message - assured put" on page 51, the application should send the confirm flow again. There is no danger in doing this since the target queue manager ignores any confirm flows it receives for messages that it has already confirmed. This is shown in the following example, taken from the example program examples.application.example6.

Example - Java:

This example is taken from the examples.application.example6 example application:

```
boolean msgPut     = false;
  /* put successful?    */
boolean msgConfirm = false;
  /* confirm successful?    */
int maxRetry       = 5;
  /* maximum number of retries    */

long confirmId = MQe.uniqueValue();

int retry = 0;
while( !msgPut &&
        retry < maxRetry )
{
  try
  {
    qmgr.putMessage( "RemoteQMgr",
              "RemoteQueue",
              msg, null,
              confirmId );
   msgPut = true;
/* message put successful             */
  }
  catch( Exception e )
  {
    /* handle any exceptions  */
    /* set resend flag for
    retransmission of message  */
    msg.putBoolean( MQe.Msg_Resend, true );
    retry ++;
  }
}

if ( !msgPut )
  /* was put message successful?*/
    /* Number of retries has
    exceeded the maximum allowed,
   /*so abort the put*/
   /* message attempt */
return;

retry = 0;
while( !msgConfirm &&
          retry < maxRetry )
{
  try
  {
    qmgr.confirmPutMessage( "RenoteQMgr",
                "RemoteQueue",
                        msg.getMsgUIDFields());
    msgConfirm = true;
/* message confirm successful*/
  }
  catch ( Exception e )
  {
    /* handle any exceptions*/
    /* An Except_NotFound
    exception means */
  /*that the message has already    */
    /* been confirmed */
    if ( e instanceof MQeException &&
       ((MQeException)e).code() == Except_NotFound )
      putConfirmed = true;
    /* confirm successful */
    /* another type of exception -
    need to reconfirm message */
    retry ++;
  }
}
```

Example - C:

This example is taken from the examples.application.example6 example application:

```
MQEINT32 maxRetry = 5;

rc = mqeQueueManager_putMessage(hQMgr,
                    &exceptBlock,
                    hQMgrName,
                    hQName, hMsg,
                    NULL, confirmId);

/* if the put attempt fails,
    retry up to the maximum number*/
/*of retry times permitted,
    setting the re-send flag. */
while (MQERETURN_OK != rc
          && --maxRetry > 0 ) {
    rc = mqeFields_putBoolean(hMsg, &exceptBlock,
                      MQE_MSG_RESEND, MQE_TRUE);
    if(MQERETURN_OK == rc) {
       rc = mqeQueueManager_putMessage(hQMgr, &exceptBlock,
                            hQMgrName, hQName,
                            hMsg, NULL, confirmId);
    }
}

if(MQERETURN_OK == rc) {
    MQeFieldsHndl hFilter;
    maxRetry = 5;
    rc = mqeFieldsHelper_getMsgUidFields(hMsg,
                        &exceptBlock,
                        &hFilter);
    if(MQERETURN_OK == rc) {
          rc = mqeQueueManager_confirmPutMessage(hQMgr,
                          &exceptBlock,
                          hQMgrName, hQName,
                          hFilter);
    }
     while (MQERETURN_OK != rc
                && --maxRetry > 0 ) {
             rc = mqeQueueManager_confirmPutMessage(hQMgr,
                                    &exceptBlock,
                                      hQMgrName,
                                      hQName,
                                      hFilter);
    }
}
```

Get message - assured get

Assured message `get` works in a similar way to `put`. If a `get` message command is issued with a `confirmId` parameter greater than zero, the message is left locked on the queue on which it resides until a confirm flow is processed by the target queue manager. When a confirm flow is received, the message is deleted from the queue. Figure 7 on page 55 describes a get of synchronous messages:

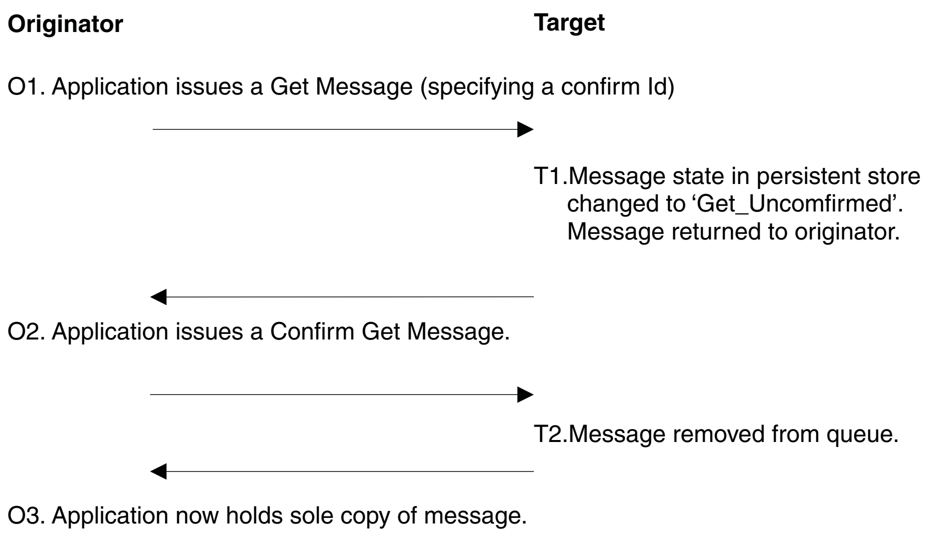

Originator
Target
O1. Application issues a Get Message (specifying a confirm Id)
T1.Message state in persistent store changed to 'Get_Uncomfirmed'. Message returned to originator.
O2. Application issues a Confirm Get Message.
T2.Message removed from queue.
O3. Application now holds sole copy of message.

Figure 7. Assured get of synchronous messages

Example (Java) - assured get:

This example code is taken from the examples.application.example6 example program.

```
boolean msgGet     = false;
/* get successful?    */
boolean msgConfirm = false;
/* confirm successful?    */
MQeMsgObject msg   = null;
int maxRetry       = 5;
/* maximum number of retries    */

long confirmId = MQe.uniqueValue();
int retry = 0;
while( !msgGet && retry < maxRetry)
{
  try
  {
    msg = qmgr.getMessage( "RemoteQMgr",
                         "RemoteQueue",
                         filter, null,
                         confirmId );
    msgGet = true;
  /* get succeeded    */
  }
  catch ( Exception e )
  {
    /* handle any exceptions */
    /* if the exception is of type
      Except_Q_NoMatchingMsg, meaning that   */
    /* the message is unavailable
      then throw the exception  */

    if ( e instanceof MQeException )
      if ( ((MQeException)e).code() ==
                   Except_Q_NoMatchingMsg )
        throw e;
    retry ++;
  /* increment retry count    */
  }
}

if ( !msgGet )
  /* was the get successful?        */
```

```
    /* Number of retry attempts has
    exceeded the maximum allowed, so abort  */
   /* get message operation   */
  return;

while( !msgConfirm && retry < maxRetry )
{
  try
  {
    qmgr.confirmGetMessage( "RemoteQMgr",
                            "RemoteQueue",
                            msg.getMsgUIDFields() );
    msgConfirm = true;
  /* confirm succeeded    */
  }
  catch ( Exception e )
  {
    /* handle any exceptions */
    retry ++;    /* increment retry count */
  }
}
```

Example (C) - assured get:

This example code is taken from the examples.application.example6 example program.

```
MQEINT32 maxRetry = 5;

rc = mqeQueueManager_getMessage(hQMgr,
                           &exceptBlock,
                              hQMgrName,
                              hQName, hMsg,
                              NULL, confirmId);

/* if the get attempt fails, retry
    up to the maximum number of*/
/*retry times permitted,
  setting the re-send flag. */
while (MQERETURN_OK != rc  &&
                           --maxRetry > 0 ) {
    rc = mqeFields_getBoolean(hMsg,
                         &exceptBlock,
                            MQE_MSG_RESEND,
                            MQE_TRUE);
    if(MQERETURN_OK == rc) {
        rc = mqeQueueManager_getMessage(hQMgr,
                                &exceptBlock,
                                   hQMgrName,
                                   hQName, hMsg,
                                   NULL,
                                   confirmId);
    }
}

if(MQERETURN_OK == rc) {
    MQeFieldsHndl hFilter;
    maxRetry = 5;
    rc = mqeFieldsHelper_getMsgUidFields(hMsg,
                                   &exceptBlock,
                                   &hFilter);
    if(MQERETURN_OK == rc) {
          rc = mqeQueueManager_confirmGetMessage(hQMgr,
                                        &exceptBlock,
                                           hQMgrName,
                                           hQName,
                                           hFilter);
    }
```

```
     while (MQERETURN_OK != rc  &&
                         --maxRetry > 0 ) {
             rc = mqeQueueManager_confirmPutMessage(hQMgr,
                                         &exceptBlock,
                                            hQMgrName,
                                            hQName,
                                            hFilter);
    }
}
```

Undo command:

The value passed as the `confirmId` parameter also has another use. The value is used to identify the message while it is locked and awaiting confirmation. If an error occurs during a `get` operation, it can potentially leave the message locked on the queue. This happens if the message is locked in response to the `get` command, but an error occurs before the application receives the message. If the application reissues the `get` in response to the exception, then it will be unable to obtain the same message because it is locked and invisible to MQe applications.

However, the application that issued the `get` command can restore the messages using the `undo` method. The application must supply the `confirmId` value that it supplied to the `get` message command. The `undo` command restores messages to the state they were in before the `get` command.

The `undo` command also has relevance for the mqeQueueManager_`putMessage` and mqeQueueManager_`browseMessagesAndLock` commands. As with `get` message, the `undo` command restores any messages locked by the mqeQueueManager_`browseMessagesandLock` command to their previous state.

If an application issues an `undo` command after a failed mqeQueueManager_`putMessage` command, then any message locked on the target queue awaiting confirmation is deleted.

The `undo` command works for operations on both local and remote queues.

Undo command example - Java:

```
boolean msgGet     = false;
/* get successful?    */
boolean msgConfirm = false;
/* confirm successful?    */
MQeMsgObject msg   = null;
int maxRetry       = 5;
/* maximum number of retries    */

long confirmId = MQe.uniqueValue();
int retry = 0;
while( !msgGet && retry < maxRetry )
{
  try
  {
    msg = qmgr.getMessage( "RemoteQMgr",
                    "RemoteQueue",
                    filter, null,
                            confirmId );
    msgGet = true;
  /* get succeeded    */
  }
  catch ( Exception e )
  {
    /* handle any exceptions   */
    /* if the exception is of type
      Except_Q_NoMatchingMsg, meaning that   */
    /* the message is unavailable
      then throw the exception */
    if ( e instanceof MQeException )
```

```
      if ( ((MQeException)e).code() == Except_Q_NoMatchingMsg )
        throw e;
    retry ++;    /* increment retry count    */
    /* As a precaution, undo the message
      on the queue. This will remove   */
    /* any lock that may have been put on
      the message prior to the        */
    /* exception occurring   */
    myQM.undo( qMgrName, queueName, confirmId );
  }
}

if ( !msgGet )
    /* was the get successful?        */
      /* Number of retry attempts has
      exceeded the maximum allowed, so abort  */
      /* get message operation   */
  return;

while( !msgConfirm && retry < maxRetry )
{
  try
  {
    qmgr.confirmGetMessage( "RemoteQMgr",
                    "RemoteQueue",
                            msg.getMsgUIDFields() );
    msgConfirm = true;
  /* confirm succeeded    */
  }
  catch ( Exception e )
  {
    /* handle any exceptions         */
    retry ++;
  /* increment retry count    */
  }
}
```

Undo command example - C:

```
MQeFieldsHndl hMsg;
rc = mqeQueueManager_getMessage(hQMgr, &exceptBlock,
                      &hMsg, hQMgrName,
                        hQName, hFilter,
                        NULL, confirmId);
/* if unsuccessful, undo the operation */
if(MQERETURN_OK != rc) {
    rc = mqeQueueManager_undo(hQMgr, &exceptBlock,
                      hQMgrName, hQName,
                      confirmId);
}
```

Network topologies and message resolution

Introduction to message routes and their use with MQe

Overview

This topic explains, in detail, the concept of message routes and how to use them with MQe.

Several features of MQe allow the routing of messages to be altered dynamically. However, you need to ensure that there are no ’in doubt’ messages that would be affected by the change. If a message is put with a non-zero confirm ID, and then the MQe network topology is changed to alter the routing of the subsequent confirmGetMessage call, the unconfirmed message will not be found. MQe protocol treats a failure to confirm a put as an indication that the put message has been confirmed already, and therefore assumes success. This could leave an unconfirmed message on a queue, which represents a loss of a message, and therefore breaks the assured delivery promise.

Since MQe uses the same two step process to assure delivery of asynchronously sent messages, regardless of whether a zero or non-zero confirmId is used, changing the network topology can break the assured delivery of asynchronous message sends.

Notation

The topics within *Network topologies and message resolution* use a consistent notation for illustrating the resources. This allows the areas of specific interest to be shown prominently, while the less relevant parts of a system can be hidden. This is easier to show with a diagram:

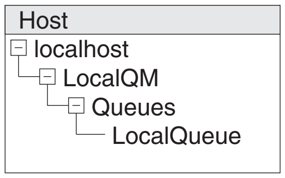


Figure 8. A host and the MQe resources on it

The following diagram shows the same resources in the 'dispersed' form:

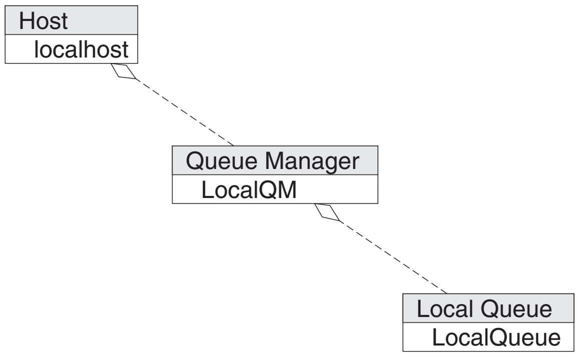


Figure 9. A host and the MQe resources on it: 'dispersed' form

The line with a diamond shape shows that the queue manager is the child of the host. This preserves the parent/child relationship from the tree, which would otherwise be lost by separating the elements.

Introduction

The route that a message takes through an MQe network can depend upon many resources (queues, connection definitions, listeners and so on). These need to be correctly set up, often in pairs whose settings need to be complementary. Failure to set up the correct resources, or setting certain of their values incorrectly can result in failure to deliver messages. Since the task of setting up a network that correctly routes messages can initially appear complex, this topic describes the theory underlying message resolution.

A common source of confusion with MQe is the differentiation between a local queue that exists on a remote machine (or queue manager), and a local definition of that queue on the remote machine. Both of these entities are commonly referred to as 'remote queue's. In order to clarify these, the term 'remote queue reference' is used to describe a local definition of a queue that resides on another (remote) machine (or queue manager).

Local queue resolution

Local message putting is fundamental to MQe. Messages, if they are to be useful, must always end up on a local queue. Message route resolution is the mechanism by which a message travels through an MQe network to its ultimate destination.

The following diagram shows a simple local message put.

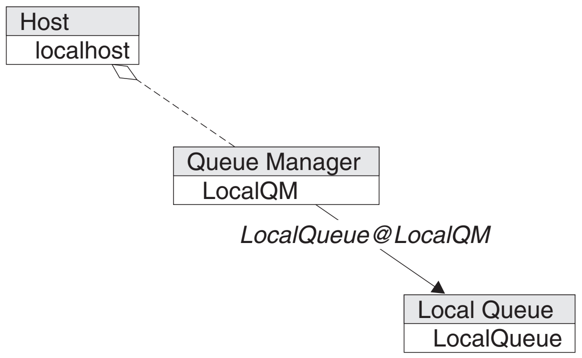


Figure 10. A simple local message put

The message route is shown for a message put to (QueueManager)LocalQM destined for the (Queue)LocalQueue@LocalQM. This is clearly a put to a local queue, as the queue's 'queue manager name' is the same as the name of the queue manager to which the message is put.

The message route is shown with an arrow labelled with the message route name. The arrow indicates the direction in which the message flows. The text on the label indicates the currently used target name (this can change during message resolution). LocalQM looks for a queue to accept a message for LocalQueue@LocalQM. The process of determining which queue to place a message on is called Queue Resolution. LocalQM finds an exact match for the destination, the local queue. It then puts the message onto the local queue. The message will then reside on the local queue until it is retrieved via the getMessage() API call.

Local queue alias

Local queues can have aliases. If we add a queue alias to the local queue we provide it with another name by which it will be known. So the local queue LocalQueue@LocalQM could be given an alias of 'LocalQueueAlias', as shown in the following diagram:

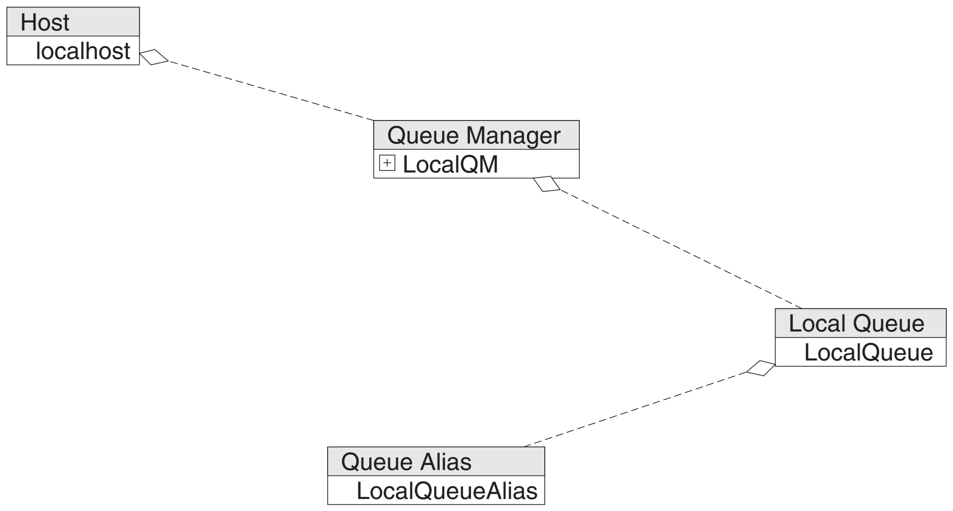


Figure 11. LocalQueue@LocalQM with an alias of 'QueueAlias'.

Messages addressed to LocalQueueAlias@LocalQM would be directed by the queue manager to LocalQueue@LocalQM. We could envisage this as the message being placed on the matching alias, almost as if the alias were a queue, and then the alias moves the message to the correct destination, as shown in the following diagram:

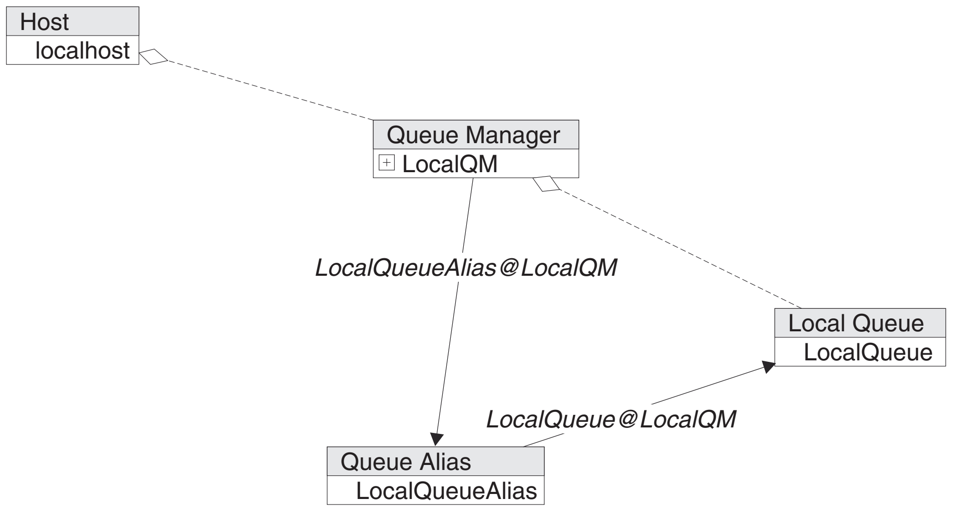


Figure 12. A message being placed on a matching alias

The redirection of the message by the alias is accompanied by a change in the 'destination queue name' from LocalQueueAlias@LocalQM to LocalQueue@LocalQM. The fact that the message was originally put to the alias is completely lost. This can be seen by the labelling of the message route from the alias to the queue. In this particular case the change of 'put name' is of little or no importance, but this is important in more complex message resolutions.

The resolution of the queue alias is performed just before the message is routed to the queue. The resolution is as late as it could possibly be, and is sometimes termed 'late resolution'.

Queue manager alias

Queue aliases enable you to refer to queues by more than one name. Queue Manager Aliases enable you to refer to queue managers by more than one name. We can define a Queue Manager Alias ’AliasQM’ referring to the local queue manager, as shown in the following diagram:

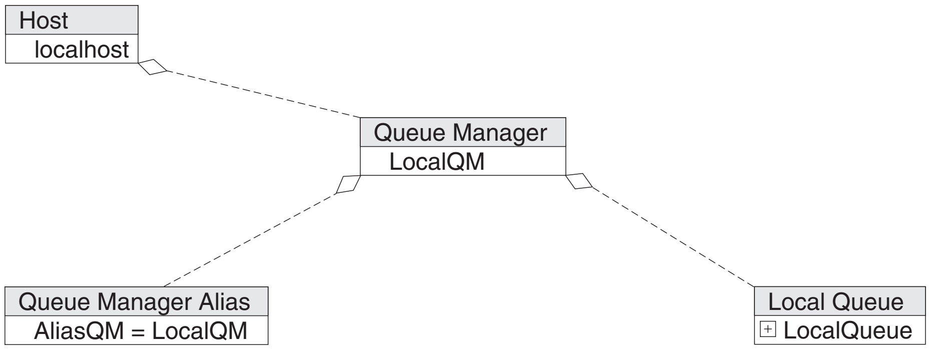


Figure 13. Defining a queue manager alias

Messages addressed to ’AliasQM’ are routed to ’LocalQM’, as shown in the following diagram:

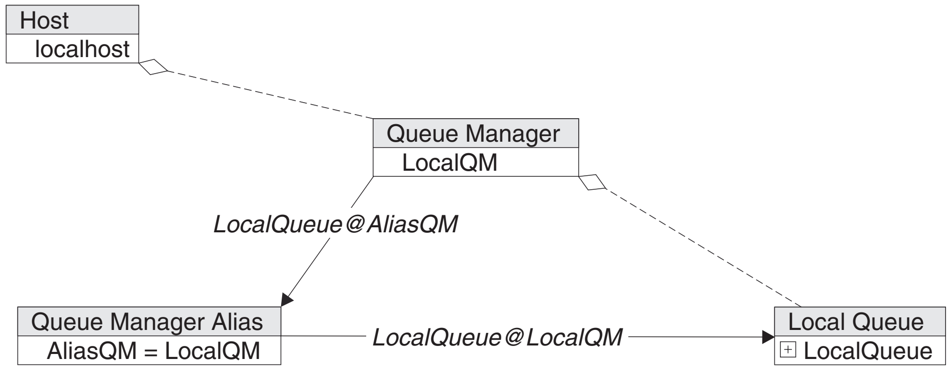


Figure 14. Addressing messages to a queue manager alias

The redirection of the message by the alias is accompanied by a change in the ’destination queue name’ from LocalQueue@AliasQM to LocalQueue@LocalQM. The fact that the message was originally put to the alias is completely lost. This can be seen by the labelling of the message route from the alias to the queue. Queue Manager Aliases are resolved at the beginning of message resolution. Queue Manager Aliases are very effective as part of complex topologies

To complete the picture we can resolve both the Queue Manager Alias and the Queue Alias, as shown in the following diagram:

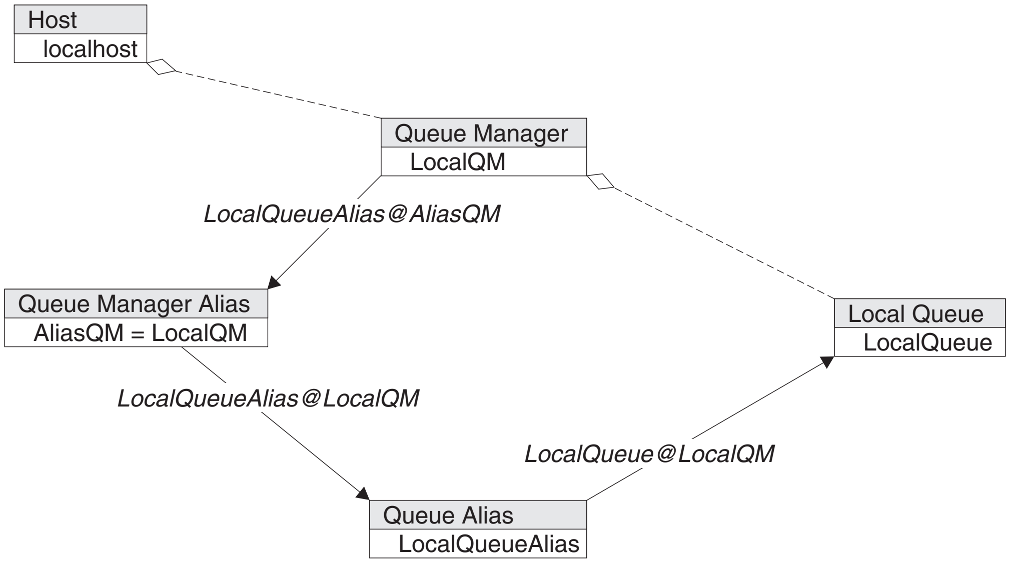


Figure 15. Resolving the queue manager alias and the queue alias

Here we put a message to LocalQueueAlias@AliasQM, and it is resolved first via the Queue Manager Alias, and then through the Queue Alias.

Resolution of queueManager aliases happens as soon as the request reaches a queue manager. The effect is to substitute the aliased string for the aliasing string. So for the first example above, as soon as the putMessage("AliasQM",....) call crosses the API, it is converted to a putMessage("LocalQM",....) call. This resolution is also performed when a message is put to a remote queue manager. On a remote queue manager the queue aliases on that queue manager are used, not those on the originating queue manager.

An alias can point to another alias. However, circular definitions have unpredictable results. An alias can also be made of the local queue manager name. This allows a queue manager to behave as if it were another queue manager. This pretence means that we can remove a queue manager entirely from the network, and by creating suitable queue manager aliases elsewhere we can allocate its workload to another queue manager. This feature is useful when modifying MQe network topologies, because servers, under the control of system administrators, can be moved, removed or renamed without breaking the connectivity of clients, which may not be so readily accessible.

Remote queue resolution

Remote queue resolution involves connection definitions and network resolution. It requires a setup where there are two queue managers, one of which is the local queue manager that you use to put the message, and the other is the queue manager to which you want the message to go. The remote queue manager must have a listener, and the local queue manager must have a connection definition describing the listener, as shown in the following diagram:

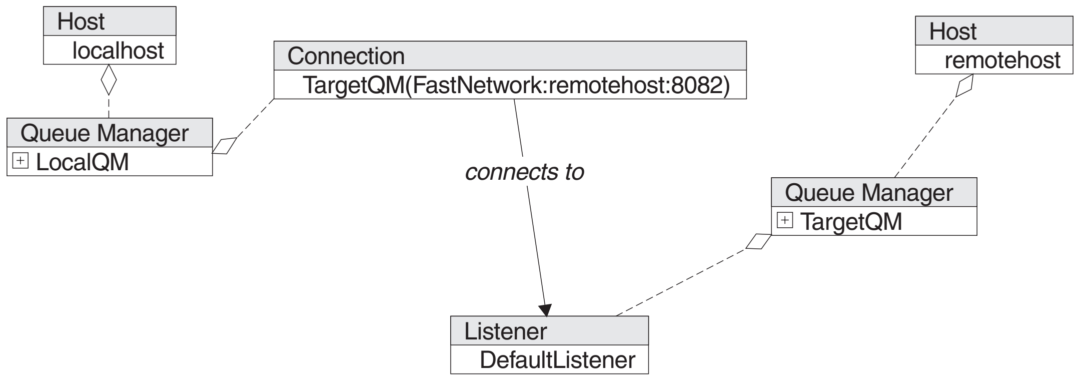


Figure 16. Local and remote queue managers with a definition and listener pair

The connection definition/listener pair allows MQe to establish the network communications necessary to flow the message. The connection definition contains information about communicating with a single queue manager. The connection definition is named for the queue manager to which it defines a route. So in this example the connection definition is called TargetQM, and contains the information necessary to establish connection with (QueueManager)TargetQM. This information includes the address of the machine upon which the queue manager resides (remote host in this example), the port upon which the queue manager is listening (8081 in this example), and the protocol to use when conversing with the queue manager (FastNetwork in this example).

You need a remote queue reference on LocalQM representing the destination queue TargetQueue which resides on TargetQM. There are therefore two entities called TargetQueue@TargetQM. One is the ’real’ queue, that is a local queue, and one is a reference to the real queue, a remote queue reference, as shown in the following diagram:

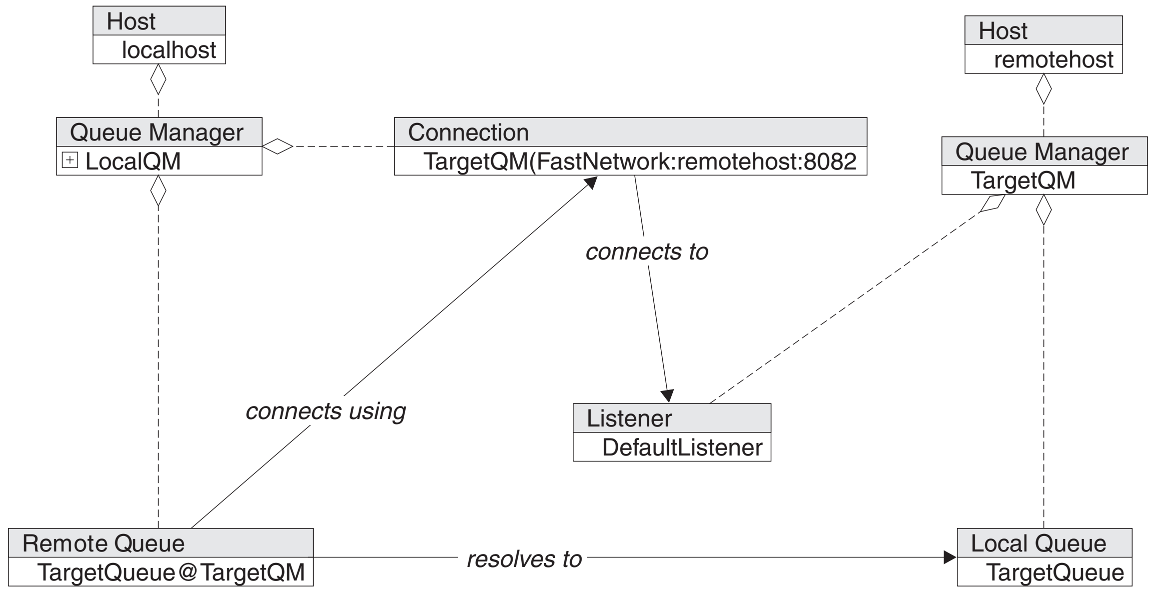


Figure 17. A remote queue reference.

The message resolution for a put on LocalQM to TargetQueue@TargetQM works as shown in the following diagram:

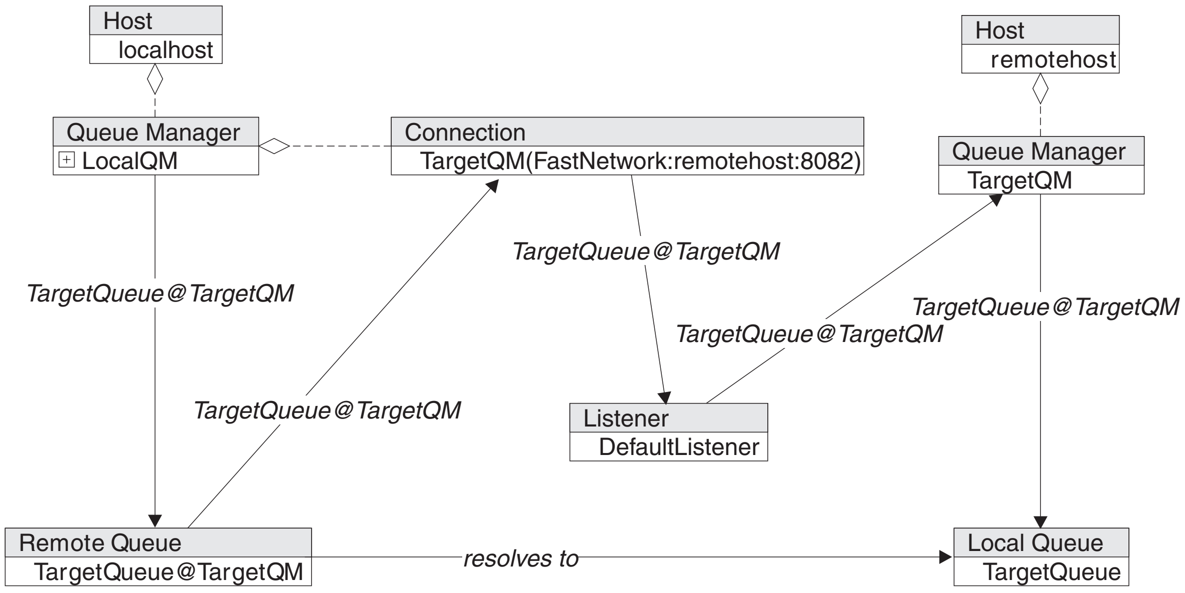


Figure 18. Message resolution for a put

The message route is as follows:

- The message is put on LocalQM addressed to TargetQueue@TargetQM.
- LocalQM performs queue resolution and finds the remote queue reference as an exact match. LocalQM places the message onto the remote queue reference.
- The remote queue reference then performs connection resolution. It looks for a connection that will allow it to pass the message to the queue manager owning the final queue. The remote queue reference finds the connection definition called TargetQM and passes the message to it.
- The connection definition now moves the message to its partner listener, which puts the message to the remote queue manager.
- The remote queue manager performs queue resolution just as if the message had been put locally, finds TargetQueue@TargetQM, and puts the message on it.

Although the connection definition and listener are vital to the message resolution, they do not affect the routing in this example. This is shown in the following diagram:

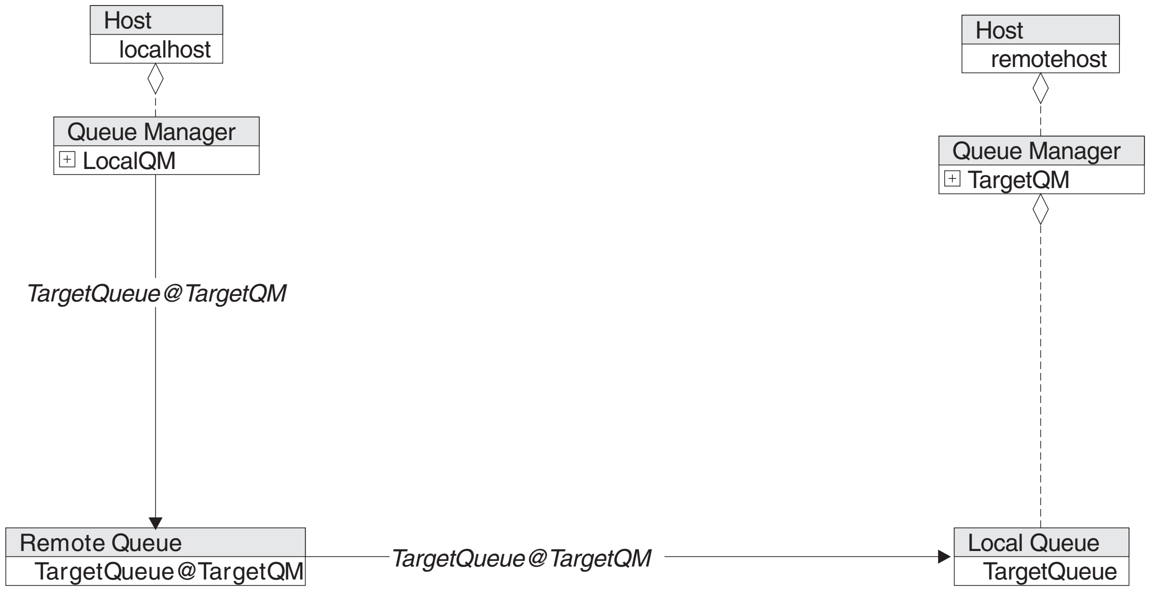


Figure 19. Message resolution for a put

In later examples the connection definitions play a more important role, and they are shown explicitly. For now assume the presence of the logical link formed by the listener and not show them in the diagrams. It is often much more convenient to use a simplified view of the message route. You can do this by thinking of the four elements that contribute to this message resolution as a single, composite, entity. This entity is a Message Route, as shown in the following diagram:

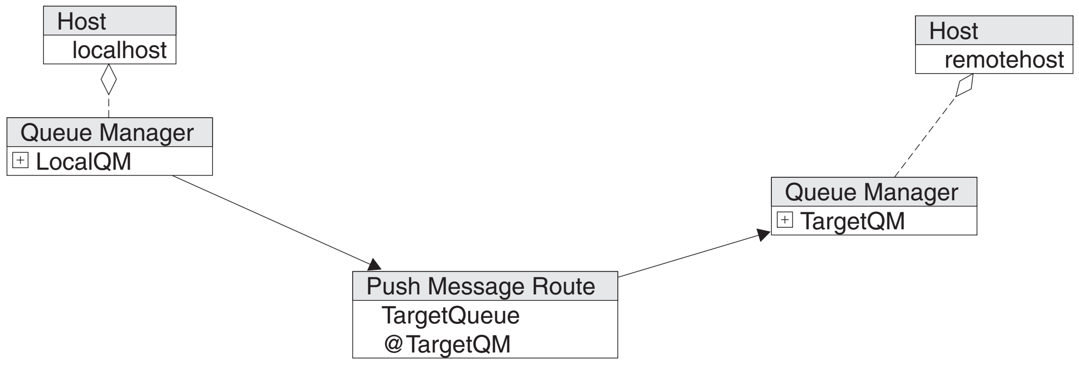


Figure 20. A message route entity

Here you can see the message route that indicates that all messages put to LocalQM and addressed to TargetQueue@TargetQM will be moved directly to the destination. A Message Route is valid only if all the necessary components (Connection Definition, Listener, Remote Queue Definition, and destination queue) are present and correctly configured.

The Message Route is defined as a Push Message Route because messages are pushed from the source queue to the destination queue, by LocalQM.

Aliases on remote queues

You can use aliases on the remote queue, as the last step is simply queue resolution performed on TargetQM. The Queue Alias on the target queue appears to the local system as if it were a queue. The remote queue definition on the local system is therefore named for the Queue Alias, rather than the target queue. The following diagram makes this clear (note that the connection definition and the listener are hidden):

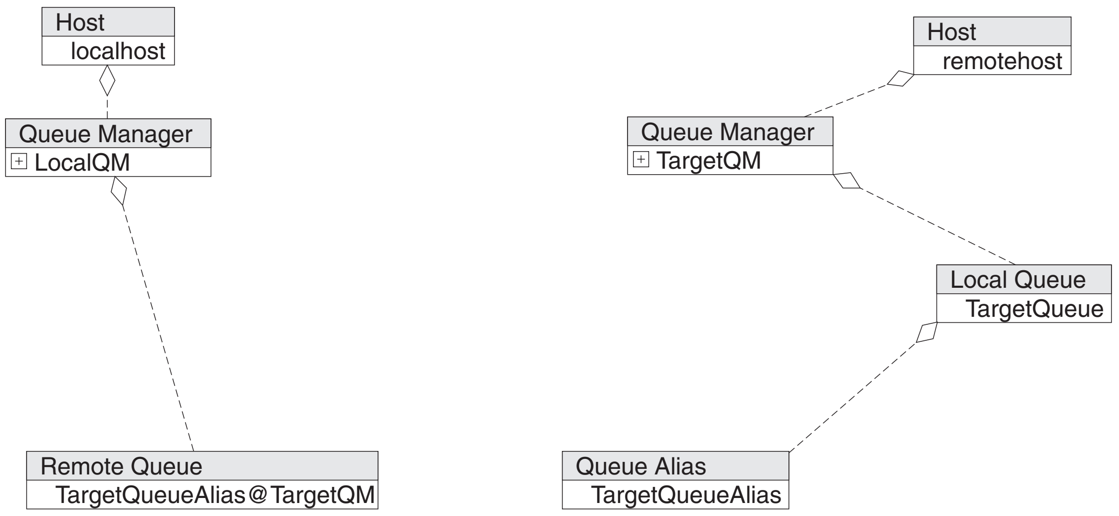


Figure 21. Using aliases on the remote queue

Here a remote queue reference is defined which actually refers to an alias for a queue on TargetQM. When you perform a put on LocalQM addressed to QueueAlias@TargetQM the resolution works as shown in the following diagram:

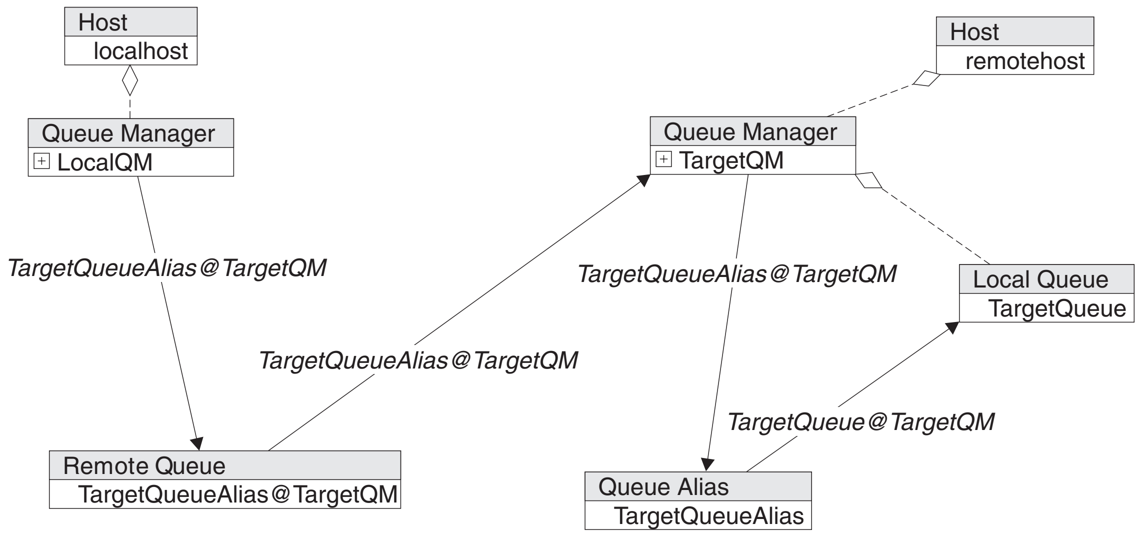


Figure 22. Message resolution for a put to a remote queue, using a Queue alias defined on TargetQM

- Queue resolution on LocalQM finds the remote queue reference. The fact that this is a reference to a queue alias is completely immaterial to queue resolution.
- Connection resolution works entirely as described above
- queue resolution on TargetQM now behaves exactly as local queue resolution of a queue alias described earlier.

Note that the destination name for the message remains QueueAlias@TargetQM until queue resolution onTargetQM. The Remote Queue Definition completes the requirements for another Message Route, as shown in the following diagram:

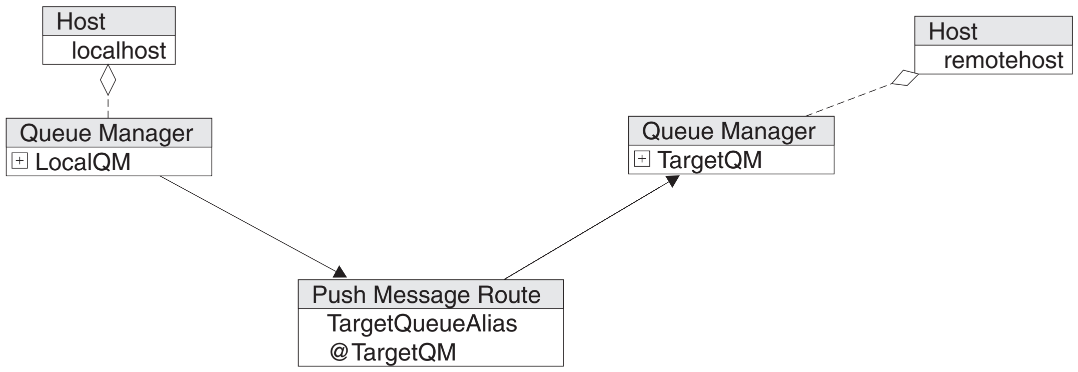


Figure 23. Message route entity of messages put to TargetQueueAlias on TargetQM

Parallel routes

Aliases allow the creation of parallel routes between a source and a destination. This is sometimes useful when you want to send messages synchronously if possible, but asynchronously if the remote end is not currently connected. You can do this with the setup illustrated in the following diagram:

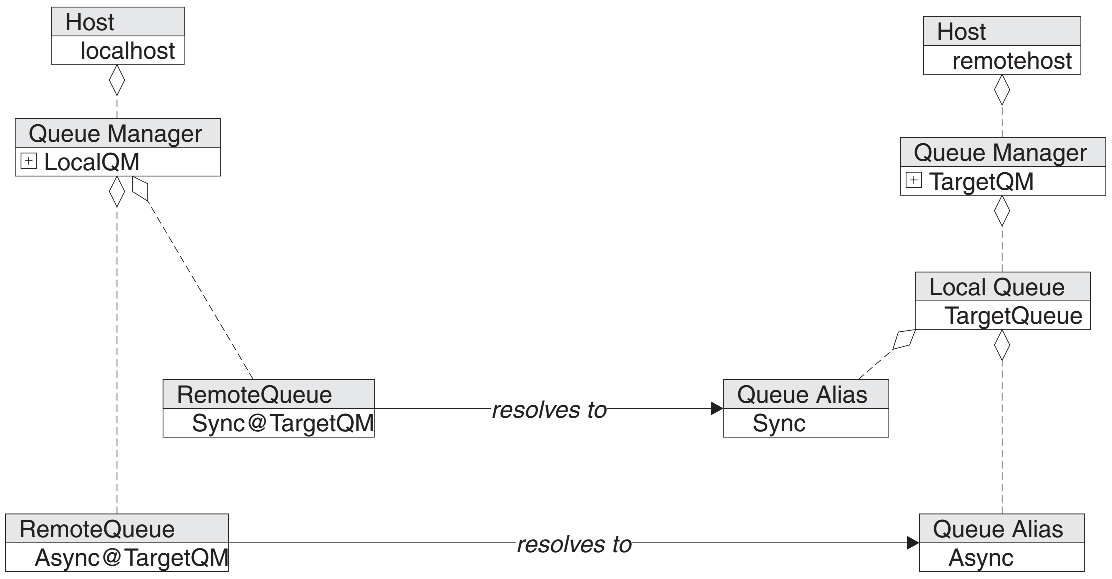


Figure 24. Creating parallel routes between source and destination

Here two aliases have been defined on the target queue. One alias will be used to route synchronous traffic to the target queue, one will be used to route asynchronous traffic.

On LocalQM two remote queue definitions have been defined, one pointing at each alias. You can create an asynchronous Remote Queue Definition called Async@TargetQM, and a synchronous Remote Queue Definition called Sync@TargetQM. By choosing the name of the queue that you put to (Sync@TargetQM or Async@TargetQM) you can choose the route that the message follows, even though the destination is the same. First, the resolution of the synchronous route by putting a message to Sync@TargetQM, as shown in the following diagram:

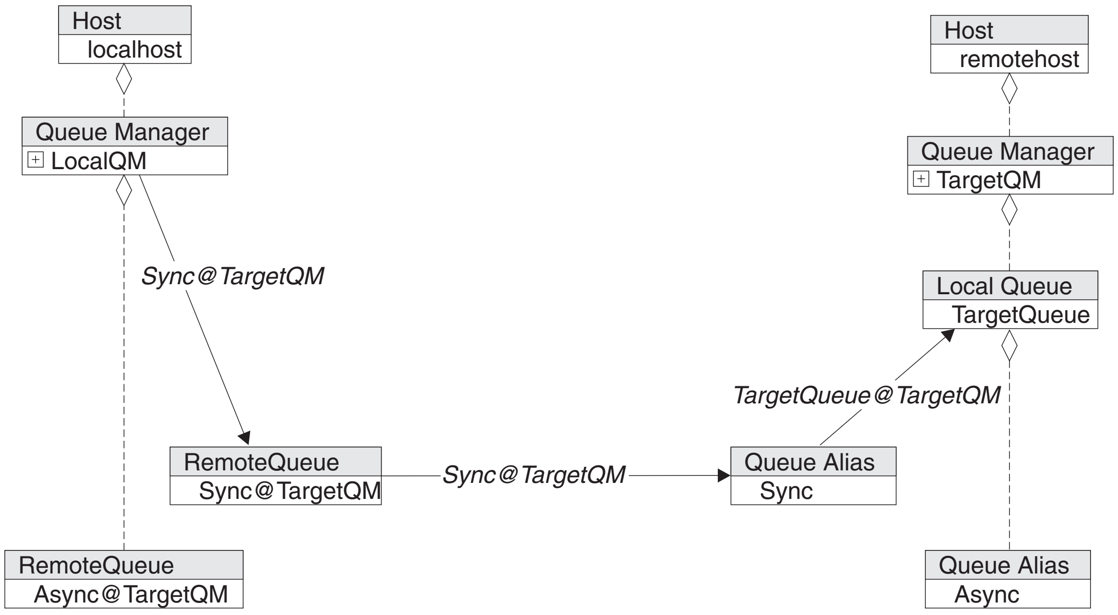


Figure 25. Resolving the synchronous route

And secondly the asynchronous resolution using AsyncAlias@TargetQM, as shown in the following diagram:

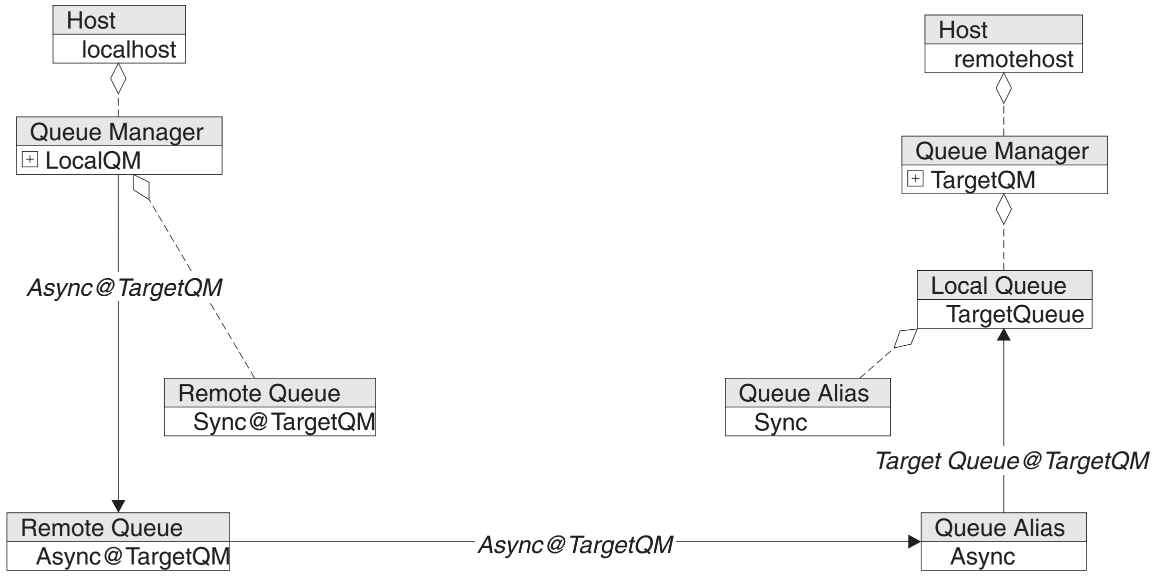


Figure 26. Resolving the asynchronous route

You could choose to view this as a pair of Push Message Routes, as shown in the following diagram:.

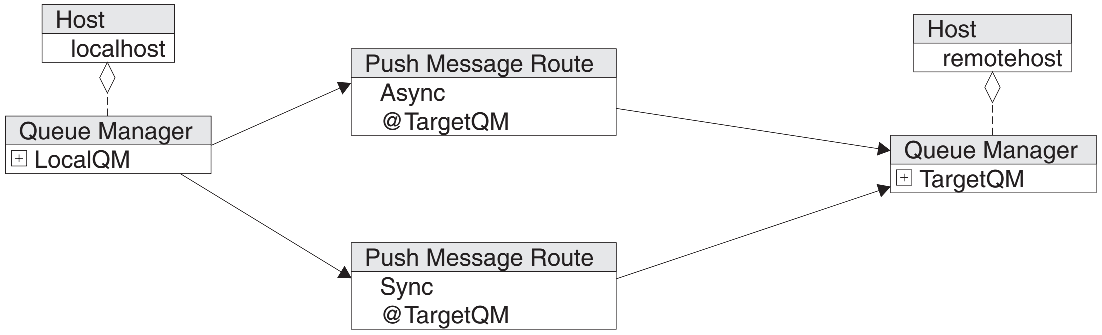


Figure 27. A pair of push message routes

Chaining remote queue references

Remote queue references can be chained together to form a longer route. This requires the use of “Via connections” on page 76.

Pushing store and forward queues

MQe has a queue type that accepts messages on a queue manager basis rather than on a queue basis. These are called Store and Forward (S&F) queues. S&F queues maintain a list of queue manager names, called Queue Manager Entries (QMEs). The S&F queue will accept messages for any queue manager represented by a QME. This acceptance is independent of the destination queue name, and so allows one queue (the S&F queue) to route all messages for a given, or several given queue managers.

S&F queues can operate in two modes, pushing mode and pulling mode. In pushing mode the messages are moved to the next queue manager just as with remote queue references. In pulling mode the messages are removed from the S&F queue by the action of a Home Server Queue. This section deals only with the pushing of messages, pulling messages with a home server queue is described in another section. A typical pushing S&F queue system might look like this:

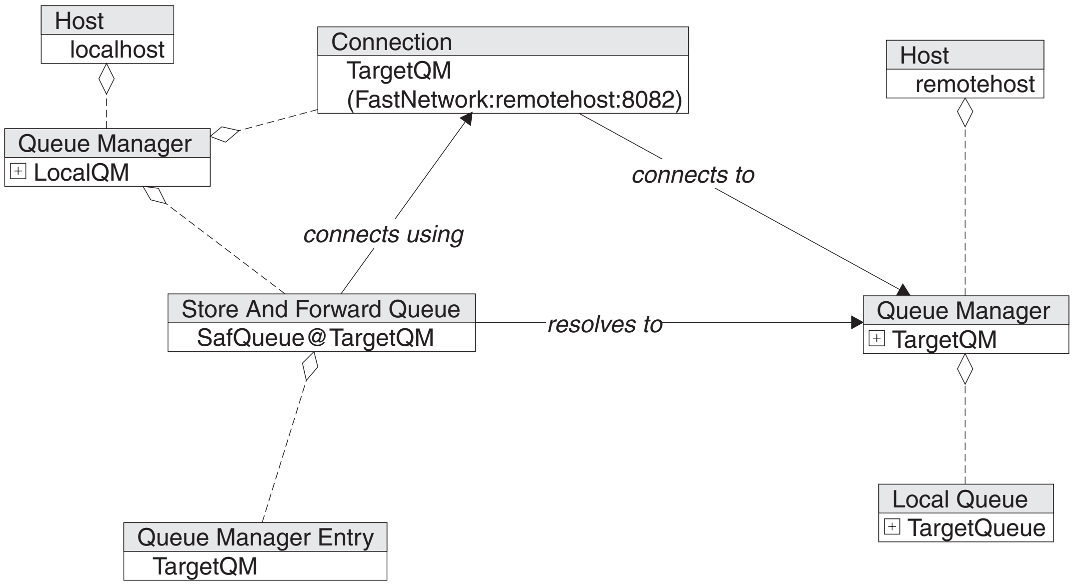


Figure 28. A typical pushing S&F queue system

A S&F queue called SafQueue has a queue manager entry (QME) for TargetQM. This allows it to accept messages for any queue on TargetQM. In common with ordinary Remote Queues, a Store and Forward queue requires a connection definition/listener pair set up in order to push messages. Unlike a normal Remote Queue Definition, a Store and Forward Queue effectively pushes to a Queue Manager rather than to a queue. The message arrives at the Queue Manager, where queue resolution is performed. When a message is put to LocalQM addressed to TargetQ@TargetQM the resolution is as follows:

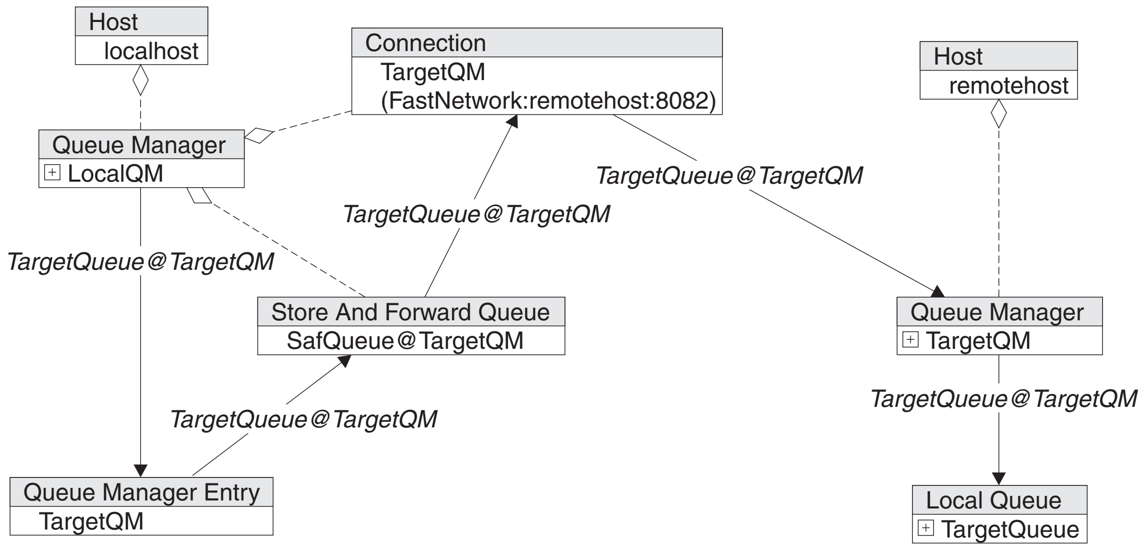


Figure 29. Routing of a message put to LocalQM and addressed to TargetQ@TargetQM

- LocalQM performs queue resolution which finds the queue manager entry TargetQM on SafQueue. LocalQM puts the message to the QME.
- Putting a message to the QME is equivalent to putting the message on the S&F queue owning the QME.
- The S&F queue performs connection resolution and finds the connection definition, and so uses it to push messages to RemoteQM.
- The queue manager then performs queue resolution and places the message on the target queue.

The Store and Forward queue forms part of a Multi Message Route. This abstract entity represents the potential for messages addressed to any queue on TargetQM, and so is called *@TargetQM, as shown in the following diagram:

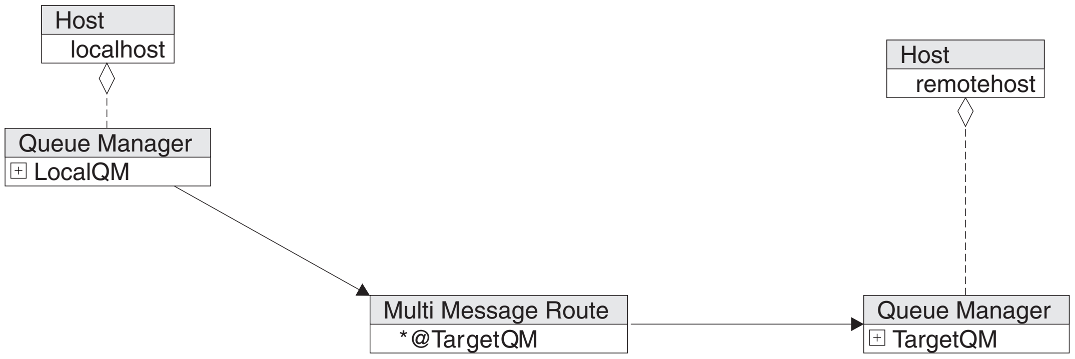


Figure 30. A multi message route

If there is no queue to which the message can be put, then it is not delivered. This prevents any further messages from being pushed from that Store and Forward queue to that Queue Manager.

S&F queues and remote queue references

Because Store and Forward (S&F) queues can accept messages for any queue on a given queue manager, they can appear to be in conflict with a remote queue reference. In such cases the remote queue reference takes precedence, because it is more specific. So if add a remote queue reference to the S&F queue resolution, the message route resolution changes immediately, and the S&F queue becomes irrelevant, as shown in the following diagram:

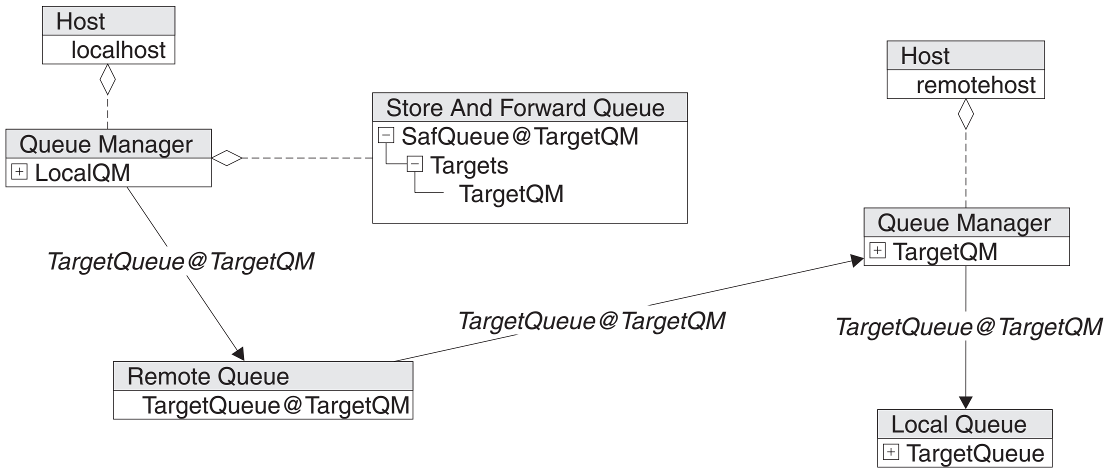


Figure 31. How routes using remote queue definitions take precedence over store-and-forward queue routes

The queue resolution finds the best (most exact) match for the message address.

So a message put to QueueAlias@TargetQM goes via the S&F queue (asynchronous transmission), but a put to TargetQueue@TargetQM goes synchronously via the remote queue reference.

Chaining S&F queues

Pushing store and forward queues can be chained together into a more complex route, as shown in the following diagram:

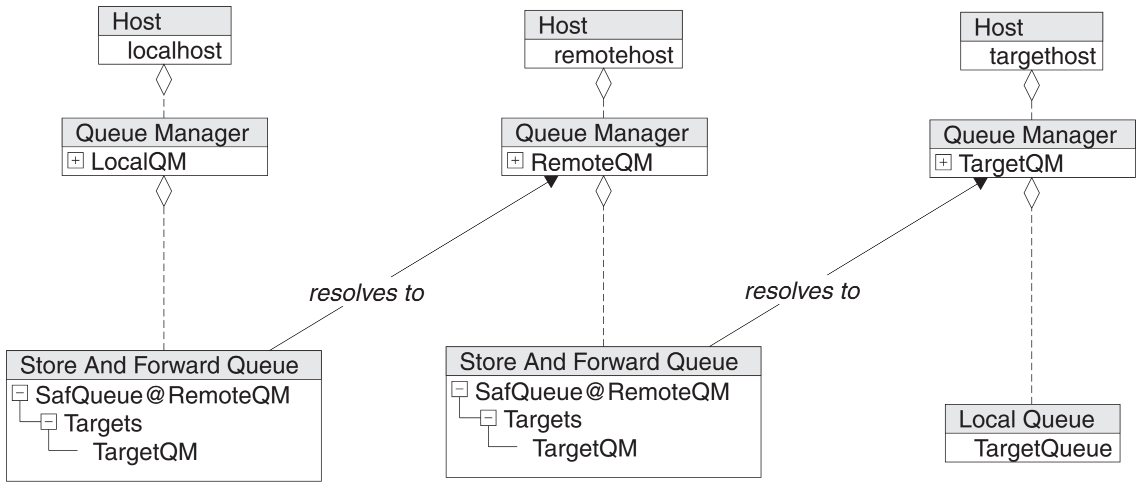


Figure 32. Pushing S&F queues chained together

The Store and Forward queue on LocalQM (SaFQueue@RemoteQM) has a Queue Manager Entry for TargetQM, but actually pushes to RemoteQM. LocalQM requires a connection definition to RemoteQM, but not to TargetQM. A message can then be transported via the intermediate S&F queue, as shown in the following diagram:

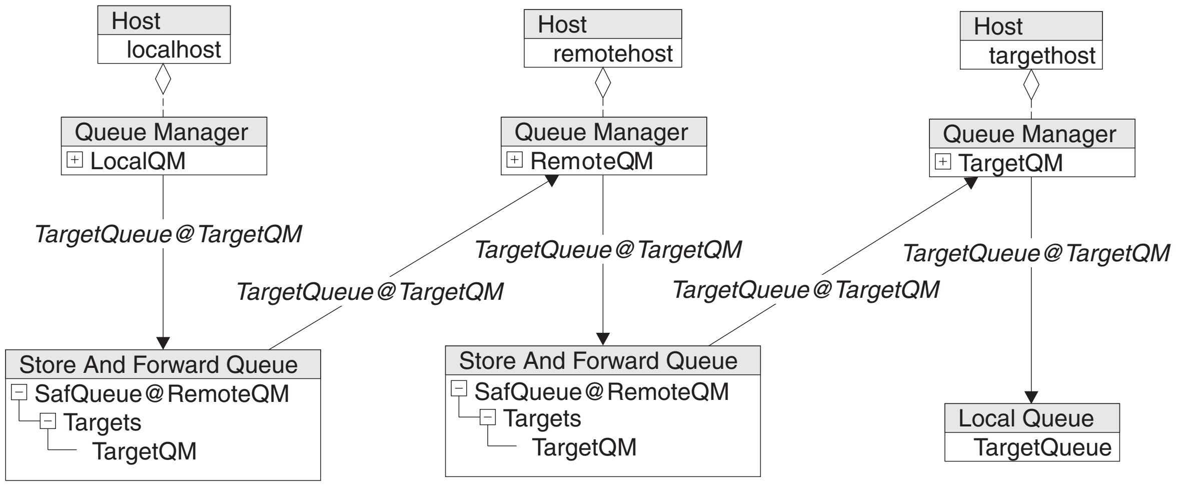


Figure 33. Transporting messages via an intermediate S&F queue

This works because the combination of queue resolution and connection resolution on LocalQM results in the message being put to the S&F queue on RemoteQM, which can then move it to its destination. The chain of Store and Forward Queues could be arbitrarily long, with each queue manager in the chain needing to know only about the next queue manager in the chain. The Message Routes express this very succinctly, as shown in the following diagram:

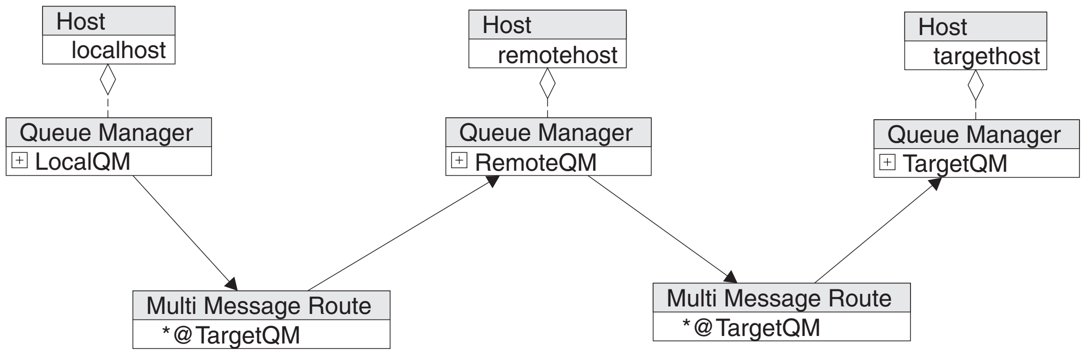


Figure 34. A chain of store and forward queues

Home server queues

Home server queues pull messages from store and forward queues. The S&F queue may be a ’pushing’ S&F queue (that is, has a valid connection definition). Home server queues only pull messages across a single ’hop’, (that is, from a remote queue manager with which it is directly connected) and only pull messages whose intended destination is the local queue manager - the queue manager upon which the home server queue resides. A typical Home Server Queue configuration is illustrated below:

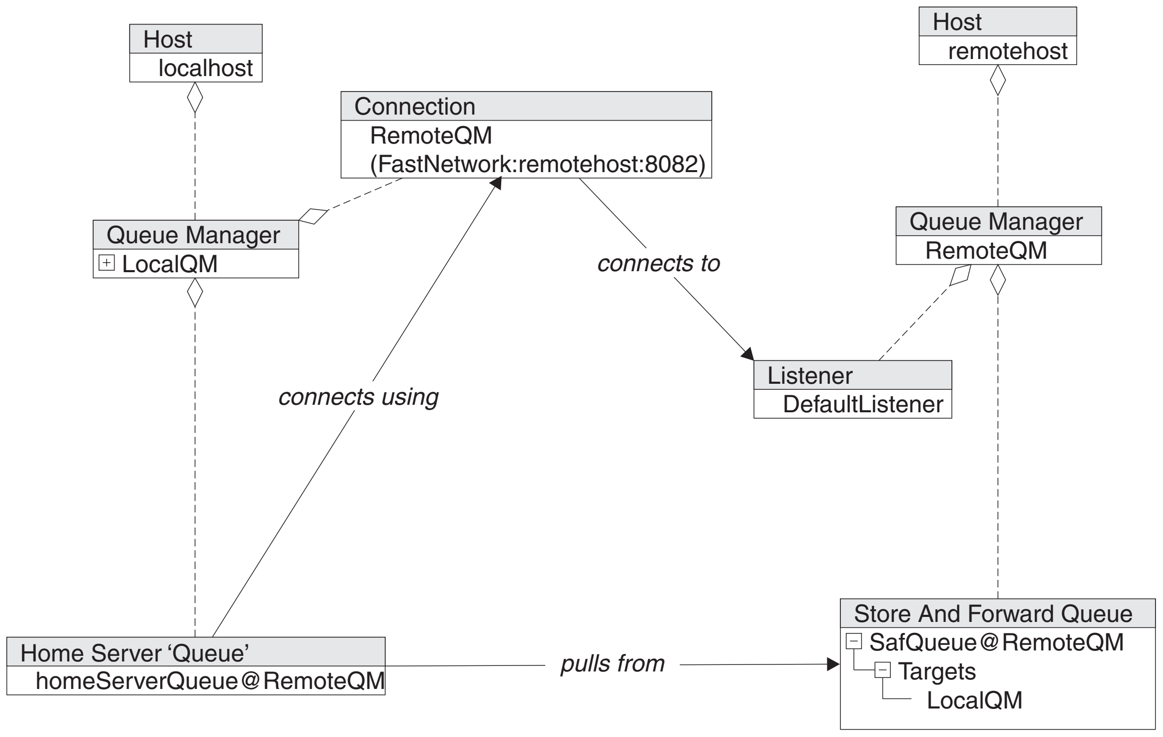


Figure 35. A home server queue configuration

The diagram shows a simple HomeServerQueue setup. In this configuration the server queue manager has no connection definition to the client; instead it has a store queue (that is, a store and forward queue with no target queue manager) that collects all messages bound for the client. This message collection embraces all queue destinations on the client.

The client pulls the messages from the store queue using a home server queue pointing at the store queue on the client. The home server queue never stores messages itself, it collects them from the store queue and delivers them to their destinations on the client. The client makes the connection request to the server using its connection definition.

The home server queue ′homeServerQueue@RemoteQM′ attempts to pull messages from the queue manager ′RemoteQM′. It requires a connection definition to be able to do this. The home server queue is able to pull messages only if there is a store and forward queue that is storing messages for LocalQM.

Messages that are pulled from RemoteQM are then ′pushed′ to local queues on LocalQM. This is shown in the following diagram, where a Home Server Queue on LocalQM is pulling messages (for LocalQM) from RemoteQM. In this case a message for TargetQueue@LocalQM is shown being pulled, and the resolution at the queue manager has been hidden for clarity. In reality, the Home Server Queue presents each pulled message to the local queue manager for resolution, as shown in the following diagram:

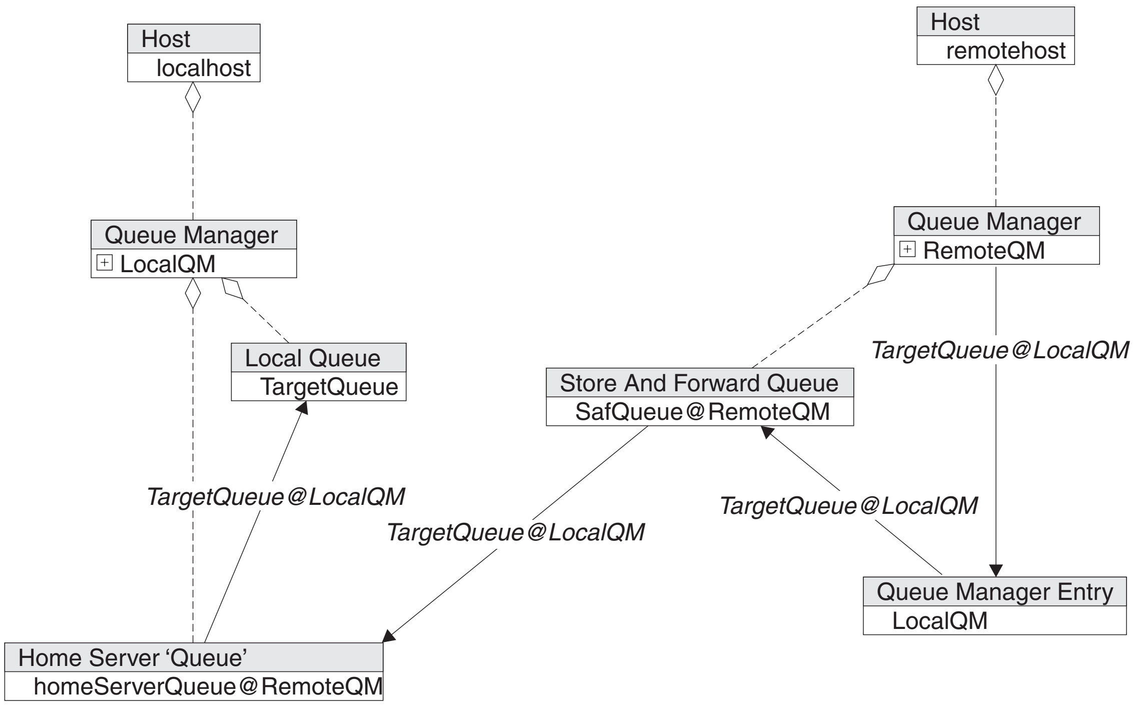


Figure 36. A home server queue pulling messages

The pull message route can be viewed at a more abstract level, as shown in the following diagram:

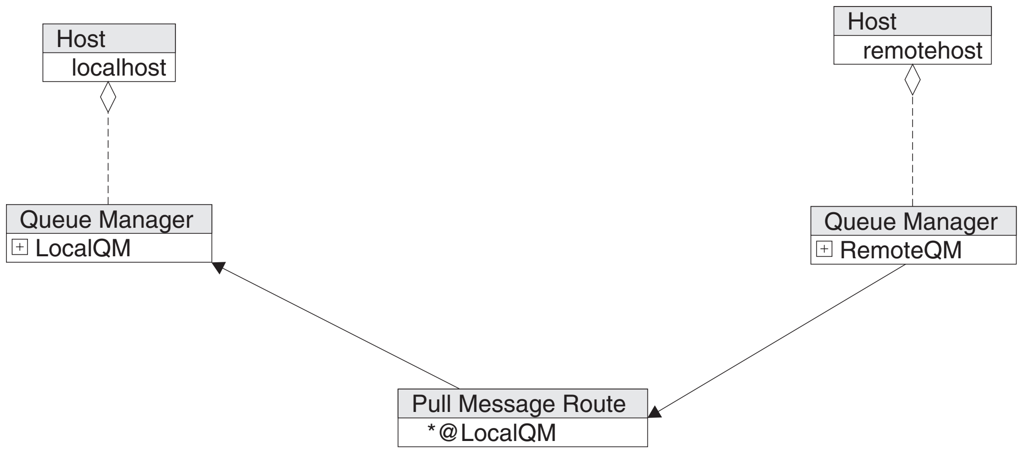


Figure 37. An abstract pull message route

How are pulled message routes useful, and where would you use them? The most important feature of a pulled message route is that the flow of messages is under the control of the local queue manager. This makes it very useful to a client that spends much of its time disconnected. If you had to rely on the server pushing message, the server would need to continuously poll the client to check if it was available. This would not be a good solution for large numbers of clients, as much of the servers time would be spent polling for disconnected clients.

Instead, with a Home Server queue, each client pulls messages when it is connected, and the server only has to deal with real requests from connected clients. One concrete example of this is the administration of queue managers that do not have listener capability. Administration messages for the client are placed upon a Store and Forward queue. The client can then use a Home Server queue to pull these when it is connected. Administration reply messages could then be pushed using normal push remote queue, as shown in the following diagram:

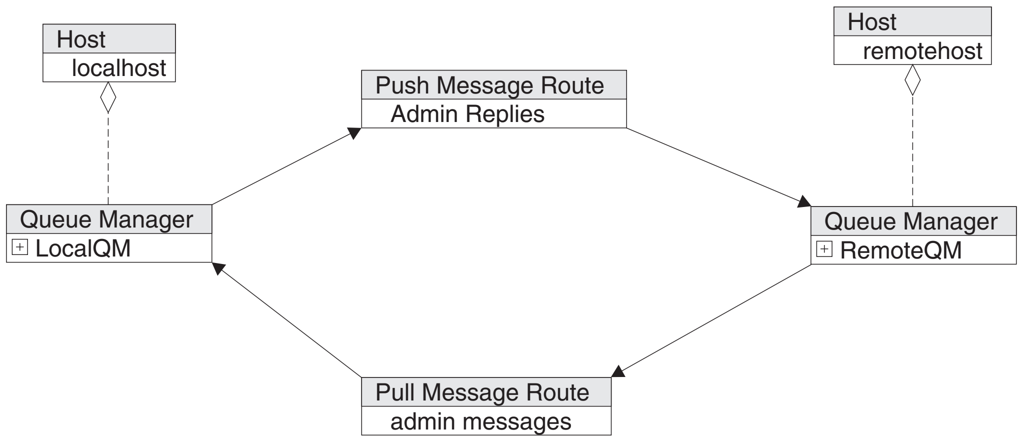


Figure 38. Administering queue managers that do not have listener capability

Via connections

Via connections allow messages to be routed via an intermediate queue manager. For example, you might want messages from LocalQM to travel to TargetQM via RemoteQM. You can already do this with 'pushing' store and forward queues, but via connections provide another mechanism, as shown in the following diagram:

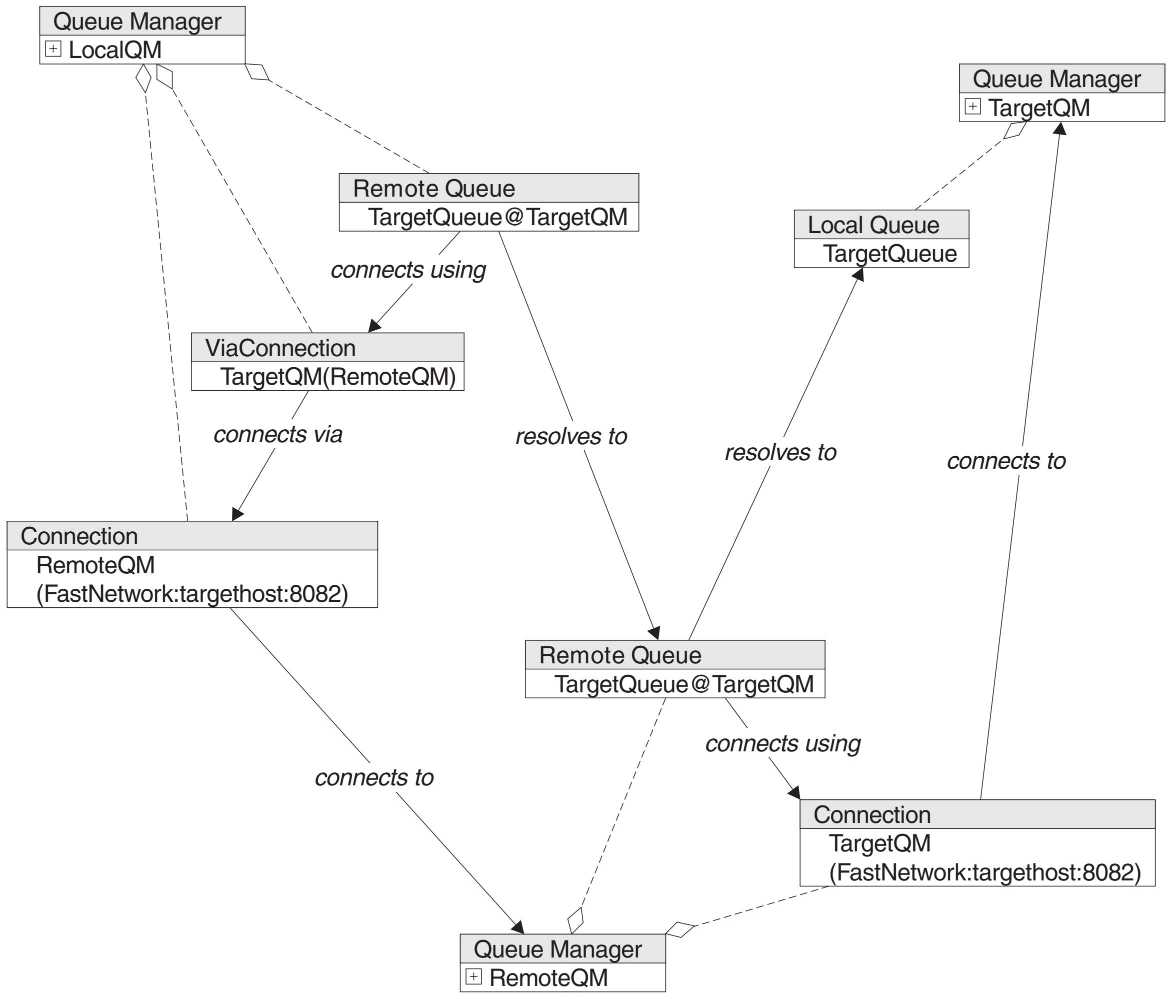


Figure 39. Via connections

The diagram above illustrates the components being used. The connection definition called 'TargetQM' on LocalQM does not contain the address of TargetQM, but simply refers to the connection definition called 'RemoteQM'. This means that any messages destined for TargetQM will be sent to RemoteQM, and RemoteQM will be able to move the messages onward. In the diagram above, RemoteQM has the necessary connection to move the message to TargetQM.

The message flows as expected, as shown in the following diagram:

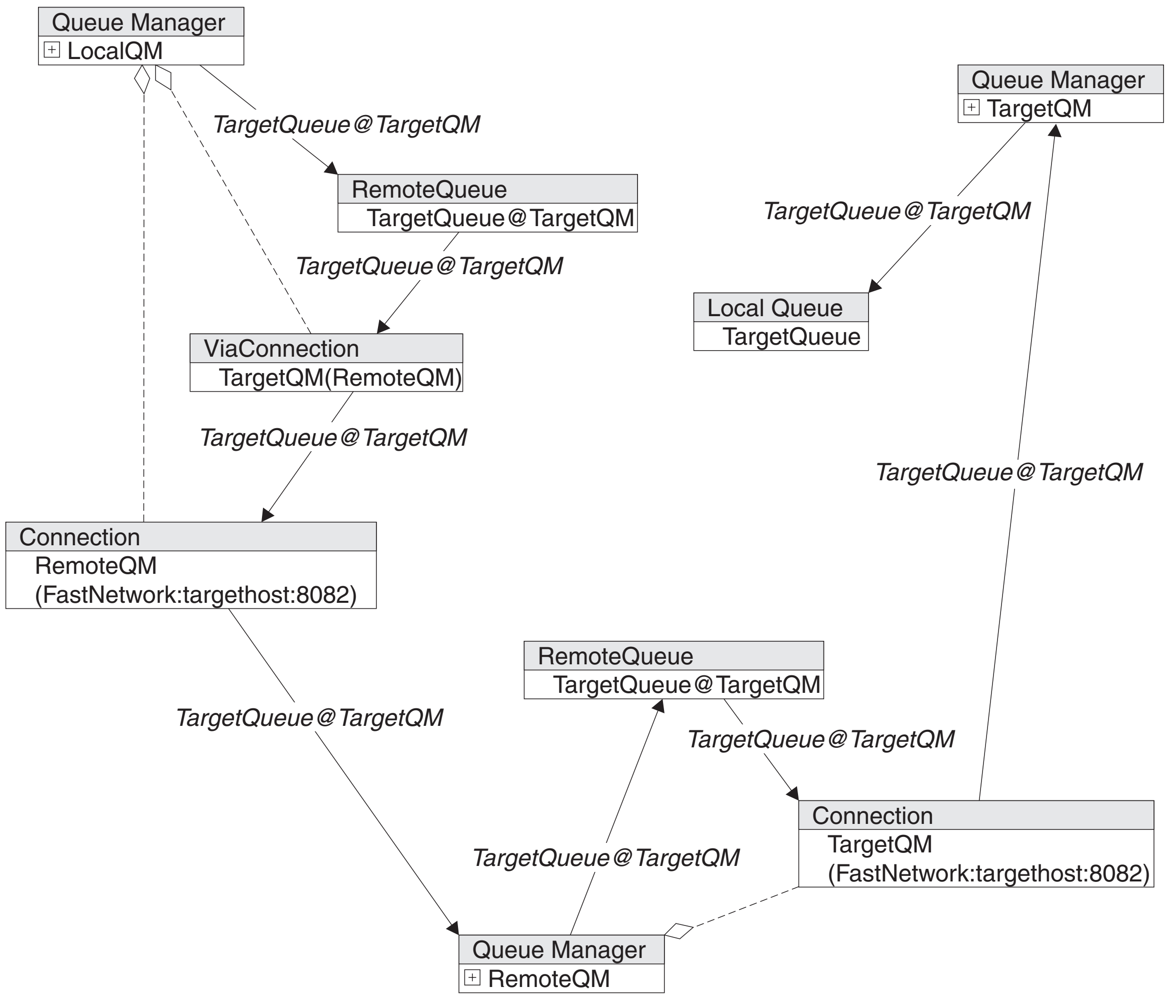


Figure 40. Message flow using a via connection

The Remote Queue on LocalQM uses Connection Resolution to find the Via Connection. This then passes the message on to the real connection which moves the message to RemoteQM. On RemoteQM queue resolution proceeds as for the simple case.

You can see the topology most clearly using Message Routes, as shown in the following diagram:

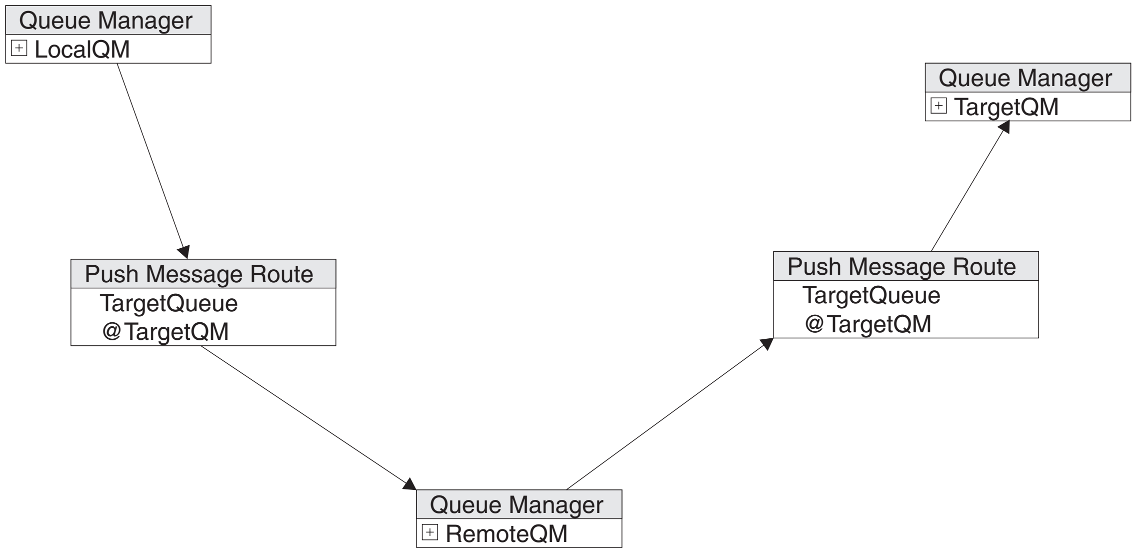


Figure 41. Via connections expressed using message route schema

This is known as ’chaining remote queues’. The central remote queue can be synchronous, asynchronous, or even a store and forward queue.

Rerouting with queue manager aliases

Fail-over is a common situation that illustrates the important part that Queue Manager Aliases play in routing.

In the following examples, you can see a client communicating with a server, and a have a backup server that can be used if the main server fails, or is taken down for maintenance:

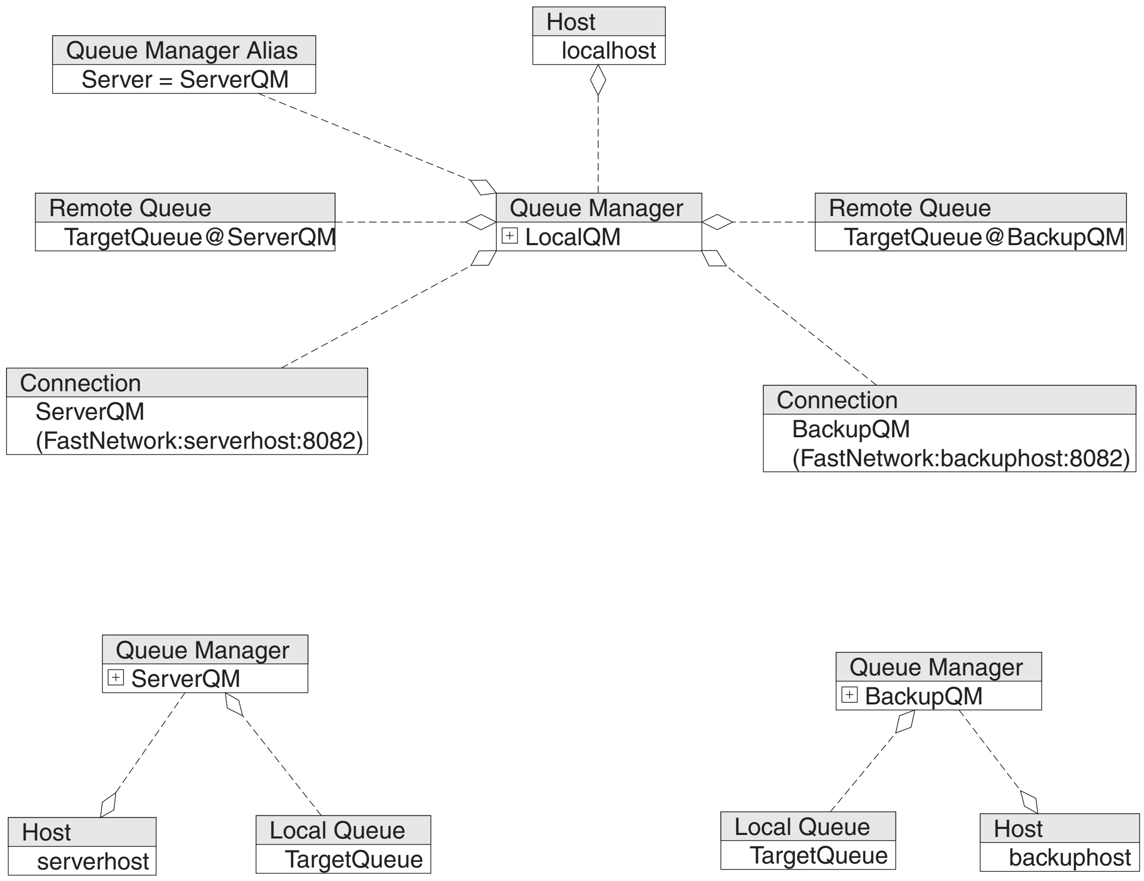


Figure 42. Queue manager aliases and fail-over.

The diagram above shows the local client queue manager, with a connection to ServerQM and a remote queue definition for TargetQueue@ServerQM. The server (bottom left) has a local queue as the target for the example message, and this is mimicked by the backup server (bottom right). Additionally, on the client queue manager, there is a Queue Manager Alias mapping the name Server to ServerQM. This mapping is then used for messages put to the server. The message resolution is shown below for the normal operating configuration, where a message put to TargetQueue@Server is directed to TargetQueue@ServerQM:

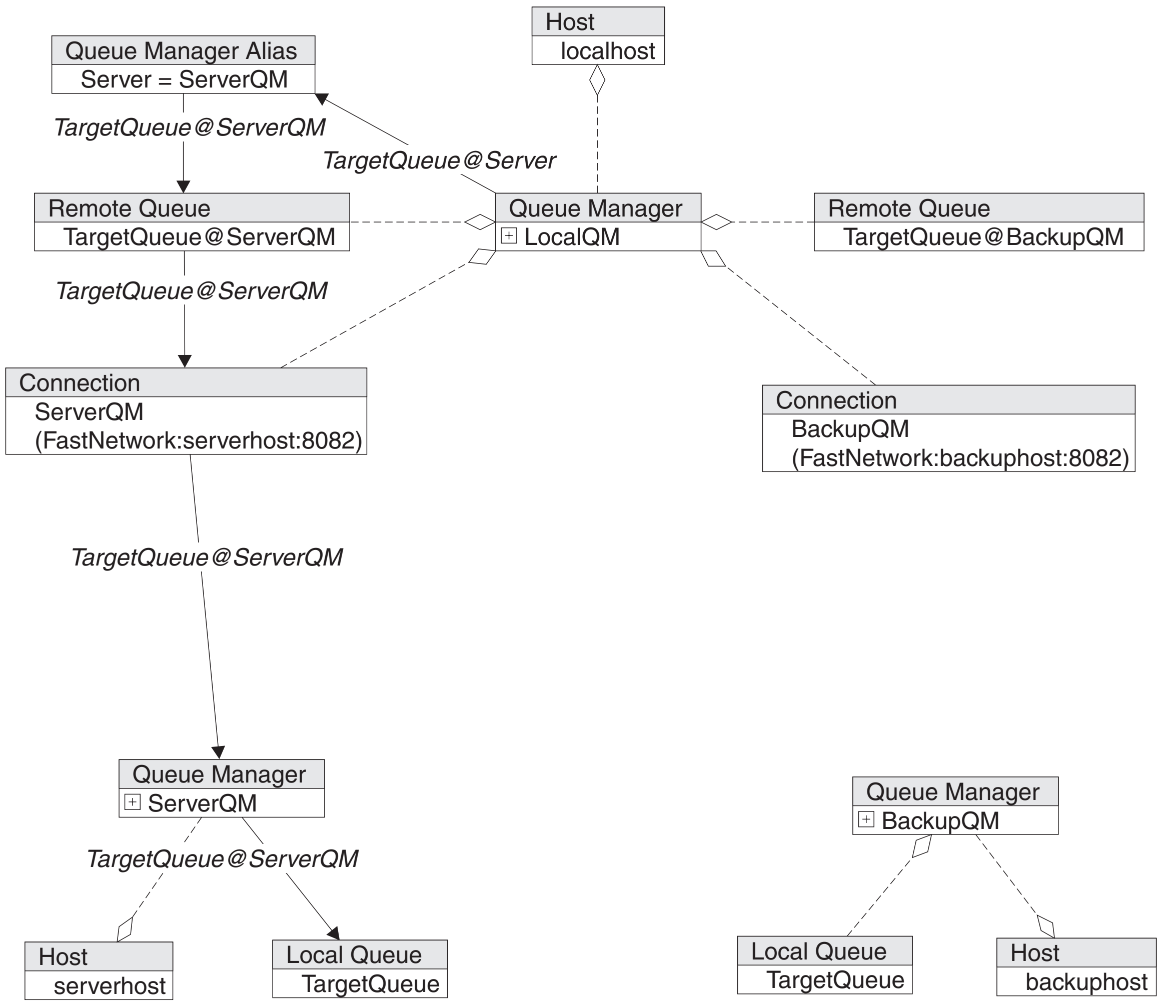


Figure 43. Routing traffic using a "server" alias

The alias maps messages for Server to ServerQM, and this selects the remote queue definition TargetQueue@ServerQM. If the network administrator needs to route traffic to the backup server, only the Queue Manager Alias needs to be changed (it is in fact deleted, and recreated with a different target name, in this case BackupQM):

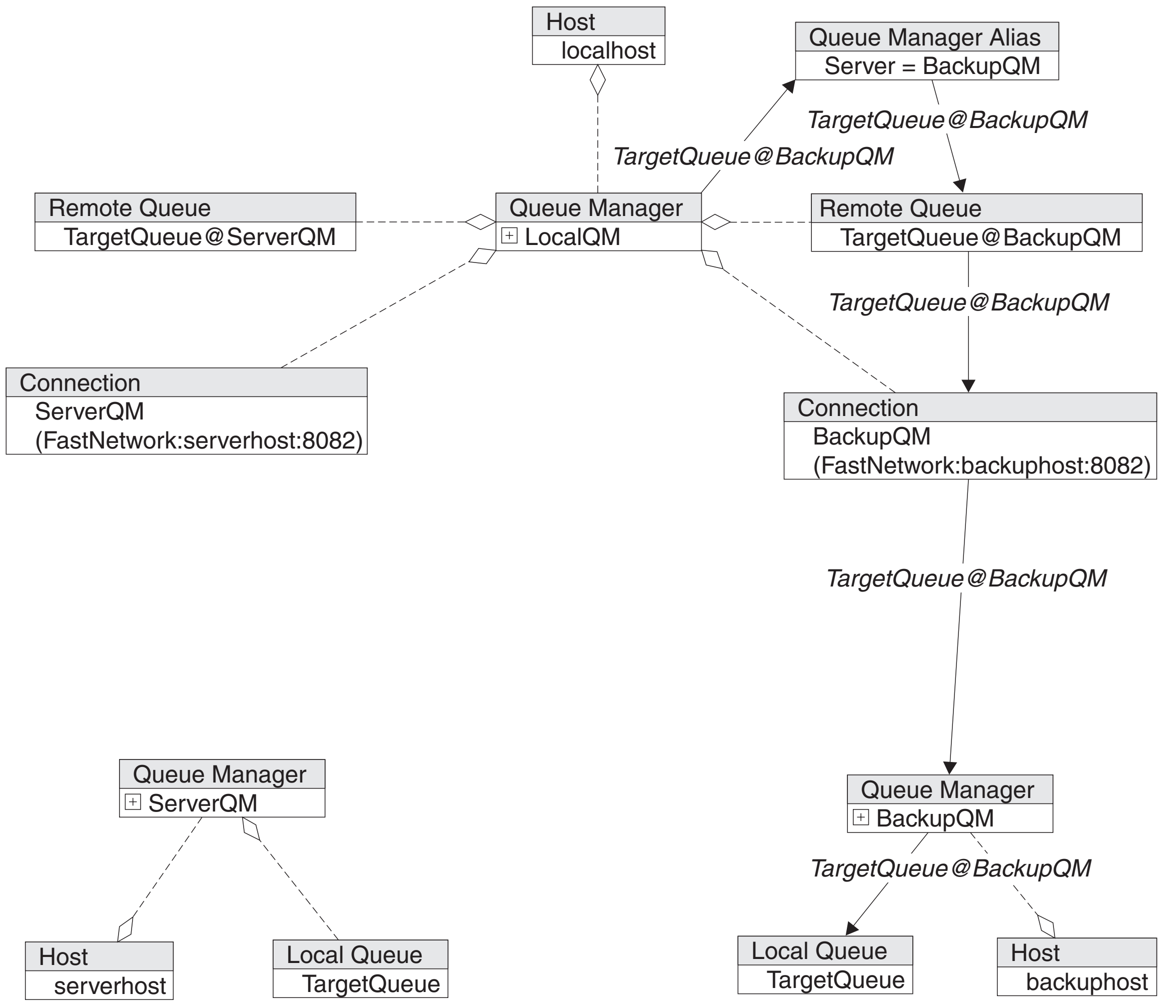


Figure 44. Routing traffic to the backup server, using a "server" alias

The change of alias reroutes the message to a different remote queue, and hence on to the backup queue manager and to TargetQueue@BackupQM. There is a pair of message routes, one to each server, and a Queue Manager Alias to choose between the message routes, as shown in the following diagram:

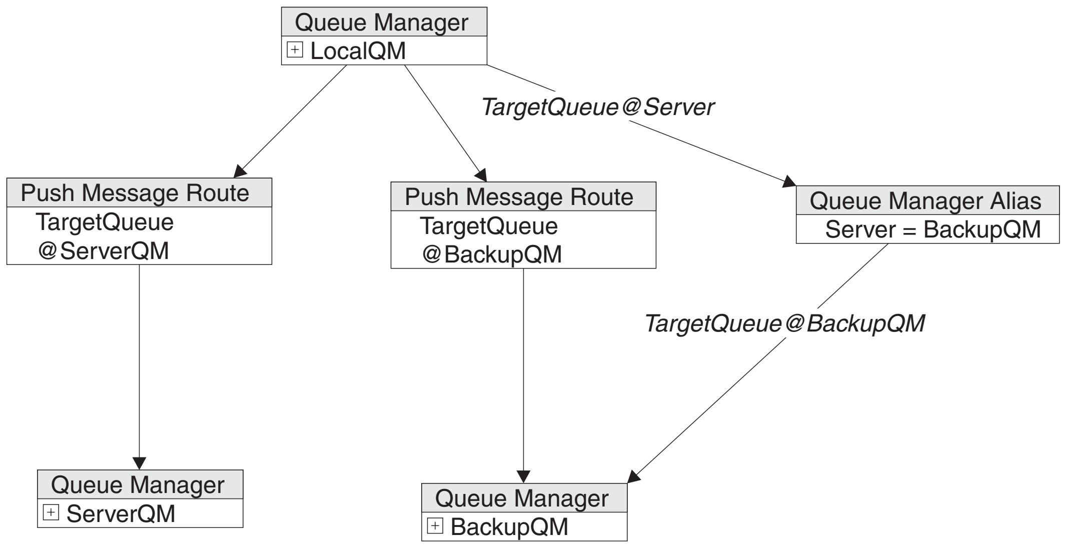


Figure 45. Choosing between message routes

The example above required a change to every client on a system that requires rerouting to a backup server. If there are a large number of clients this might be impractical. In addition, each client requires two complete message route definitions (a remote queue and a connection definition for each). You can avoid the need to change the client by having a second server ready to listen on the same address and port as the first. When the administrator wants to change over the first can be brought down, and the second can change over. In this situation it might be useful to keep the names of the servers different. The backup server can be given a Queue Manager Alias mapping BackupQM to ServerQM. This allows BackupQM to take the place of ServerQM.

MQe-MQ bridge message resolution

A connection between MQe and MQ queue managers involves a collection of objects. The following diagram shows only the entities that form the communications link between the two queue managers:

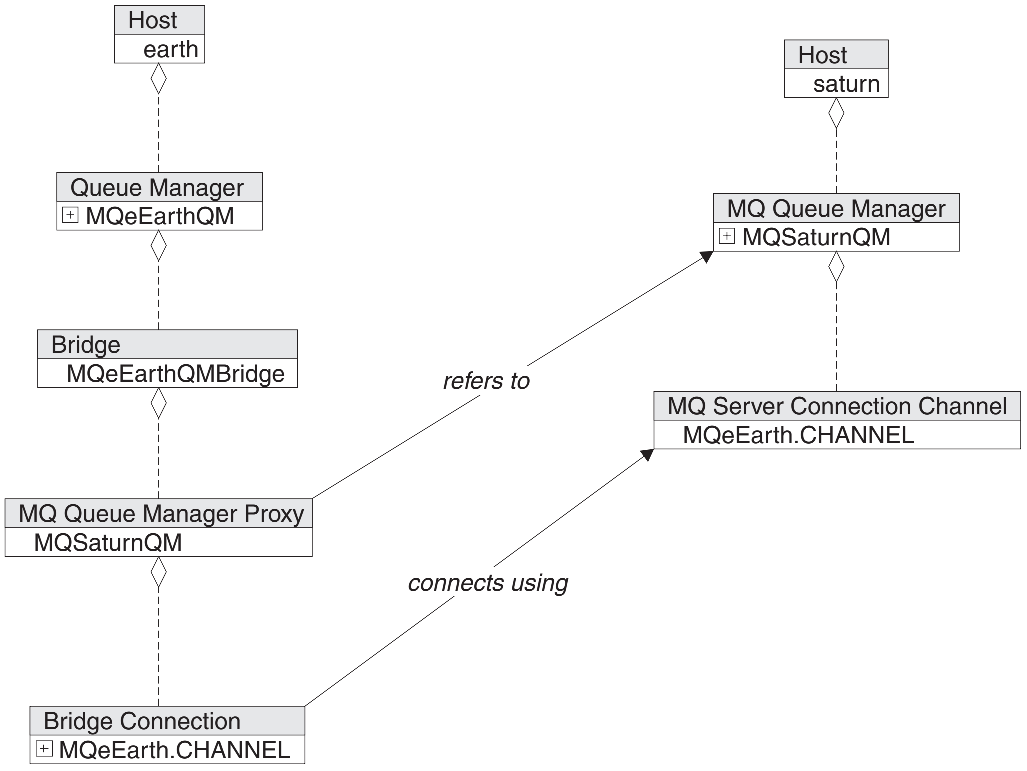


Figure 46. Connecting MQe and MQ queue managers.

The important entities are:

- (Bridge)MQeEarthQMBridge - a bridge resource owned and controlled by the MQeEarthQM queue manager.
- (MQ Queue Manager Proxy)MQSaturnQM - describes MQSaturnQM and how to connect to it.
- (BridgeConnection)MQeEarth.CHANNEL - a communications path between MQeEarthQM and MQSaturnQM.
- (MQ Server Connection Channel) MQeEarth.CHANNEL - a standard MQ server channel providing an entry point to MQSaturnQM for MQeEarthQM.

These entities are described in more details in other parts of this documentation. These entities are used in the following examples of bridge connectivity, but are not shown in the diagrams.

Pulling messages from MQ

By setting up a Transmit queue on MQ, and a bridge listener on an MQe queue manager, you can enable the queue manager to pull messages from the transmit queue. Although in theory this is sufficient to pull messages from the transmission queue, you cannot place messages onto the transmission queue without creating extra queues on an MQ queue manager.

Single pull route:

To allow the messages to be correctly routed, you can create extra queues on an MQ queue manager. The simplest form is to create a remote queue on MQ to allow messages addressed to TargetQueue@MQeEarthQM to be accepted by the MQ queue manager, as shown in the following diagram:

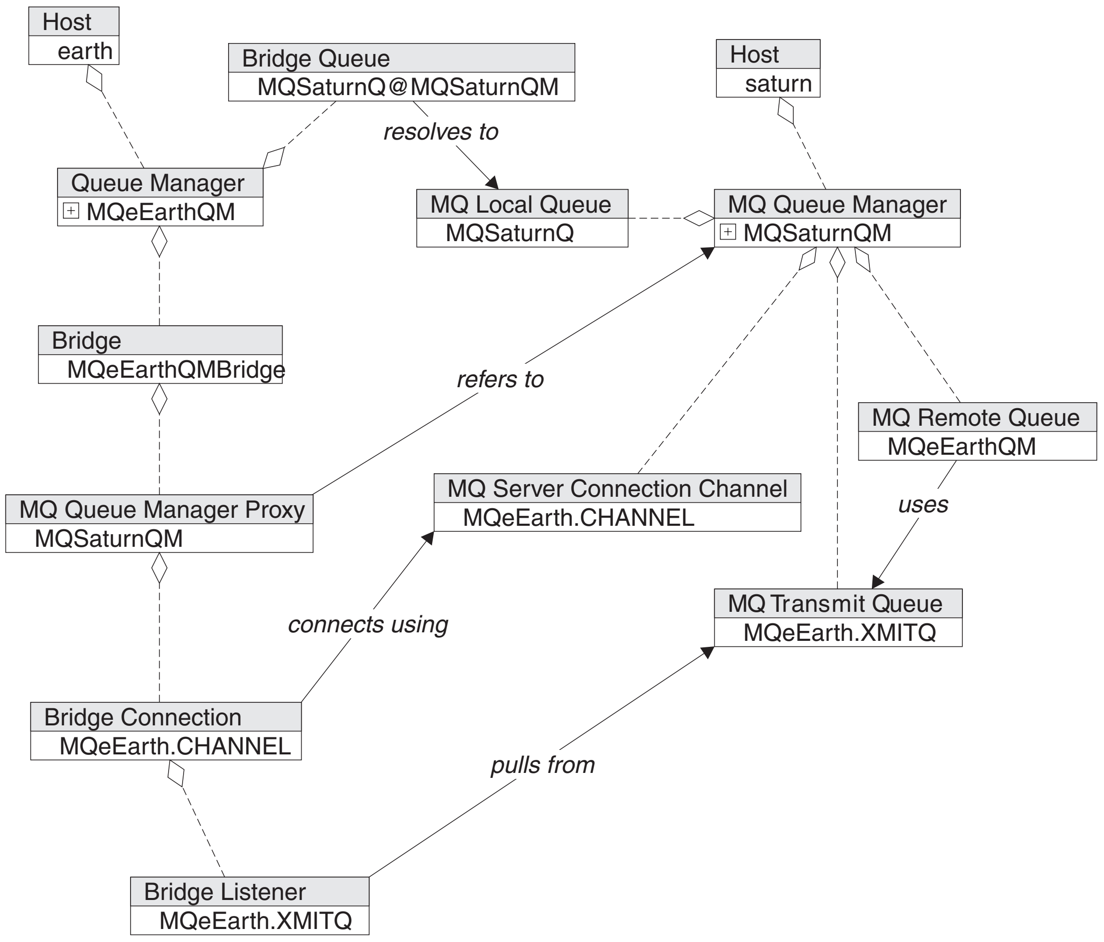


Figure 47. Creating a remote queue on MQ

Messages addressed to TargetQueue@MQeEarthQM are placed upon the MQ Transmit queue. The bridge listener then pulls them from the transmit queue and presents them to the MQe queue manager. Message resolution then takes place, as shown in the following diagram:

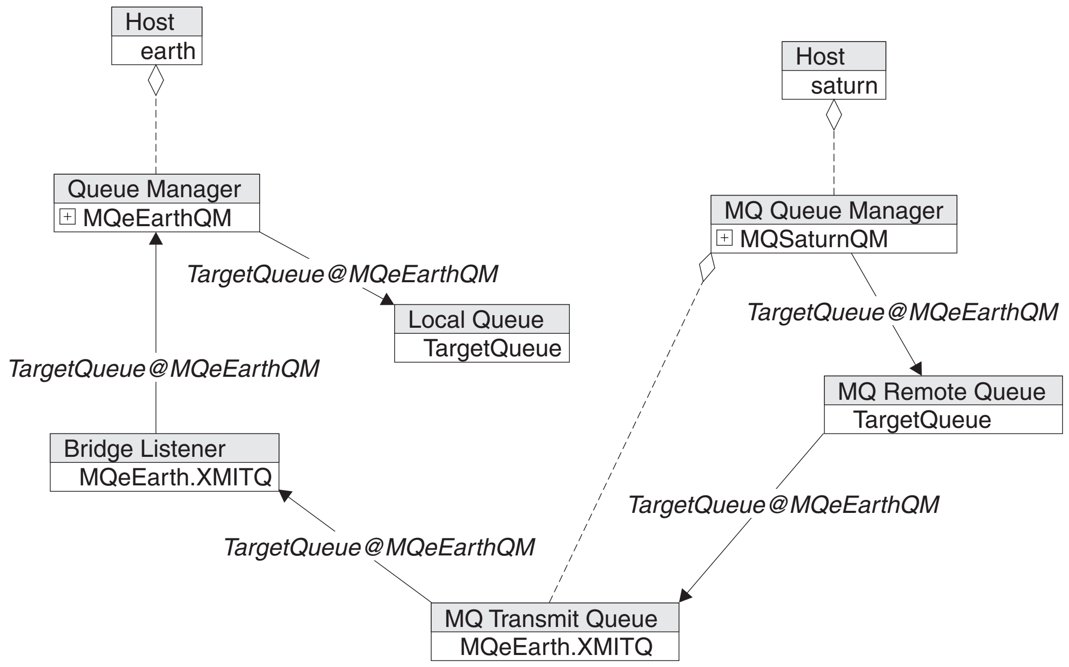


Figure 48. Bridge listener pulling from an MQe transmit queue

This is effectively a single pull message route:

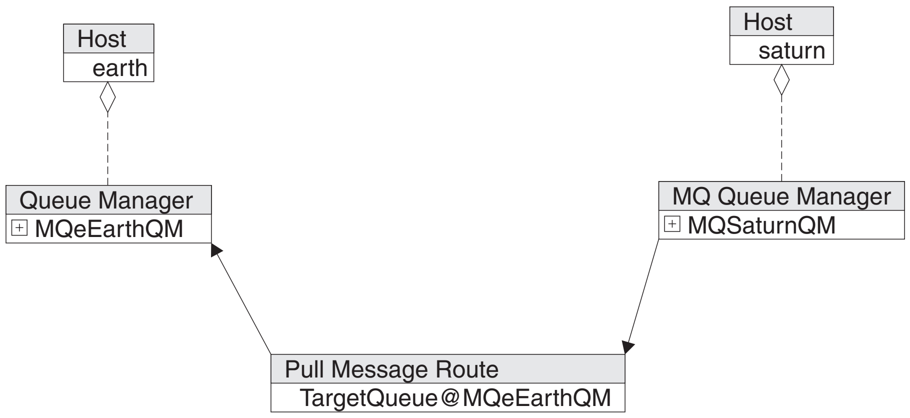


Figure 49. A single pull message route

Multiple pull route:

It is generally more efficient to use a multiple pull message route as this requires the same number of resource definitions, but will handle all the traffic for the MQe queue manager. This is done using a Remote queue manager alias on MQ (effectively a remote queue where the target queue name is the same as the target queue manager name), as shown in the following diagram:

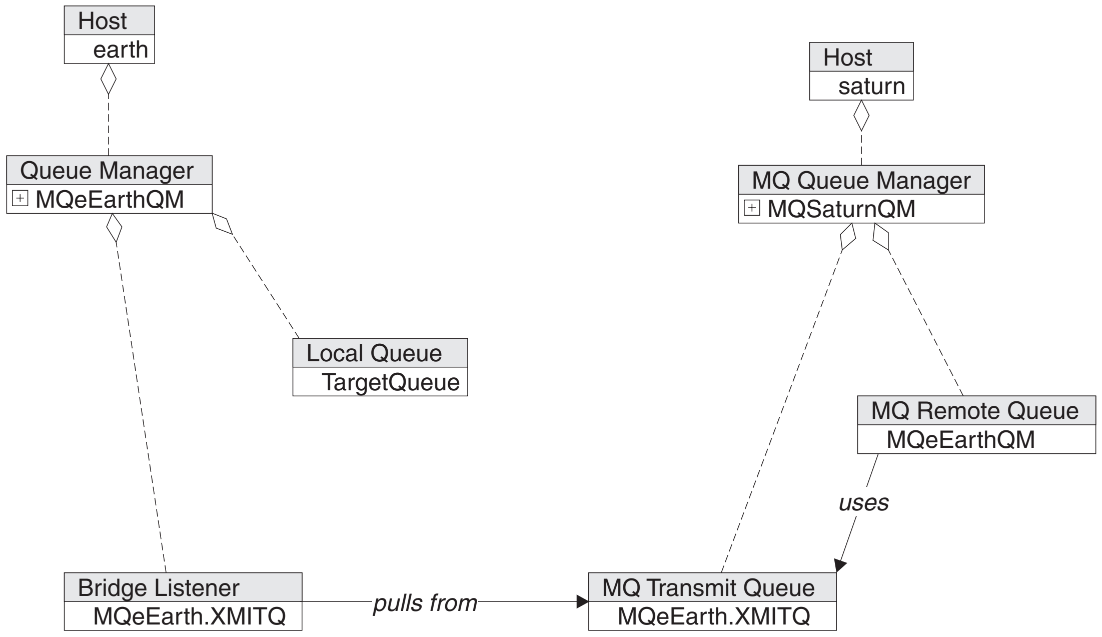


Figure 50. A multiple pull message route

Message resolution works as before, but now messages for any queue on MQeEarthQM will be moved, making this a multiple pull message route, as shown in the following diagram:

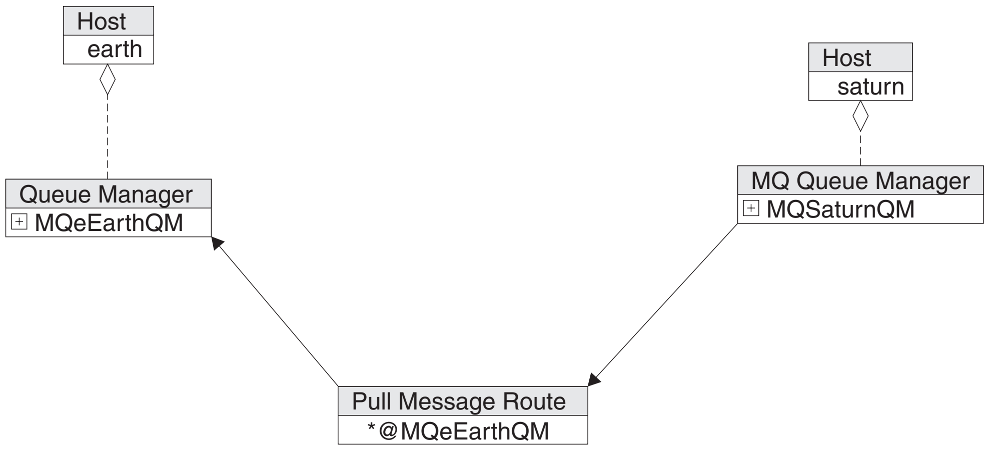


Figure 51. Multiple pull route, expressed using message route schema

Pushing messages to MQ

Pushing messages to MQ is quite straightforward. Again you need to presume the presence of the common components described in "MQe-MQ bridge message resolution" on page 83, but now you need to create a Bridge Queue which is an MQe Remote queue that refers to a queue on an MQ queue manager, as shown in the following diagram:

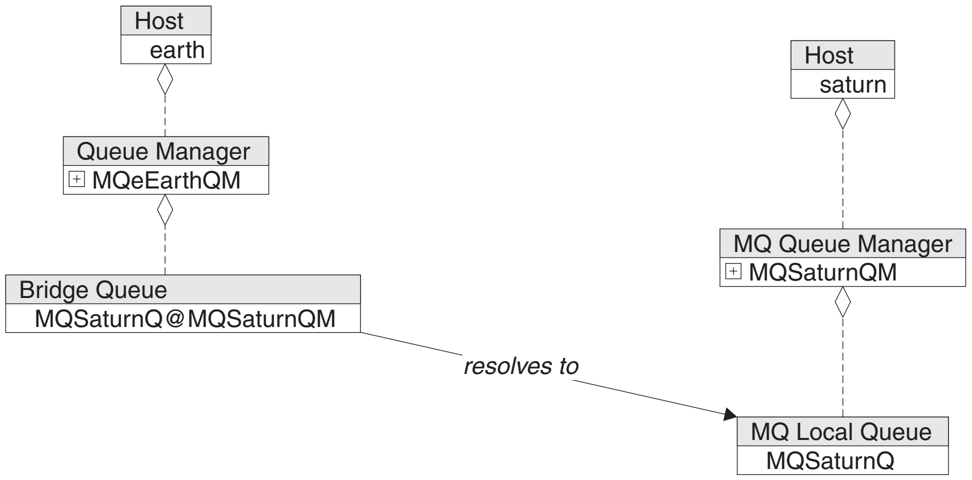


Figure 52. Pushing messages to MQ

Messages travel as expected across this remote queue definition, as shown below:

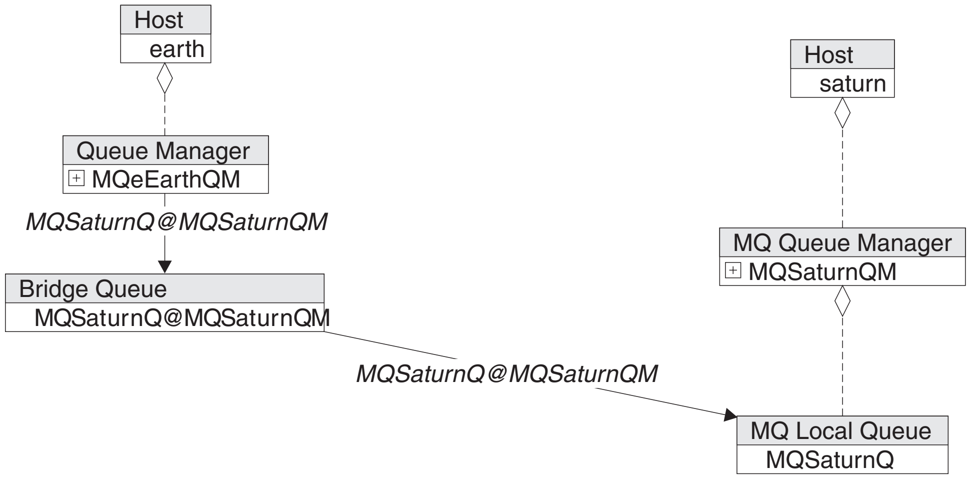


Figure 53. Messages travelling across a remote queue definition

This is exactly the same as a simple push message route between two queue managers, as shown below:

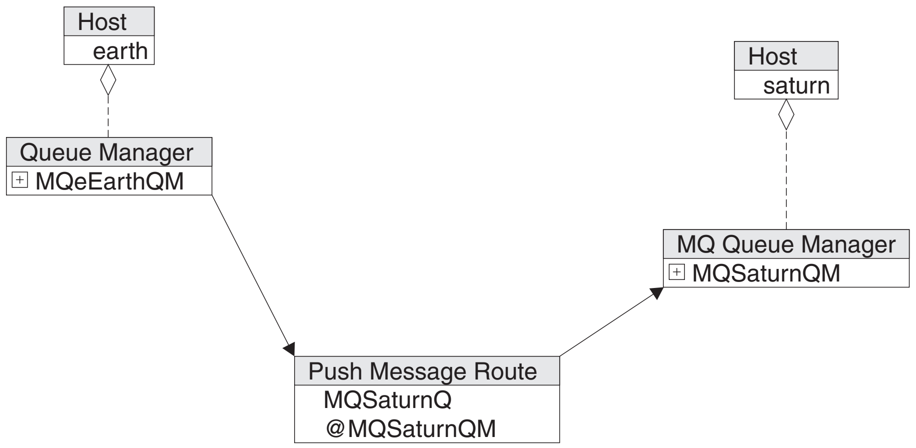


Figure 54. Simplified view of route pushing messages to MQ

Connecting a client to MQ via a bridge

A common topology is to allow messages to flow between MQ and a client MQe queue manager. This cannot happen directly, but requires an intermediate bridge-enabled MQeQueue manager. The client can then be a small footprint device with no knowledge of MQ. Additions are needed to allow a client (MQeMoonQM, on a device called moon) to communicate with MQ, as shown in the following diagram:

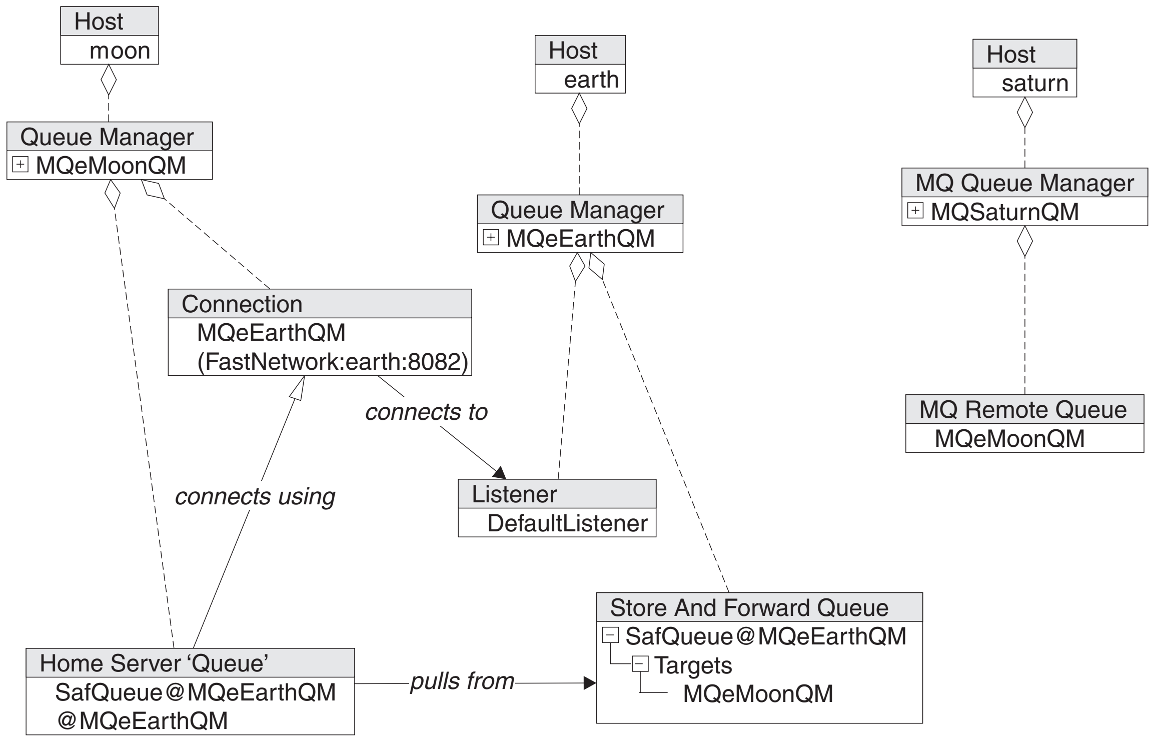


Figure 55. A client communicating with MQ

This adds the following:

- (Host)moon
- (QueueManager) MQeMoonQM on (Host)moon

- A connection definition from MQeMoonQM to a matching listener on MQeEarthQM to provide the connectivity between the two MQe queue managers.
- A store and forward queue on MQeEarthQM that accepts and holds messages for MQeMoonQM, and a home server queue on MQeMoonQM that pulls messages from the store and forward queue.
- A remote queue definition on the MQ queue manager that routes messages for MQeMoonQM to the transmission queue MQeEarth.XMITQ. This allows messages for MqeMoonQM to be placed on the transmission queue, from where they are pulled to MQeEarthQM.

The topology is more readily seen as message routes, as shown in the following diagram:

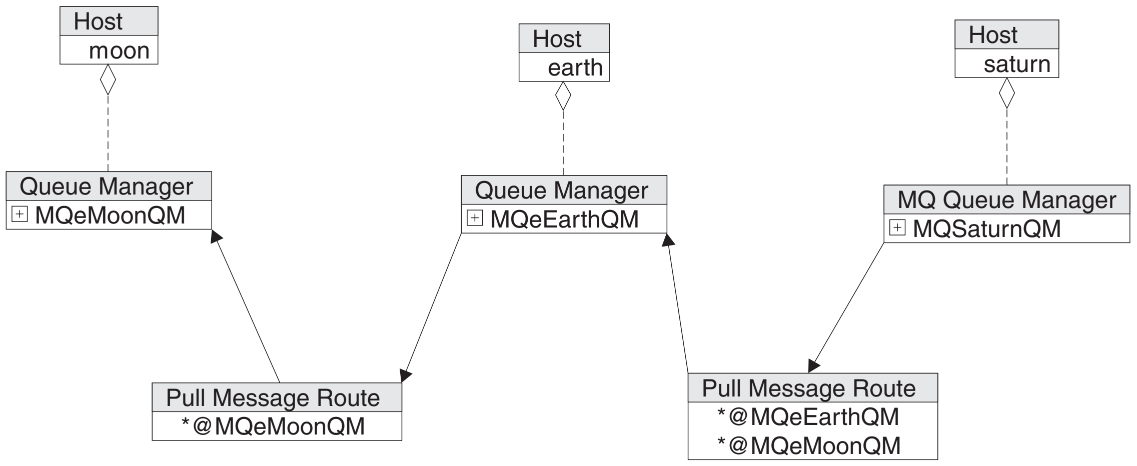


Figure 56. Simplified pull routes from MQ through an MQe gateway to an MQe device style queue manager

Messages can be pushed to MQ by using a via connection to chain remote queues, as shown below:

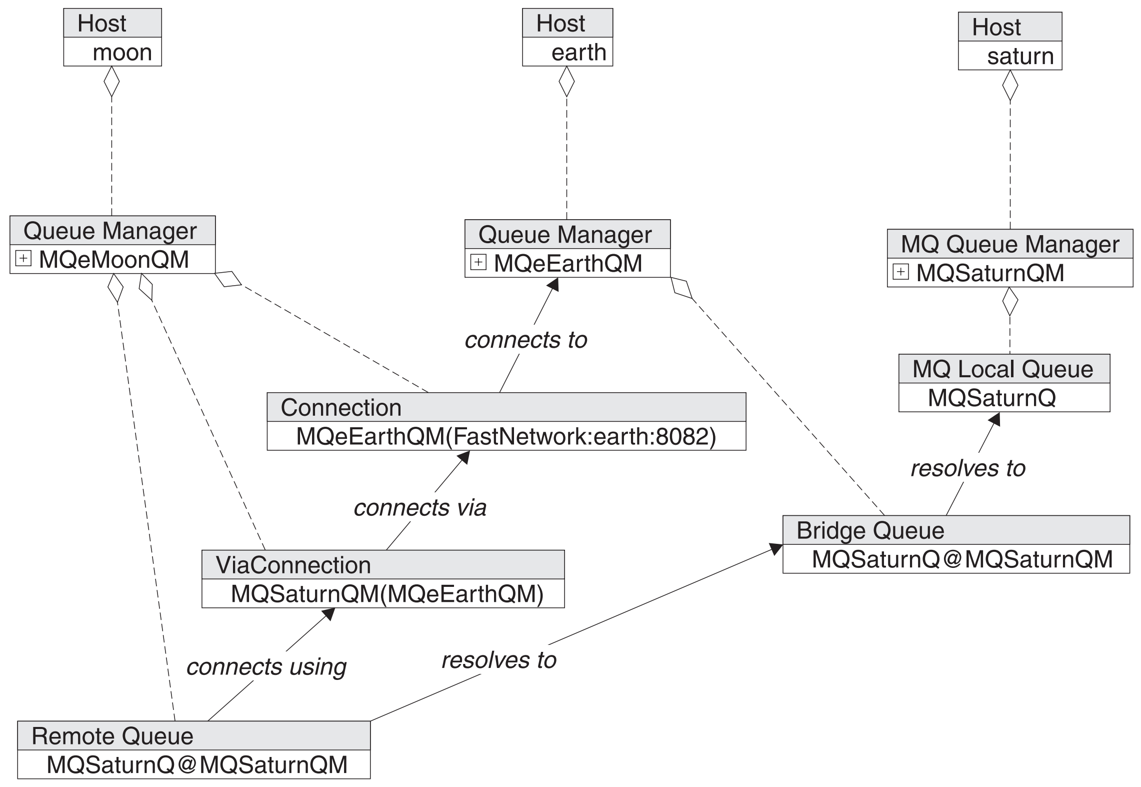


Figure 57. Pushing messages using a via connection

Here a via connection has been added to route messages destined for MQSaturnQM vian MQeEarthQM, and a remote queue definition for MQSaturnQ@MQSaturnQM has been added. The messages can now flow from the client to MQ, as shown in the following diagram:

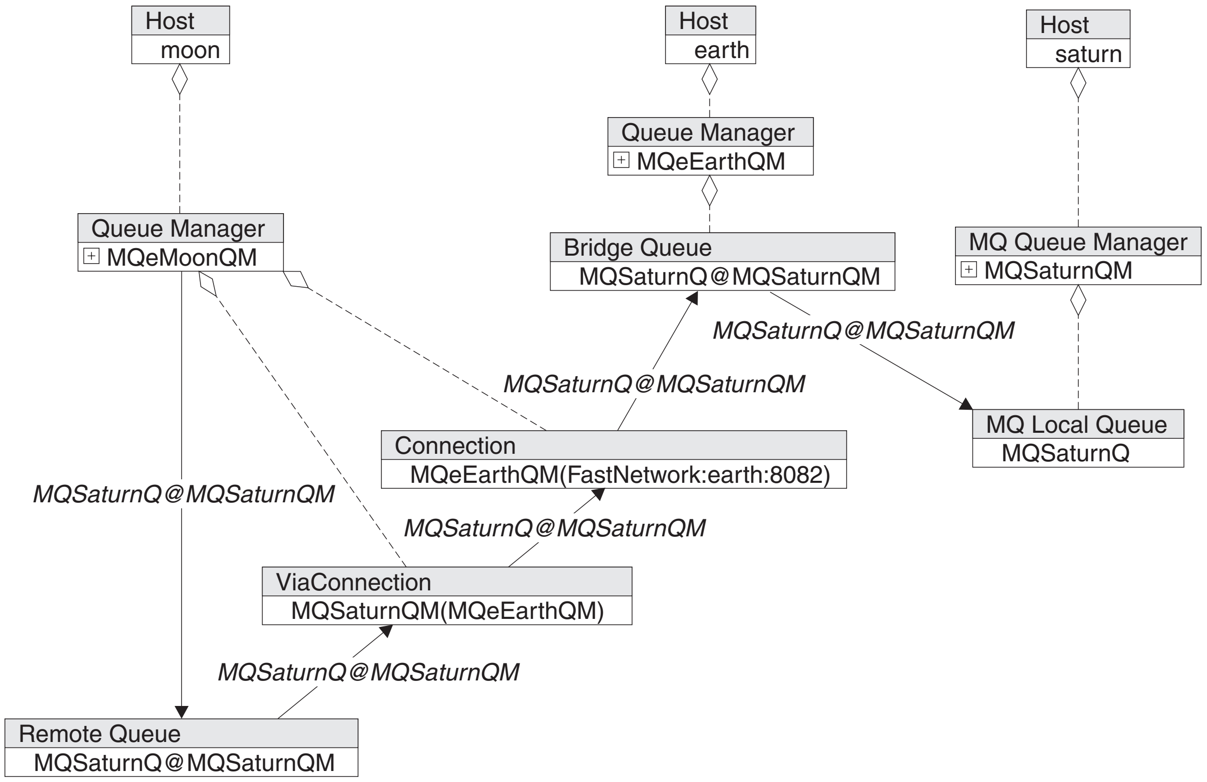


Figure 58. Pushing messages to MQ

This topology is more easily understood as a collection of message routes, as follows:

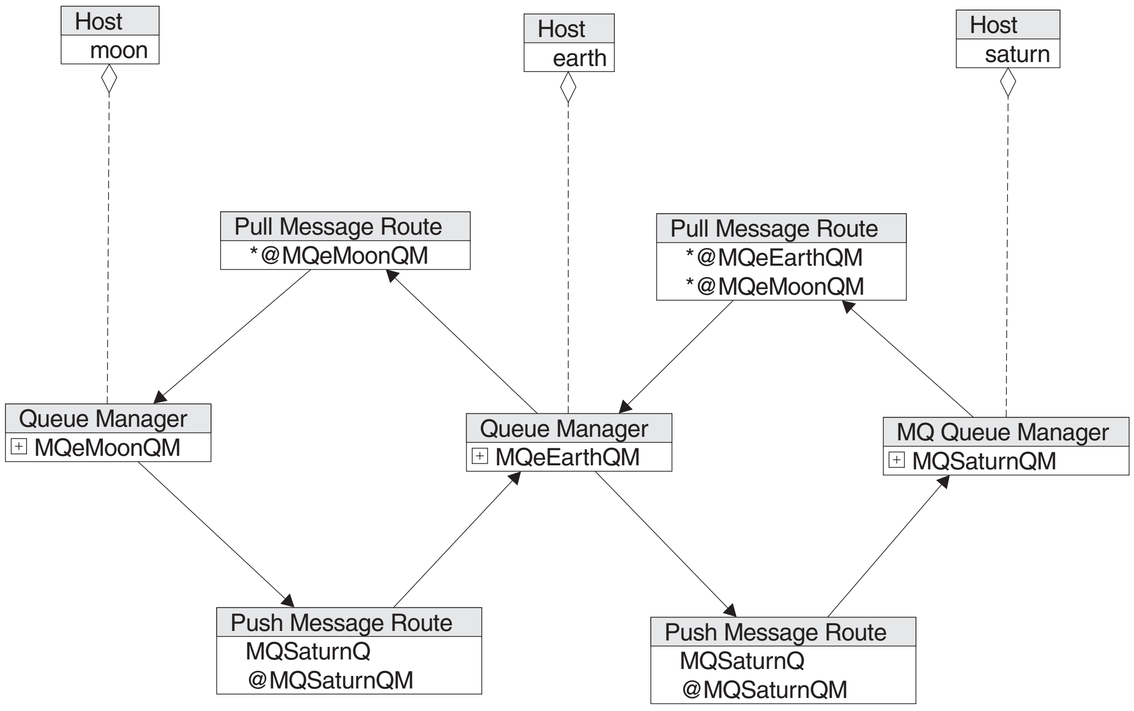


Figure 59. Simplified view showing routes which push messages from a device style MQe queue manager to an MQ queue manager

Pushing messages to MQ with a via connection

A common topology allows messages to flow between MQ and a client MQe queue manager. This cannot happen directly, but requires an intermediate bridge-enabled MQeQueue manager. The client can then be a small footprint device with no knowledge of MQ. If you start with the configuration we have above, the following additions are needed to allow a client (MQeMoonQM, on a device called moon) to communicate with MQ, as shown in the following diagram:

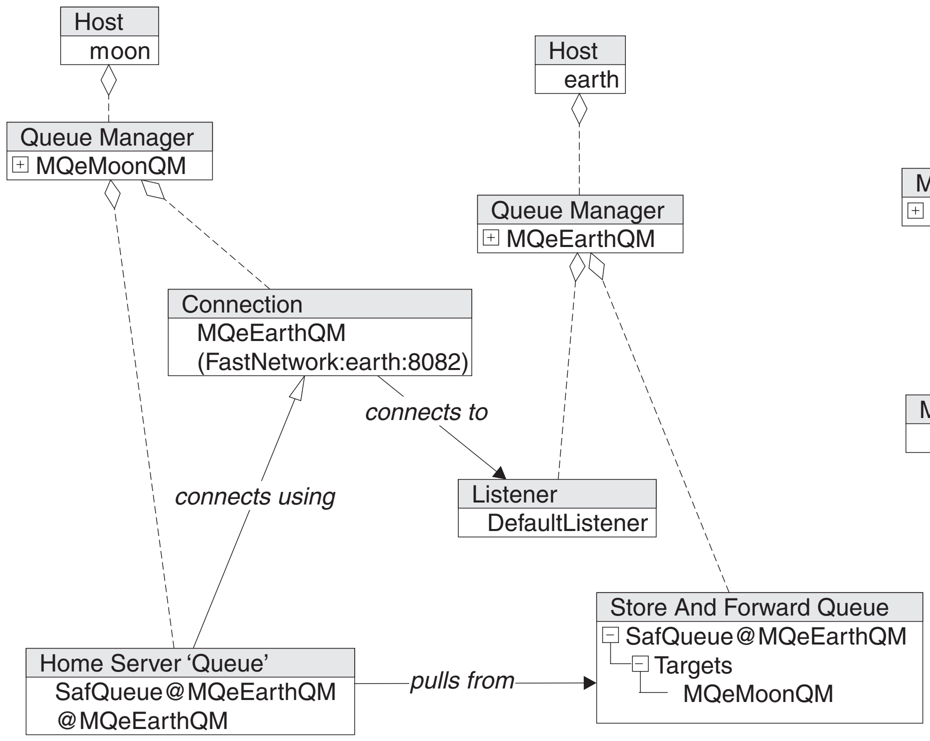


Figure 60. A client communicating with MQ

The following have been added:

- (Host)moon
- (QueueManager) MQeMoonQM on (Host)moon
- A connection definition from MQeMoonQM to a matching listener on MQeEarthQM to provide the connectivity between the two MQe queue managers.
- A store and forward queue on MQeEarthQM that accepts and holds messages for MQeMoonQM, and a home server queue on MQeMoonQM that pulls messages from the store and forward queue.
- A remote queue definition on the MQ queue manager that routes messages for MQeMoonQM to the transmission queue MQeEarth.XMITQ. This allows messages for MqeMoonQM to be placed on the transmission queue, from where they are pulled to MQeEarthQM.

The topology is more readily seen as message routes, as shown in the following diagram:

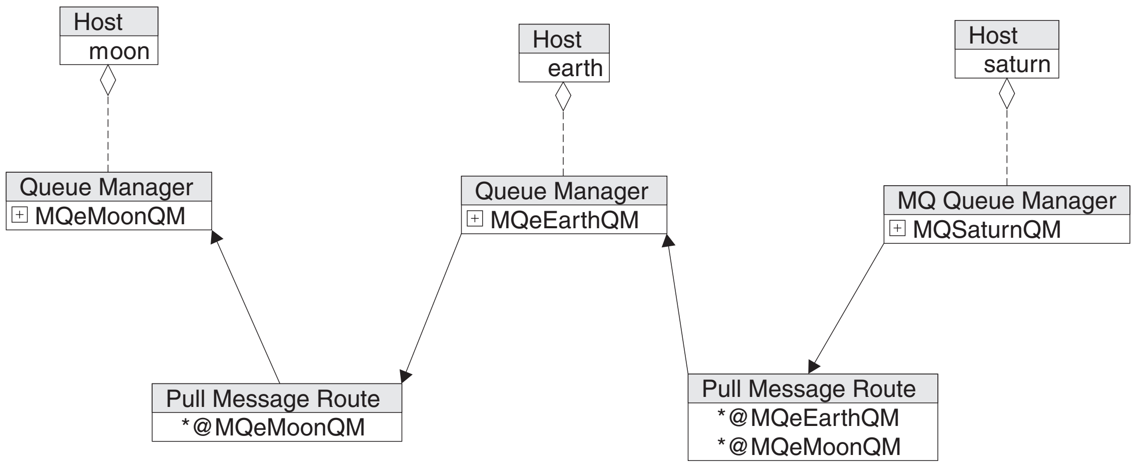


Figure 61. Simplified pull routes from MQ through an MQe gateway to an MQe device style queue manager

Messages can be pushed to MQ by using a via connection to chain remote queues, as shown in the following diagram:

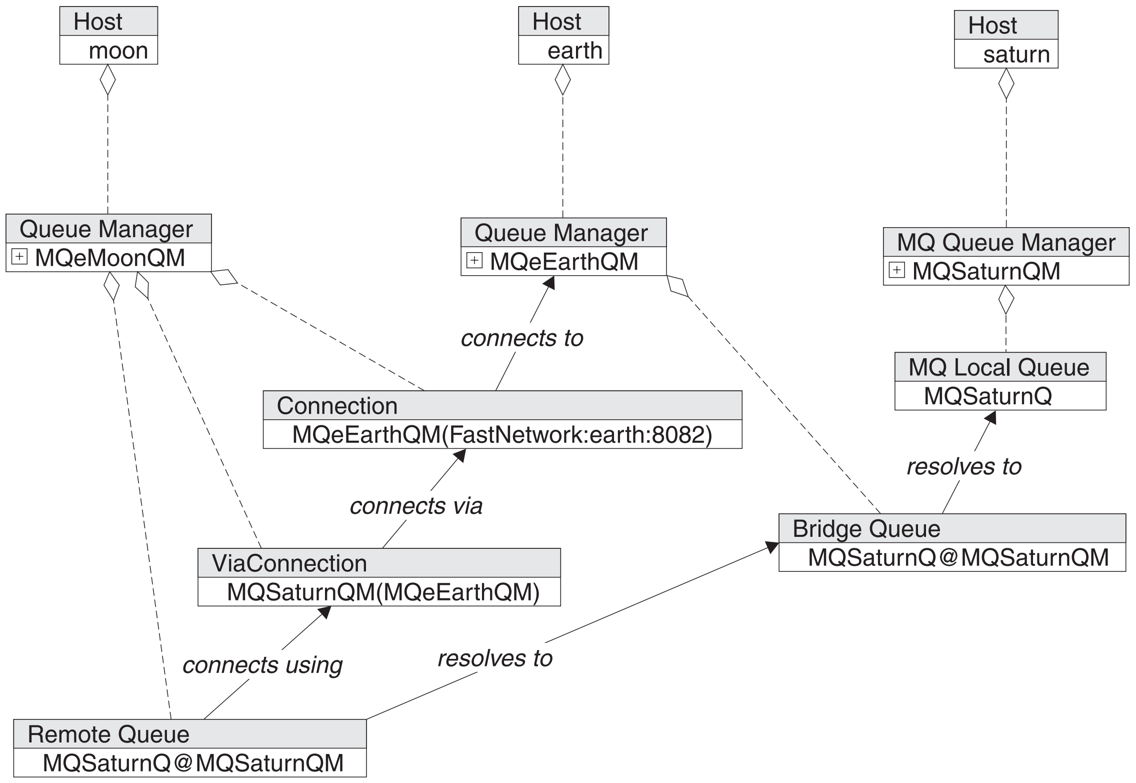


Figure 62. Pushing messages using a via connection

Here we have added a via connection, to route messages destined for MQSaturnQM vian MQeEarthQM, and we have added a remote queue definition for MQSaturnQ@MQSaturnQM. The messages can now flow from the client to MQ, as shown in the following diagram:

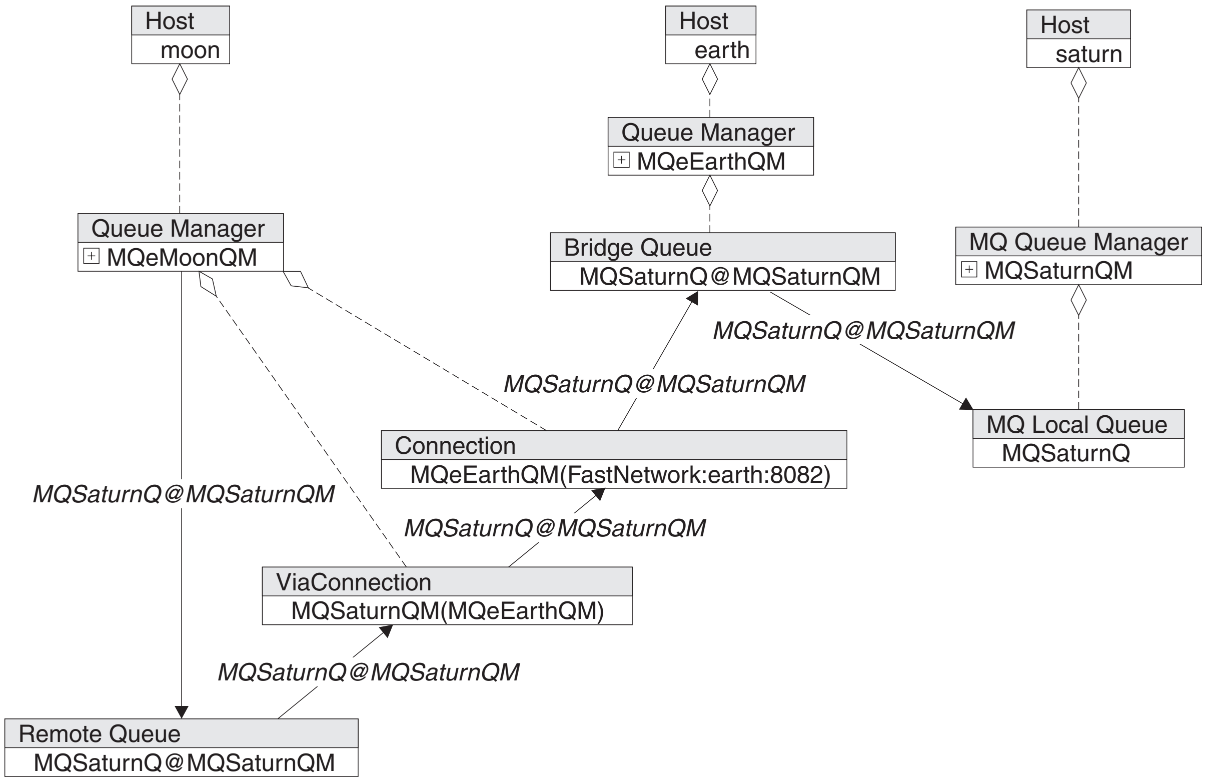


Figure 63. Pushing messages to MQ

This topology is more easily understood as a collection of message routes, as shown in the following diagram:

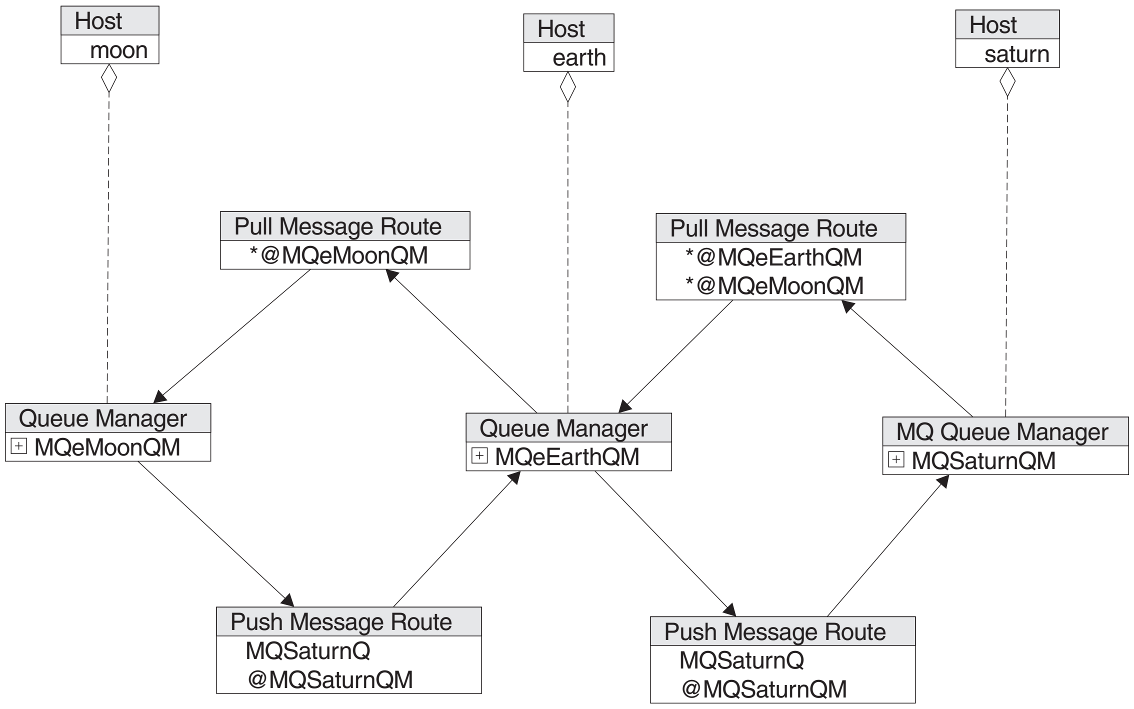


Figure 64. Simplified view showing routes which push messages from a device style MQe queue manager to an MQ queue manager

Security considerations

Remote queue definitions define the security requirements that must be satisfied by channels moving messages to target queues. The queue manager attribute rule defines the rules for upgrading channels; consequently with a sufficiently flexible rule, multiple security requirements can be met by a single channel.

When a message must be stored on a queue, either en route or at the destination, then the queue attribute rule determines if the channel security meets the requirements of the queue. Note however that there are message transfers that do not involve a channel, for example, when a home server places a message it has received from a store queue on to its destination queue. In these cases there are no security requirements to be satisfied in the transfer, but the message will be stored in its destination queue in a manner controlled by that queue's security characteristics. When the home server queue gets the message from the store queue, a channel is involved (with characteristics determined by the home server queue and which must be acceptable to the store queue). However, when the home server queue passes the message to the destination queue, there are no channel characteristics to be compared with the destination queue's security characteristics.

In a single hop, message transfer, the security checking is between the source and target queue managers. In multiple hop, asynchronous message transfers, security checking occurs stepwise over each hop.

Resolution rules

Resolution rules always start with a message being presented to a queue manager, with a specified destination queue manager name and a specified destination queue name. This is equivalent to the API call putMessage(queueManagerName, queueName, msg,....). The destinationQueueManagerName and destinationQueueName must identify a local queue onto which the message should eventually be placed.

Rule 1: Resolve queue manager aliases

If the queue manager has an alias mapping destinationQueueManagerName to another name, for example realQueueManagerName, then this substitution is made first, and the call:

```
putMessage(destinationQueueManagerName, destinationQueueName
```

is effectively transformed to

```
putMessage(realQueueManagerName, destinationQueueName.
```

From this point on destinationQueueManagerName is completely forgotten, and realQueueManagerName is used.

Queue resolution

The queue manager now looks for a queue to place the message on, selecting the queue with the best match according to the rules shown in *Exact match*, *Queue alias match*, *S&F queue*, *Queue discovery*, and *Failure*, below:

'Exact' match

Local queue or remote queue definition where the queue name matches the destinationQueueName and the queue's queue manager name matches the destinationQueueManagerName.

The term 'queues queue manager name' needs to be explained further. For a local queue this is the same as the name of the queue manager where the queue resides. For a local queue localQ@localQM, localQM is the queue's queue manager name.

For a remote queue definition remoteQ@remoteQM residing on localQM, the queues queue manager name is remoteQM.

Queue Alias Match

If a queue (remote definition or local) has a matching queue manager name and an alias and this alias matches destinationQueueName then this queue will considered a match. Effectively the put message call:

```
putMessage(destinationQueueManagerName, queueAliasName
```

is transformed to

```
putMessage(destinationQueueManagerName, realQueueName.
```

at this point. The original name of the queue used in the put call is entirely forgotten from this point on in the resolution.

S&F queue

If there is no exact match the queue manager searches for an inexact match. An inexact math is a Store and Forward queue that will accept messages for the given queue manager name. The search for a store and forward queue ignores the destinationQueueName. If an appropriate Store And Forward queue is found, then the message is put to it, using the destinationQueueManagerName and destinationQueueName, and the StoreAndForward queue stores the destination with the message.

Queue Discovery

If no queue has been found that will accept the message, and the message is not for a local queue, the queue manager tries to find the remote destination queue and create a remote queue definition for it automatically. This is called queue discovery. The queue manager can only perform discovery if:
- There is a connection definition to the destination queue manager
- There is an active communications path to the destination queue manager
- The destination queue exists
- There is a via connection to a queue manager where a remote connection definition exists

If discovery is successful, the newly created remote queue definition is used. This behaves as if an exact match on a remote queue definition had been found in the first place.

The remote queue definition created by discovery is always synchronous, even if the queue to which it resolves is asynchronous, or is a Store and forward queue.

Failure

If no queue has been found by the above steps, the message put is deemed to have failed.

Push across network

A message placed upon a remote queue is pushed across the network. The queue first locates a connection definition with the correct name, and then puts the message to the remote queue manager using the connection definition as the entry to the communications link.

The queue seeks a connection definition whose name is the same as the queue’s queue manager name. The connection may be a normal connection, or a via connection.

Normal

A normal connection points to a listener upon the destination queue manager. The put message command is routed directly to the destination queue manager. The putMessage call is then resolved just as if it had been placed on the queue manager via the API.

Via

A via connection points at another connection called the ’real’ connection. All commands performed on the via connection are delegated to the real connection. Via connections can be chained, and so the command may travel ’via’ several indirections before reach a real connection. The names of the put message destination are not changed by the use of a via connection.

Eventually the command is routed to a ’normal’ connection definition, then across the network to a queue manager, where the message put is resolved.

Home server pulling

Home server queues pull messages from Store and forward queues. The route of the pull spans only a single network hop. Only messages for the queue manager hosting the home server queue are pulled down. Messages pulled from the store and forward queue are presented to the queue manager using a normal put method call, and are then resolved as normal. The messages pulled down this way should all be destined for local queues.

Using aliases

Introduction to the use of aliases with MQe queues and queue managers

Aliases can be assigned for MQe queues to provide a level of indirection between the application and the real queues. For example, a queue can be given a number of aliases and messages sent to any of these names will be accepted by the queue.

Using queue aliases

See “Using queue aliases” on page 13 for information about the ways in which aliasing can be used with MQe queues.

Using queue manager aliases

This topic describes the ways in which aliasing can be used with MQe queue managers.

Addressing a queue manager with several different names

Suppose you have a queue manager `SERVER23QM` on the server `SAMPLEHOST`, listening on port `8082`. You have an application `SERVICEX` that accesses this queue manager, and wants to refer to the queue manager as `SERVICEXQM`. This can be achieved using an alias for the queue manager as follows:

- **Configure a connection on the SERVER23QM :**

  Connection Name/Target queue manager:
  : `SERVICEXQM`

  Description:
  : Alias definition to enable `SERVER23QM` to receive messages sent to `SERVICEXQM`

  Channel:
  : ″null″

Network Adapter:
"null"

Network adapter options:
"null"

- **Create a local queue on the SERVER23QM queue manager:**

  Queue Name:
  SERVICEXQ

  Queue Manager:
  SERVER23QM

  The server-side application takes messages from this queue, and process them, sending messages back to the client.

an MQe application can now put messages to the SERVICEXQ on either the SERVER23QM queue manager, or the SERVICEXQM queue manager. In either case, the message will arrive on the SERVICEXQ.

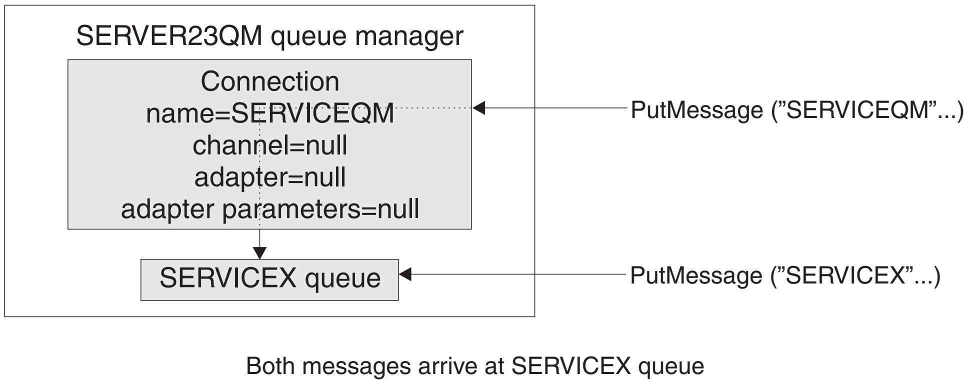


Figure 65. Addressing a queue manager with two different names

If the SERVICEXQ queue is moved to another queue manager, the connection alias can be set up on the new queue manager, and the applications do not need to be changed.

Different routings from one queue manager to another

Using the scenario in "Addressing a queue manager with several different names" on page 98, an MQe queue manager on a mobile device (MOBILE0058QM) can now access the SERVICEXQ queue in a number of different ways.

Aliasing on the sending side:

Using this method of routing, the receiving queue manager does not know that the sending queue manager has given it an alias name. The aliasing is confined to the sending queue manager only.

On the mobile device:

- Create a connection from MOBILE0058QM to the SERVER23QM queue manager:

  Connection name
  SERVER23QM

  Network Adapter parameter
  Network:SAMPLEHOST:8082

- Create an alias called SERVICEXQM for queue manager SERVER23QM

When a message is sent from the mobile device application to the `SERVICEXQM` queue manager, MQe maps the `SERVICEXQM` name to `SERVER23QM` in the connection , and sends the message to the `SERVER23QM` queue manager.

If the `Mobile58QM` then wished to send its messages to a different server queue manager, `Server24QM`, it would remove the alias `SERVICEXQM` from the `Server23QM` connection, and add it to a `Server24QM` connection. This has no impact on the receiving queue managers, or the sending applications.

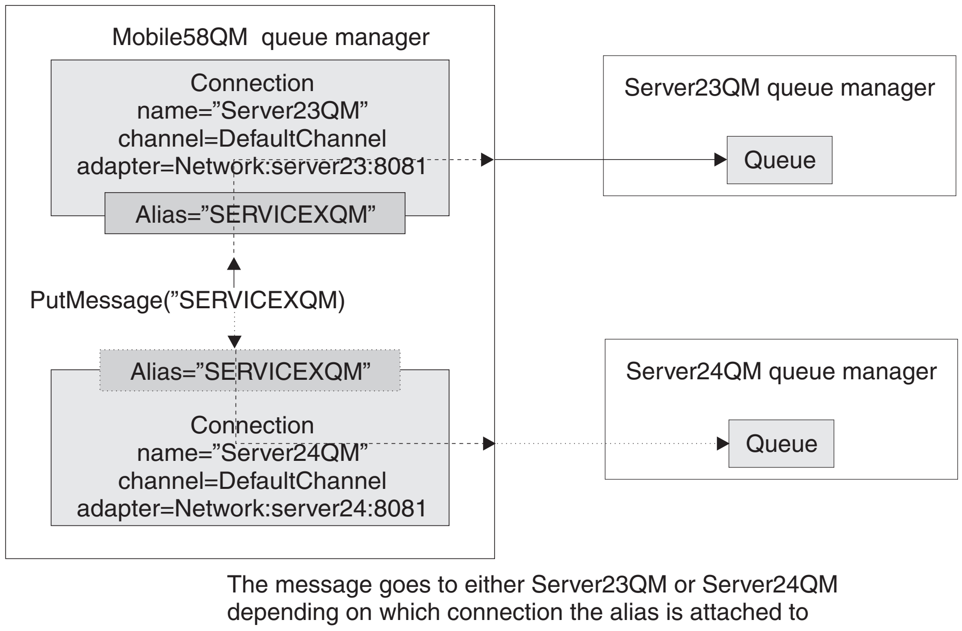


Figure 66. Addressing a queue manager with two different names

Virtual queue manager on the receiving side:

Using this method, the sending queue managers think that their messages are routed through an intermediate queue manager before reaching the target queue manager. The target queue manager doesn't actually exist. The 'intermediate' queue manager captures all the message traffic for this virtual target queue manager.

On the mobile device:

- Create a connection from `MOBILE0058QM` to the `SERVER23QM` queue manager:

  Connection name
  : `SERVER23QM`

  Network Adapter parameter
  : `Network:SAMPLEHOST:8082`

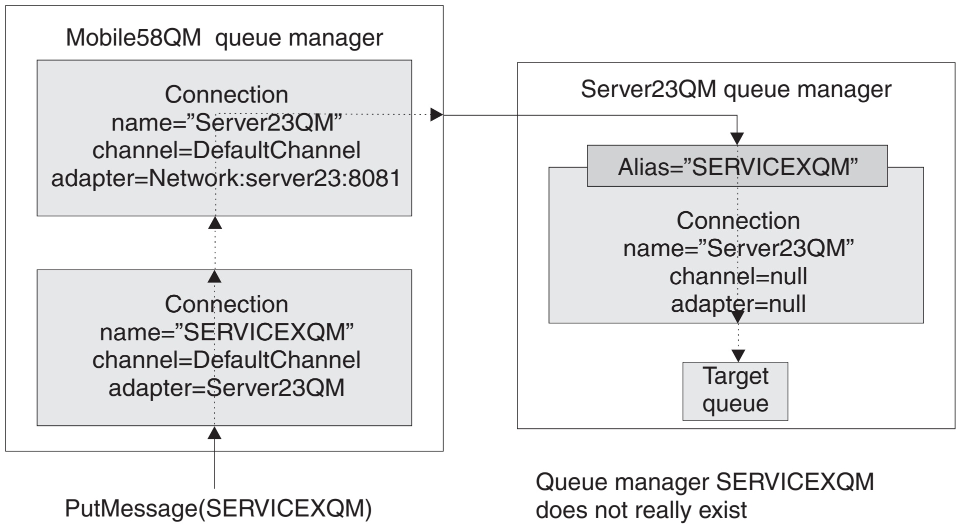
- Create a second connection to the `SERVICEXQM` that routes messages through the first connection:

  Connection name
  : `SERVICEXQM`

  Network Adapter parameter
  : `SERVER23QM`

  Note: This is not an alias. It is a *via routing*, indicating that messages headed for `SERVICEXQM` are to be routed via the `SERVER23QM` queue manager on the receiving side.

The via routing on the mobile device causes any messages that are put to SERVICEXQM to be directed to Server23QM. Server23QM gets the messages and notes that they are destined for the SERVICEXQM queue manager. It resolves the SERVICEXQM name and finds that it is an alias which represents the Server23QM queue manager (itself). The Server23QM queue manager then accepts the messages and puts them onto the queue.



Mobile58QM queue manager
Connection name="Server23QM" channel=DefaultChannel adapter=Network:server23:8081
Connection name="SERVICEXQM" channel=DefaultChannel adapter=Server23QM
PutMessage(SERVICEXQM)
Server23QM queue manager
Alias="SERVICEXQM"
Connection name="Server23QM" channel=null adapter=null
Target queue
Queue manager SERVICEXQM does not really exist

Figure 67. Addressing a queue manager with two different names

As an alternative to the above, you can keep the SERVICEXQM in existence, but move it from its original machine to the same machine (but a different JVM) as the Server23QM queue manager. SERVICEXQM needs to listen on a different port, so the connection from Server23QM to SERVICEXQM needs to be changed as well.

Using adapters

Describes the use of storage adapters and communications adapters in MQe applications, and explains how to write your own adapters

This chapter describes how to implement adapters in an MQe application. You can use MQe adapters to map MQe to storage or communications device interfaces. You can also write your own adapters.

This chapter contains the following sections:
- Storage adapters
- Communications adapters
- How to write adapters

Storage adapters

MQe provides the following storage adapters:

Storage adapters

MQeCaseInsensitiveDiskAdapter
Provides support for case insensitive matching when locating a specific file in permanent storage.

MQeDiskFieldsAdapter
Provides support for reading and writing to persistent storage.

MQeMappingAdapter
Provides support for mapping long file names to short file names.

MQeMemoryFieldsAdapter
Provides support for reading and writing to non-persistent storage.

MQeMidpFieldsAdapter
Provides support for reading and writing to permanent storage within a MIDP environment.

MQeReducedDiskFieldsAdapter
Provides support for high speed writing to permanent storage.

Note that you cannot alter the behavior of these adapters. For more information on the specific behavior of each storage adapter, refer to the MQe Java Programming Reference and the MQe C Programming Reference.

Communications adapters

MQe provides the following communications adapters:.

Communications adapters

MQeMidpHttpAdapter
Provides support for reading and writing to the network using the HTTP 1.0 protocol in a MIDP environment.

MQeTcpipHistoryAdapter
Provides support for reading and writing to the network using the TCP protocol. This adapter provides the best TCP performance by chaching recently used data. Therefore, we recommend that you use this adapter.

MQeTcpipLengthAdapter
Provides support for reading and writing to the network using the TCP protocol.

MQeTcpipHttpAdapter
Provides support for reading and writing to the network using the HTTP 1.0 protocol. Also provides support for passing HTTP requests through proxy servers.

Note: If using the Microsoft® JVM, the http:proxyHost and http:proxyPort properties are automatically set by the JVM using the settings in the Internet Explorer. If the use of proxies is not required for MQe, set the http.proxySet Java property to false.

MQeUdpipBasicAdapter
Provides support for reading and writing to the network using the UDP protocol. This adapter uses only one port on the server. The behavior of this adapter is particularly sensitive to the various Java property settings, as detailed in the MQe Java Programming Reference.

MQeWESAuthenticationAdapter
Provides support for passing HTTP requests through MQe authentication proxy servers and transparent proxy servers.

You can modify the behavior of these adapters using Java properties. For more information on how to use these properties and their effect on each communications adapter, refer to the MQe Java Programming Reference.

You can also write your own adapters to tailor MQe for your own environment. The next section describes some adapter examples that are supplied to help you with this task.

How to write adapters

You can also write your own adapters to tailor MQe for your own environment. This topic describes some adapter examples that are supplied to help you with this task.

This example is not intended as a replacement for the adapters that are supplied with MQe, but as a simple introduction on how to create a communications adapter.

To use your communications adapter, you must specify the correct class name when creating the listener on the server queue manager, and specify the connection definition on the client queue manager.

All communications adapters must inherit from MQeCommunicationsAdapter and must implement the required methods. In order to show how this might be done we shall use the example adapter, `examples.adapters.MQeTcpipLengthGUIAdapter`. This is a simple example that accepts data to be written. It also places the data length and the amount of data to be written to standard out, at the front of the data. When the adapter reads data, the data length is written to standard out. Proper error checking and recovery is not carried out. This must be added to any adapter written by a user.

MQe adapters use the default constructor. For this reason, an `activate()` method is used in order to set up the adapter with an `open()` method used to prepare the adapter for communication.

The `activate()` method is called only once in the life cycle of an adapter and is, therefore, used to set up the information from MQePropertyProvider. The MQePropertyProvider looks internally to verify that the specified property is available. If it is not available, it checks the Java properties. In this way, it is possible for a user to specify a property that may be set by the application or JVM command line. The MQeCommunicationsAdapter provides two variables that allow the adapter to identify its role within the communications conversation:

- If the adapter is being used by the MQeListener, the variable `listeningAdapter` is set to true.
- If the adapter has been created by the listening adapter in response to an incoming request, the `responderAdapter` variable is set to true.

The following code, taken from the `activate()` method, shows how to obtain the information from the MQePropertyProvider.

```
if (!listeningAdapter) {
  // if we are not a listening adapter we need the
  address of the server
  address = info.getProperty
      (MQeCommunicationsAdapter.COMMS_ADAPTER_ADDRESS);
}
```

The `open()` method is called before each conversation and must, therefore, be used to set information that needs to be reset for each request or response. For example, an adapter that is not persistent needs to create a socket each time it is opened. The following code shows the use of the variables that identify the role of the adapter role within the conversation:

```
  if (listeningAdapter && null == serverSocket) {
  serverSocket = new ServerSocket(port);
} else if (!responderAdapter && null == mySocket) {
  mySocket = new Socket(InetAddress.getByName(address), port);
}
```

Once the `activate()` and `open()` methods have been called, the listening adapter `waitForContact` method is called. This method must wait at named location. In an IP network, this will be a named port. When a request is received, a new adapter is created.

Note: This method must set the listeningAdapter to false and the responderAdapter to true.

Once the adapter has been set up correctly, you must must returned it to the caller. The following code shows how to do this:

```
MQeTcpipLengthGUIAdapter clientAdapter =
    (MQeTcpipLengthGUIAdapter)
      MQeCommunicationsAdapter.createNewAdapter(info);

    // set the boolean variables so the adapter
  //  knows it is a responder. the listening
    // variable will have been set to true as
  //  the MQePropertyProvider has the relevant
    // information to create
  //  this listening adapter.  We must therefore reset the
    // listeningAdapter variable to false and the
  //responderAdapter variable to true.
  clientAdapter.responderAdapter = true;
  clientAdapter.listeningAdapter = false;

  // Assign the new socket to this new adapter
  clientAdapter.setSocket(clientSocket);
  return clientAdapter;
```

The initiator adapter and responder adapter are responsible for the main part of the conversation. The initiator starts the conversation. The responder is created by the listening adapter, reads the request that is passed back to MQe, which then writes a response. The adapter determines how the read and the write are undertaken. The example uses a BufferedInputStream and a BufferedOutputStream.

Note: Use a a non-blocking mode of reading and writing. This enables the adapter to respond to requests to shutdown.

The following code, taken from the `waitForContact()` method, shows how the non-blocking read can be written. As MQe supports all Java runtime environments we are unable to use Java version 1.4 specific classes for our examples, although this version does contain new non-blocking classes

```
do {
      try {
        clientSocket = serverSocket.accept();
      } catch (InterruptedIOException iioe) {
          if (MQeThread.getDemandStop()) {
             throw iioe;
          }
      }
    } while (null == clientSocket);
```

An example communications adapter

This example uses the standard Java classes to manipulate TCPIP and adds a protocol of its own on top. This protocol has a header consisting of a four byte length of the data in the data packet followed by the actual data. This is so that the receiving end knows how much data to expect.

This example is not meant as a replacement for the adapters that are supplied with MQe but rather as a simple introduction into how to create communications adapters. In reality, much more care should be taken with error handling, recovery, and parameter checking. Depending on the MQe configuration used, the supplied adapters may be sufficient.

A new class file is constructed, inheriting from MQeAdapter. Some variables are defined to hold this adapter's instance information, that is the name of the host, port number and the output stream objects.

Note: With communications, ensure that the connection information is correct. For example, the http connection in J2ME has no timeout implementation. In J2SE, the client times out with an IO Exception. In Midp the server times out. If the default read-timeout has been increased for the J2SE client, the same exception is thrown, that is `com.ibm.mqe.MQeException: Data: (code=7)`. This is because the server writes back the exception to the client and the client cannot restore this data.

The MQeAdapter constructor is used for the object, so no additional code needs to be added for the constructor.

```
public class MyTcpipAdapter extends MQeAdapter
  {
  protected     String               host         = "";
  protected     int                  port         = 80;
  protected     Object               readLock     = new Object( );
  protected     ServerSocket         serversocket = null;
  protected     Socket               socket       = null;
  protected     BufferedInputStream  stream_in    = null;
  protected     BufferedOutputStream stream_out   = null;
  protected     Object               writeLock    = new Object( );
```

Next the `activate` method is coded. This is the method that extracts from the file descriptor the name of the target network address if a connector, or the listening port if a listener. The fileDesc parameter contains the adapter class name or alias name, and any network address data for the adapter for example `MyTcpipAdapter:127.0.0.1:80`. The thisParam parameter contains any parameter data that was set when the connection was defined by administration, the normal value would be "?Channel". The thisOpt parameter contains the adapter setup options that were set by administration, for example `MQe_Adapter_LISTEN` if this adapter is to listen for incoming connections.

```
  public void   activate( String   fileDesc,
                          Object   thisParam,
                          Object   thisOpt,
                          int      thisValue1,
                          int      thisValue2 ) throws Exception
    {
    super.activate( fileDesc,
                    thisParam,
                    thisOpt,
                    thisValue1,
                    thisValue2 );
    /* isolate the TCP/IP address -
                 "MyTcpipAdapter:127.0.0.1:80"     */
    host = fileId.substring( fileId.indexOf( ':' ) + 1 );
    i    = host.indexOf( ':' );
    /* find delimiter     */
    if ( i > -1 )
    /* find it ?          */
      {
      port = (new Integer( host.substring( i + 1 ) )).intValue( );
      host = host.substring( 0, i );
      }
    }
```

The `close` method needs to be defined to close the output streams and flush any remaining data from the stream buffers. `Close` is called many time during a session between a client and a server, however, when the channel has completely finished with the adapter it calls MQe with the option `MQe_Adapter_FINAL`. If the adapter is to have one socket connection for the life of the channel then the call with `MQe_Adapter_FINAL` set, is the one to use to actually close the socket, other calls should just flush the buffers. If however a new socket is to be used on each request, then each call to MQe should close the socket, subsequent `open` calls should allocate a new socket:

```
  public void    close(  Object  opt ) throws Exception
    {
    if ( stream_out   != null )
  /* output stream ?     */
      {
      stream_out.flush();
  /* empty the  buffers  */
      stream_out.close();
  /* close it           */
      stream_out = null;
  /* clear              */
      }
    if ( stream_in    != null )
  /* input stream ?     */
      {
      stream_in.close();
  /* close it           */
      stream_in = null;
  /* clear              */
      }
    if ( socket       != null )
  /* socket ?           */
      {
      socket.close();
  /* close it           */
      socket = null;
  /* clear              */
      }
    if ( serversocket != null )
  /* serversocket ?     */
      {
      serversocket.close();
  /* close it           */
      serversocket = null;
  /* clear              */
      }
    host = "";
    port = 80;
    }
```

The control method needs to be coded to handle an MQe_Adapter_ACCEPT request, to accept an incoming connect request. This is only allowed if the socket is a listener (a server socket). Any options that were specified for the listen socket (excluding MQe_Adapter_LISTEN) are copied to the socket created as a result of the accept. This is accomplished by the use of another control option MQe_Adapter_SETSOCKET this allows a socket object to be passed to the adapter that was just instantiated.

```
public Object control( Object opt, Object ctrlObj ) throws Exception
    {
    if ( checkOption( opt, MQe.MQe_Adapter_LISTEN      ) &&
         checkOption( opt, MQe.MQe_Adapter_ACCEPT ) )
      {
      /* CtrlObj - is a string representing the
              file descriptor of the */
      /*          MQeAdapter object to be returned e.g. "MyTcpip:"   */
      Socket  ClientSocket = serversocket.accept();
    /* wait connect  */
      String  Destination  = (String) ctrlObj;
    /* re-type object*/
      int i  = Destination.indexOf( ':' );
      if ( i < 0 )
        throw new MQeException( MQe.Except_Syntax,
                                "Syntax:" + Destination );
      /* remove the Listen option   */
      String NewOpt = (String) options;
    /* re-type to string   */
      int j  = NewOpt.indexOf( MQe.MQe_Adapter_LISTEN );
      NewOpt = NewOpt.substring( 0, j ) +
           NewOpt.substring
            ( j + MQe.MQe_Adapter_LISTEN.length( ) );
                MQeAdapter Adapter = MQe.newAdapter
                    ( Destination.substring( 0,i+1 ),
                                 parameter,
                                 NewOpt + MQe_Adapter_ACCEPT,
                                 -1,
                                 -1 );
      /* assign the new socket to this new adapater  */
      Adapter.control( MQe.MQe_Adapter_SETSOCKET, ClientSocket);
      return( Adapter );
      }
    else
    if ( checkOption( opt, MQe.MQe_Adapter_SETSOCKET ) )
      {
      if ( stream_out != null )  stream_out.close();
      if ( stream_in  != null )  stream_in .close();
      if ( ctrlObj    != null )
    /* socket supplied ?*/
        {
        socket     = (Socket) ctrlObj;
    /* save the socket     */
        stream_in  = new BufferedInputStream (socket.getInputStream ());
        stream_out = new BufferedOutputStream(socket.getOutputStream());
        }
    else
      return( super.control( opt, ctrlObj ) );
  }
```

The open method needs to check for a listening socket or a connector socket and create the appropriate socket object. Reinitialization of the input and output streams is achieved by using the control method, passing it a new socket object. The opt parameter may be set to MQe_Adapter_RESET, this means that any previous operations are now complete any new reads or writes constitute a new request.

```
  public void  open( Object opt ) throws Exception
    {
    if ( checkOption( MQe.MQe_Adapter_LISTEN ) )
      serversocket = new ServerSocket( port, 32 );
    else
      control( MQe.MQe_Adapter_SETSOCKET,
          new Socket( host, port ) );
    }
```

The read method can take a parameter specifying the maximum record size to be read.

This example calls internal routines to read the data bytes and do error recovery (if appropriate) then return the correct length byte array for the number of bytes read. Ensure that only one read at a time occurs on this socket. The opt parameter may be set to:

MQe_Adapter_CONTENT
read any message content

MQe_Adapter_HEADER
read any header information

```
{ public byte[] read( Object opt, int recordSize ) throws Exception

    int Count = 0;
  /* number bytes read   */
    synchronized ( readLock )
  /* only one at a time  */
      {
      if ( checkOption(opt, MQe.MQe_Adapter_HEADER )  )
        {
        byte lreclBytes[] = new byte[4];
  /* for the data length */
        readBytes( lreclBytes, 0, 4 );
  /* read the length     */
        int  recordSize = byteToInt( lreclBytes, 0, 4 );
        }
      if ( checkOption( opt, MQe.MQe_Adapter_CONTENT   ) )
        {
        byte Temp[] = new byte[recordSize];
  /* allocate work array */
```

```
        Count = readBytes( Temp, 0, recordSize);/* read data   */
        }
      }
    if ( Count < Temp.length )
  /* read all length ?   */
      Temp    = MQe.sliceByteArray( Temp, 0, Count );
    return ( Temp );
  /* Return the data     */
    }
```

The readByte method is an internal routine designed to read a single byte of data from the socket and to attempt to retry any errors a specific number of times, or throw an end of file exception if there is no more data to be read.

```
  protected int readByte( ) throws Exception
    {
    int intChar    = -1;
  /* input characater    */
    int RetryValue =  3;
  /* error retry count   */
    int Retry = RetryValue + 1;
  /* reset retry count   */
    do{
  /* possible retry      */
      try
  /* catch io errors     */
        {
        intChar = stream_in.read();
  /* read a character    */
        Retry   = 0;
   /* dont retry          */
        }
      catch ( IOException e )
  /* IO error occured    */
        {
        Retry   = Retry - 1;
  /* decrement           */
        if ( Retry == 0 )  throw e;
  /* more attempts ?     */
        }
      } while ( Retry != 0 );
  /* more attempts ?     */
    if ( intChar == -1 )
  /* end of file ?       */
      throw new EOFException();
  /* ... yes, EOF        */
    return( intChar );
  /* return the byte     */
    }
```

The readBytes method is an internal routine designed to read a number of bytes of data from the socket and to attempt to retry any errors a specific number of times, or throw an end of file exception if there is no more data to be read.

```
  protected int readBytes( byte buffer[],
          int offset, int recordSize )
    throws Exception
    {
    int RetryValue = 3;
    int i = 0;
  /* start index         */
    while ( i < recordSize )
  /* got it all in yet ? */
      {
  /* ... no              */
      int NumBytes = 0;
  /* read count          */
```

```
      /* retry any errors based on the QoS Retry value */
      int Retry = RetryValue + 1;
  /* error retry count   */
      do{
  /* possible retry      */
        try
  /* catch io errors     */
          {
          NumBytes = stream_in.read( buffer,
          offset + i, recordSize - i );
          Retry    = 0;
  /* no retry            */
          }
        catch ( IOException e )
  /* IO error occured    */
          {
          Retry    = Retry - 1;
  /* decrement           */
          if ( Retry == 0 )  throw e;
  /* more attempts ?     */
          }
        } while ( Retry != 0 );
  /* more attempts ?     */
      /* check for possible end of file  */
      if ( NumBytes < 0 )
  /* errors ?            */
        throw new EOFException( );
  /* ... yes             */
      i = i + NumBytes;
  /* accumulate          */
      }    return ( i );
  /* Return the count    */
    }
```

The readln method reads a string of bytes terminated by a 0x0A character it will ignore 0x0D characters.

```
    {
    synchronized ( readLock )
 /* only one at a time  */
      {
      /* ignore the 4 byte length   */
      byte lreclBytes[] = new byte[4];  /* for the data length */
      readBytes( lreclBytes, 0, 4 );
/* read the length     */

      int intChar    = -1;
 /* input characater    */
      StringBuffer Result = new StringBuffer( 256 );
      /* read Header from input stream          */
      while ( true )
 /* until "newline"     */
        {
        intChar = readByte( );
/* read a single byte  */
        switch ( intChar )
/* what character      */
          {
          case -1:
/* ... no character    */
            throw new EOFException();
/* ... yes, EOF        */
          case 10:
/* eod of line         */
            return( Result.toString() );
/* all done            */
          case 13:
/* ignore              */
```

```
            break;
          default:
/* real data           */
            Result.append( (char) intChar );
/* append to string    */
          }
/* end of line ?       */
        }
      }
    }
```

The status method returns status information about the adapter. In this example it returns for the option MQe_Adapter_NETWORK the network type (TCPIP), for the option MQe_Adapter_LOCALHOST it returns the tcpip local host address.

```
public String status( Object  opt ) throws Exception
    {
    if ( checkOption( opt, MQe.MQe_Adapter_NETWORK ) )
      return( "TCPIP" );
    else
    if ( checkOption( opt, MQe.MQe_Adapter_LOCALHOST ) )
      return( InetAddress.getLocalHost( ).toString() );
    else
    return( super.status( opt ) );
    }
```

The write method writes a block of data to the socket. It needs to ensure that only one write at a time can be issued to the socket. In this example it calls an internal routine writeBytes to write the actual data and perform any appropriate error recovery.

The opt parameter may be set to:

MQe_Adapter_FLUSH
flush any data in the buffers

MQe_Adapter_HEADER
write any header records

MQe_Adapter_HEADERRSP
write any header response records

```
  public void write( Object opt, int recordSize, byte data[] )
    throws Exception
    {
    synchronized ( writeLock )
  /* only one at a time  */
      {
      if ( checkOption( opt, MQe.MQe_Adapter_HEADER    ) ||
           checkOption( opt, MQe.MQe_Adapter_HEADERRSP )  )
        writeBytes( intToByte( recordSize ), 0, 4 );
  /* write length*/
      writeBytes( data, 0, recordSize );
  /* write the data      */
      if ( checkOption( opt, MQe.MQe_Adapter_FLUSH  ) )
        stream_out.flush( );
  /* make sure it is sent */
      }

    }
```

The writeBytes is an internal method that writes an array (or partial array) of bytes to a socket, and attempt a simple error recovery if errors occur.

```
protected void writeBytes( byte buffer[], int offset, int recordSize )
    throws Exception
    {
```

```
    if ( buffer != null )
  /* any data ?          */
      {
      /* break the data up into manageable chuncks */
      int i = 0;
  /* Data index          */
      int j = recordSize;
  /* Data length         */
      int MaxSize    = 4096;
  /* small buffer        */
      int RetryValue = 3;
  /* error retry count   */
      do{
  /* as long as data     */
        if ( j < MaxSize )
  /* smallbuffer ?       */
          MaxSize = j;
        int Retry = RetryValue + 1;
  /* error retry count   */
        do{
  /* possible retry      */
           try
  /* catch io errors     */
            {
            stream_out.write( buffer, offset + i, MaxSize );
            Retry = 0;
  /* don't retry         */
            }
          catch ( IOException e )
  /* IO error occured    */
            {
            Retry = Retry - 1;
  /* decrement           */
            if ( Retry == 0 )  throw e;
  /* more attempts ?     */
            }
          } while ( Retry != 0 );
  /* more attempts ?     */

        i = i + MaxSize;
  /* update index        */
        j = j - MaxSize;
  /* data left           */
        } while ( j > 0 );
  /* till all data sent  */
      }
    }
```

The `writeLn` method writes a string of characters to the socket, terminating with 0x0A and 0x0D characters.

The opt parameter may be set to:

MQe_Adapter_FLUSH
flush any data in the buffers

MQe_Adapter_HEADER
write any header records

MQe_Adapter_HEADERRSP
write any header response records

```
  public void writeln( Object opt, String data ) throws Exception
    {
    if ( data == null )
  /* any data ?          */
```

```
      data = "";
    write( opt, -1, MQe.asciiToByte( data + "\r\n" ) );
  /* write data  */
    }
```

This is now a complete (though very simple) TCPIP adapter that will communicate to another copy of itself, one of which was started as a listener and the other started as a connector.

An example message store adapter

This example creates an adapter for use as an interface to a message store. It uses the standard Java i/o classes to manipulate files in the store.

This example is not meant as a replacement for the adapters that are supplied with MQe, but rather as a simple introduction to creating a message store adapter.

A new class file is constructed, inheriting from MQeAdapter. Some variables are defined to hold this adapter's instance information, such as the name of the file/message and the location of the message store.

The MQeAdapter constructor is used for the object, so no additional code needs to be added for the constructor.

```
public class MyMsgStoreAdapter extends    MQeAdapter
                               implements FilenameFilter
  {
  protected String  filter   = "";
  /* file type filter    */
  protected String  fileName = "";
  /* disk file name      */
  protected String  filePath = "";
  /* drive and directory */
  protected boolean reading  = false;
/* opened for reading  */
  protected boolean writing  = false;
```

Because this adapter implements FilenameFilter, the following method must be coded. This is the filtering mechanism that is used to select files of a certain type within the message store.

```
  public boolean accept( File dir, String name )
    {
    return( name.endsWith( filter ));
    }
```

Next the `activate` method is coded. This is the method that extracts, from the file descriptor, the name of the directory to be used to hold all the messages.

The Object parameter on the method call may be an attribute object. If it is, this is the attribute that is used to encode or decode the messages in the message store.

The Object options for this adapter are:
- MQe_Adapter_READ
- MQe_Adapter_WRITE
- MQe_Adapter_UPDATE

Any other options should be ignored.

```
public void   activate( String  fileDesc,
                        Object  param,
                        Object  options,
                        int     value1,
                    int     value2 ) throws Exception
    {
    super.activate( fileDesc, param, options, lrecl, noRec );
    filePath    = fileId.substring( fileId.indexOf( ':' ) + 1 );
    String Temp = filePath;
  /* copy the path data  */
    if ( filePath.endsWith( File.separator ) )
  /* ending separator ?  */
      Temp      = Temp.substring( 0, Temp.length( ) -
                                File.separator.length( ) );
    else
      filePath  = filePath + File.separator;
  /* add separator       */
    File diskFile = new File( Temp );
    if ( ! diskFile.isDirectory( ) )
  /* directory ?         */
      if ( ! diskFile.mkdirs( ) )
  /* does mkDirs work ?  */
        throw new MQeException( MQe.Except_NotAllowed,
                          "mkdirs '" + filePath + "' failed" );
    filePath = diskFile.getAbsolutePath( ) + File.separator;
    this.open( null );
    }
```

The `close` method disallows reading or writing.

```
public void close( Object opt ) throws Exception
    {
    reading  = false;
/* not open for reading*/
    writing  = false;
/* not open for writing*/
    }
```

The `control` method needs to be coded to handle an `MQe_Adapter_LIST` that is, a request to list all the files in the directory that satisfy the filter. Also to handle an `MQe_Adapter_FILTER` that is a request to set a filter to control how the files are listed.

```
public Object control( Object opt, Object ctrlObj ) throws Exception
    {
    if ( checkOption( opt, MQe.MQe_Adapter_LIST   ) )
      return( new File( filePath ).list( this ) );
    else
    if ( checkOption( opt, MQe.MQe_Adapter_FILTER ) )
      {
      filter = (String) ctrlObj;
  /* set the filter      */
      return( null );
  /* nothing to return   */
      }
    else
    return( super.control( opt, ctrlObj ) );
  /* try ancestor        */
    }
```

The `erase` method is used to remove a message from the message store.

```
  public void erase( Object opt ) throws Exception
    {
    if ( opt instanceof String )
  /* select file ?       */
      {
      String FN = (String) opt;
  /* re-type the option  */
      if ( FN.indexOf( File.separator ) > -1 )
  /* directory ?         */
        throw new MQeException( MQe.Except_Syntax,
            "Not allowed" );
      if ( ! new File( filePath + FN ).delete( ) )
        throw new MQeException( MQe.Except_NotAllowed,
              "Erase failed" );
      }
    else
      throw new MQeException( MQe.Except_NotSupported,
              "Not supported" );
    }
```

The open method sets the Boolean values that permit either reading of messages or writing of messages.

```
public void open( Object opt ) throws Exception
    {
    this.close( null );
  /* close any open file */
    fileName = null;
  /* clear the filename  */
    if ( opt instanceof String )
  /* select new file ?   */
      fileName = (String) opt;
  /* retype the name     */
    reading  = checkOption( opt, MQe.MQe_Adapter_READ   ) ||
               checkOption( opt, MQe.MQe_Adapter_UPDATE );
    writing  = checkOption( opt, MQe.MQe_Adapter_WRITE  ) ||
               checkOption( opt, MQe.MQe_Adapter_UPDATE );
    }
```

The readObject method reads a message from the message store and recreates an object of the correct type. It also decrypts and decompresses the data if an attribute is supplied on the activate call. This is a special function in that a request to read a file that satisfies the matching criteria specified in the parameter of the read, returns the first message it encounters that satisfies the match.

```
public Object readObject( Object opt ) throws Exception
    {
    if ( reading )
      {
      if ( opt instanceof MQeFields )
        {
        /* 1. list all files in the directory */
        /* 2. read each file in turn and restore as a Fields object */
        /* 3. try an equality check - if equal then return that object */
        String List[] = new File( filePath ).list( this );
        MQeFields Fields = null;
        for ( int i = 0; i < List.length; i = i + 1 )
          try
            {
            fileName = List[i];
      /* remember the name   */
            open( fileName );
      /* try this file       */
            Fields = (MQeFields) readObject( null );
            if ( Fields.equals( (MQeFields) opt ) )
      /* match ?     */
              return( Fields );
            }
          catch ( Exception e )
      /* error occured       */
            {
            }
      /* ignore error        */
        throw new MQeException( Except_NotFound, "No match" );
        }
      /* read the bytes from disk   */
      File diskFile = new File( filePath + fileName );
      byte data[] = new byte[(int) diskFile.length()];
      FileInputStream InputFile = new FileInputStream( diskFile );
      InputFile.read( data );              /* read the file data  */
      InputFile.close( );                  /* finish with file    */
      /* possible Attribute decode of the data       */
      if ( parameter instanceof MQeAttribute )
    /* Attribute encoding ?*/
        data = ((MQeAttribute) parameter).decodeData( null,
                                                       data,
                                                       0,
                                                       data.length );
      MQeFields FieldsObject = MQeFields.reMake( data, null );
      return( FieldsObject );
      }
    else
      throw new MQeException( MQe.Except_NotSupported,
              "Not supported" );
    }
```

The status method returns status information about the adapter. In this examples it can return the filter type or the file name.

```
public String status( Object  opt  ) throws Exception
    {
    if ( checkOption( opt, MQe.MQe_Adapter_FILTER   ) )
      return( filter   );
    if ( checkOption( opt, MQe.MQe_Adapter_FILENAME ) )
      return( fileName );
    return( super.status( opt ) );
    }
```

The writeObject method writes a message to the message store. It compresses and encrypts the message object if an attribute is supplied on the activate method call.

```
public void writeObject( Object  opt,
                           Object  data ) throws Exception
    {
    if ( writing && (data instanceof MQeFields) )
      {
      byte dump[] = ((MQeFields) data).dump( );
  /* dump object */
      /* possible Attribute encode of the data                        */
      if ( parameter instanceof MQeAttribute )
        dump = ((MQeAttribute) parameter).encodeData( null,
                                                       dump,
                                                       0,
                                                       dump.length );
      /* write out the object bytes                                   */
      File diskFile = new File( filePath + fileName );
      FileOutputStream OutputFile = new FileOutputStream( diskFile );
      OutputFile.write( dump );               /* write the data      */
      OutputFile.getFD().sync( );             /* synchronize disk    */
      OutputFile.close();                     /* finish with file    */
      }
    else
      throw new MQeException( MQe.Except_NotSupported, "Not supported" );
    }
```

This is now a complete (though very simple) message store adapter that reads and writes message objects to a message store.

Variations of this adapter could be coded for example to store messages in a database or in nonvolatile memory.

The WebSphere Everyplace Suite (WES) communications adapter

MQe provides sophisticated security that allows applications to run over HTTP, through the protection of an Internet firewall. The purpose of the WebSphere® Everyplace® communications adapter is to allow MQe applications to authenticate themselves with the WebSphere Everyplace authentication proxy and thus allow messages to flow through it. The following diagram shows a basic scenario with two applications communicating over the Internet through the WebSphere Everyplace authentication proxy.

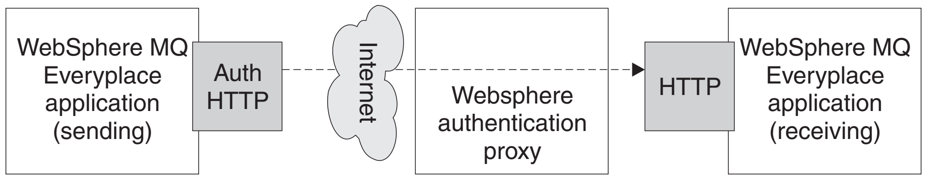

WebSphere MQ Everyplace application (sending)
Auth HTTP
Internet
Websphere authentication proxy
HTTP
WebSphere MQ Everyplace application (receiving)

Figure 68. Applications communicating through the WebSphere authentication proxy

The MQe adapter acts as the Auth HTTP adapter on the sending application. The receiving application could use either the same adapter or the standard HTTP adapter provided with MQe.

However, the real value of MQe is that it allows asynchronous messaging to occur in a typically synchronous environment. It is possible to gather enqueued requests from the receiving application and deal with them time-independently. The following diagram shows how incoming requests could be made to reach MQ servers asynchronously.

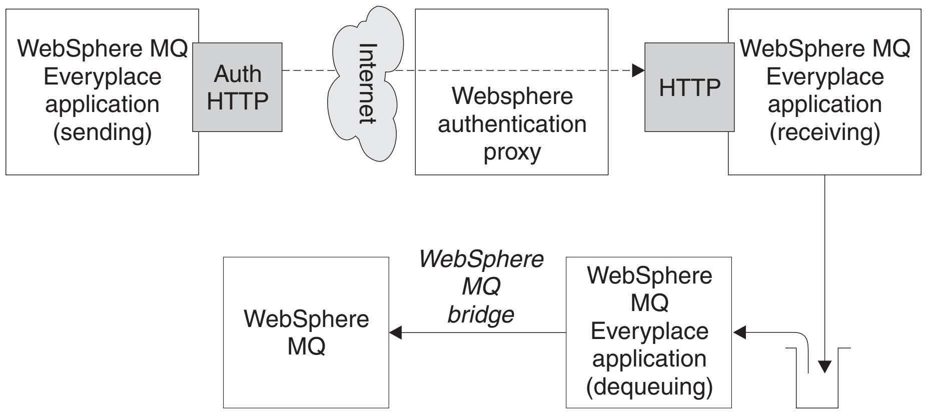

WebSphere MQ Everyplace application (sending)
Auth HTTP
Internet
Websphere authentication proxy
HTTP
WebSphere MQ Everyplace application (receiving)
WebSphere MQ
WebSphere MQ bridge
WebSphere MQ Everyplace application (dequeuing)

Figure 69. Applications communicating asynchronously through the WebSphere Authentication Proxy

In each of these environments the WebSphere authentication proxy is adding the ability to control access to the receiving applications. The adapter code supports this by adding (application-supplied) user ID and password information to each outgoing HTTP request. The WebSphere authentication proxy accepts these requests and verifies that the supplied credentials are valid for the current environment. If the credentials are valid the proxy forwards the request to the receiving application.

The WebSphere Everyplace Suite (WES) adapter files

In a standard MQe installation the WebSphere Everyplace adapter consists of, and is supported by the following files:

...\Java\com\ibm\mqe\adapters\MQeWESAuthenticationAdapter.class
- The WebSphere Everyplace adapter class.

...\Java\examples\application\Example7.class
- Compiled example application that uses the adapter

...\Java\examples\application\Example7.java
: - Source for the example application

...\Java\examples\adapters\WESAuthenticationGUIAdapter.class
: - Compiled example adapter that adds a user interface to the WebSphere Everyplace adapter. As with other example classes, this class is not meant as a replacement for the base WES adapter class, but rather as a demonstration of how to tailor the WES adapter to suit your requirements.

...\Java\examples\adapters\WESAuthenticationGUIAdapter.java
: - Source for the example adapter

If your environment *CLASSPATH* variable is set to find all classes within the MQe Java folder, the WebSphere Everyplace adapter class files will be accessible from within the Java environment. If the files are not accessible, issue a command such as:

```
set CLASSPATH=%CLASSPATH%;c:\mqe\java
```

This makes the new classes visible to Java. (The exact format of this command may vary from system to system.) Once this is complete you should be able to use the WebSphere Everyplace adapter classes in the same way as any other MQe classes.

Using the WebSphere Everyplace Suite (WES) adapter

This section provides information on how to use the WebSphere Everyplace adapter. The information is divided into three parts:

General operation
: This describes in detail, how to use the adapter in your applications

Using the Authentication Dialog Example
: This describes how to use an example class, examples.adapters.WESAuthenticationGUIAdapter. This class is derived from the base WES adapter class and provides a small user interface to collect the ID and password of the user.

Using the Application Example
: This describes how to use the supplied example file examples.application.Example7 which is configured to use the base WES adapter.

The information in this section assumes that both the WebSphere Everyplace authentication proxy and MQe have been installed and configured correctly. It is also assumed that an MQe server queue manager and an MQe client queue manager have been configured.

General operation:

1. Configure the client queue manager to send messages using the new adapter by modifying the client queue manager's configuration .ini file so that the *Network* alias points to com.ibm.mqe.adapters.MQeWESAuthenticationAdapter. Use the following command:
   ```
   (ascii)Network=com.ibm.mqe.adapters.MQeWESAuthenticationAdapter
   ```
2. Configure the server queue manager to decode the stream of data that the Client Adapter supplies using either the new adapter or the standard HTTP adapter. Do this by changing the line in the server queue manager's configuration .ini file so that the *Network* alias points to either com.ibm.mqe.adapters.MQeWESAuthenticationAdapter or com.ibm.mqe.adapters.MQeTcpipHttpAdapter. Use one of the following commands:
   ```
   (ascii)Network=com.ibm.mqe.adapters.MQeWESAuthenticationAdapter
   ```
   ```
   (ascii)Network=com.ibm.mqe.adapters.MQeTcpipHttpAdapter
   ```
3. Modify the client queue manager code so that the required user ID and password are set before the first network operation is started. For example, insert the following line near the top of your code:
   ```
   com.ibm.mqe.adapters.MQeWESAuthenticationAdapter.
   setBasicAuthorization("myUserId@myRealm", "myPassword");
   ```

Replace the parameters with a valid WES Server user ID and password.

You also need to add code to catch the new MQeException `Except_Authenticate` after each network operation, in case the supplied credentials were invalid.

4. Check that the client queue manager can still send messages to the server queue manager without going through the proxy.
5. Configure the client machine to send HTTP requests through the proxy. Depending on how WES has been configured, the adapter will need to work with either a *transparent proxy* or an *authentication proxy*.

As a *transparent proxy*

In this mode, the WES server acts as a simple HTTP proxy. In this case, you need to set the following Java application system properties that relate to proxy information:

http.proxyHost
Must be set to the host name of the WES proxy

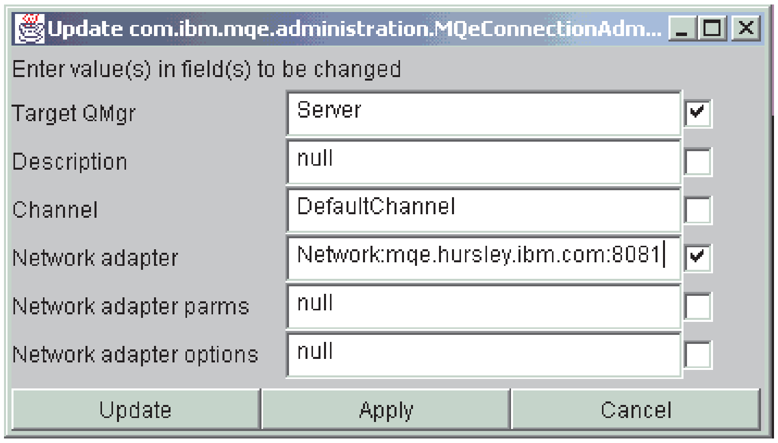
http.proxyPort
Must be set to the name of the port that the proxy is listening on

http.proxySet
Must be set to `true`, which tells the adapter to use transparent proxy mode

The above parameters can be set by adding the following to your Java application:

```
System.getProperties( ).put( "http.proxySet",  "true" );
System.getProperties( ).put( "http.proxyHost", "wes.hursley.ibm.com" );
System.getProperties( ).put( "http.proxyPort", "8082" );
```

The client queue manager's connection to the target MQe server is similar to a connection that doesn't use the WES proxy.



Enter value(s) in field(s) to be changed

Target QMgr	Server	✓
Description	null	
Channel	DefaultChannel	
Network adapter	Network:mqe.hursley.ibm.com:8081	✓
Network adapter parms	null	
Network adapter options	null	

Figure 70. Administration interface panel

You need to restart the server and client queue managers for the new settings to take effect. The client should then be able to send messages to the server through the proxy.

As an *Authentication Proxy*

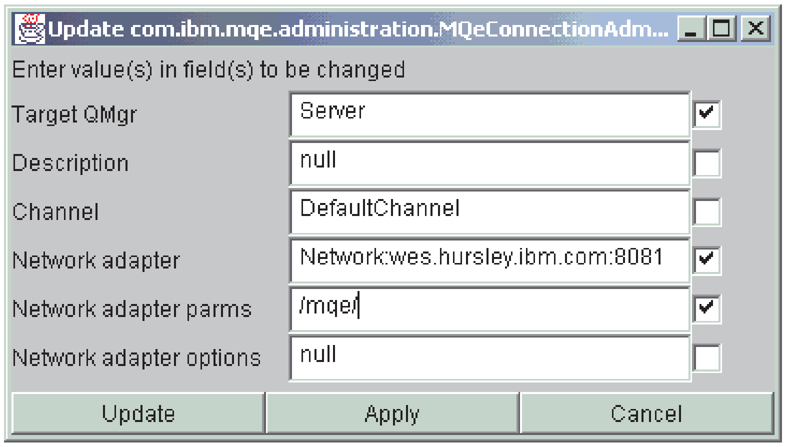
In this mode, the WES server forwards requests to services, based on the URL that you supply. For example, you may want requests for `http://wes.hursley.ibm.com/mqe` to be forwarded to an MQe queue manager running on `mqe.hursley.ibm.com:8082`.

To set this up from MQe you need to update the client's *`connection`* reference to the server.

Target network adapter
Should point to the Authentication Proxy machine and port

Network adapter parameters
Should contain the pathname to the required service

If you are using the MQe Example Administration tool, select **Connection** and then **Update** to configure this.


Update com.ibm.mqe.administration.MQeConnectionAdm...

Enter value(s) in field(s) to be changed

Target QMgr	Server	✓
Description	null	
Channel	DefaultChannel	
Network adapter	Network:wes.hursley.ibm.com:8081	✓
Network adapter parms	/mqe/	✓
Network adapter options	null	

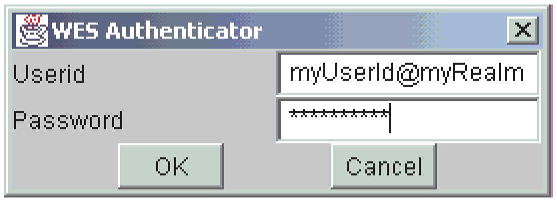
Figure 71. Administration interface panel

Note: The reference to the WES Server is entered in the **Network adapter** field, and the pathname is entered in the **Network adapter parms** field.
You need to restart the server and client queue managers for the new settings to take effect. The client should then be able to send messages to the server through the proxy.

Using the authentication dialog example:

The following information describes the use of the example class file, examples.adapters.WESAuthenticationGUIAdapter. This class adds a small user interface to the base WES adapter function.

1. Follow steps (1) and (2) of the “General operation” on page 117 procedures, but substitute ’WESAuthenticationGUIAdapter’ for ’WESAuthenticationAdapter’ in step (1).
2. Configure the client’s TCP/IP settings as in step (5) of ’General operation’.


WES Authenticator

Userid	myUserId@myRealm
Password	**********

Figure 72. WebSphere Everyplace Suite adapter user dialog

The client should now able to send messages to the server using the WESAuthenticationGUIAdapter. This adapter intercepts write calls to the WES adapter, and on the first request it pops up a dialog box that prompts for user ID and password information.

When the user clicks on **OK** or presses the **Enter** key, the setBasicAuthorization() method is called with the values from the **userid** and **password** fields. The write() is then forwarded on to the underlying WES adapter. The dialog box also has a Cancel button which, when selected, cancels the current write operation by not forwarding the request to the WES adapter. This causes an `MQeException (Except_Stopped)` to be thrown.

If authentication fails, the dialog box is redisplayed on the next write() along with any information provided by the server. In order to learn of an authentication failure, the example adapter intercepts read() calls and catches any `Except_Authenticate` MQeExceptions coming from the adapter.

Note: Web browsers do not generally send authentication information on the first flow. This typically results in a `401` or `407` response that contains the realm information. Only then does the browser send the authenticated request. User clients may wish to follow this convention.

Using the application example:

The following information describes the use of the example application file, examples.application.Example7. This example behaves in a similar way to the MQSeries® Everyplace programming example examples.application.Example1 and uses the basic WES adapter for communications.

1. Follow steps (1) and (2) of the "General operation" on page 117 procedures.
2. Configure the client's TCP/IP settings as in step (5) of "General operation" on page 117.
3. Edit the example file ...\Java\examples\application\Example7.java inserting a valid user ID and password, and then recompile the application.
4. Restart the server.
5. Run the Example7 program using the following command:

   ```
   java examples.application.Example7 Server client.ini
   ```

   where

   Server
   is the name of the remote queue manager (that the client already knows how to reach)

   client.ini
   points to the client's .ini configuration file.

   The application starts the client queue manager, authenticates with the proxy, puts a message to server and then gets a message from the server.

Using rules

Introduction to using MQe rules

MQe uses rules (which are essentially user exits) to allow applications to monitor and modify the behavior of some of its major components. Rules take the form of methods on Java classes or functions in C methods that are loaded when MQe components are initialized.

A component's rules are invoked at certain points during its execution cycle. Rules methods with particular signatures are expected to be available, so when providing implementations of rules, ensure that you use the correct signatures.

Default or example rules are provided for all relevant MQe components. You can customize these to satisfy particular user requirements. Within the Java code base, the `MQeQueueProxy` interface provides the user with accessor methods for queues, allowing the user to interact with queues in certain rule methods.

Rules may be grouped into the following categories:

- Queue manager rules.
- Queue rules.
- Attribute rules.
- Bridge rules.

Rules may also be categorized into two groups depending upon whether they can affect application behavior (modification rules) or are intended for notification purposes only (notification rules).

Queue manager rules

Queue manager rules are invoked when:

- The queue manager is activated
- The queue manager is closed
- A queue is added to the queue manager (Java code base only)
- A queue is removed from the queue manager (Java code base only)
- A put message operation occurs
- A get message operation occurs
- A delete message operation occurs
- An undo message operation occurs
- The queue manager is triggered to transmit any pending messages, as described in Transmission rules

Loading and activating queue manager rules

This topic describes how to load and activate queue manager rules in Java and C.

Java example queue manager rule:

Queue manager rules are loaded, or changed whenever a queue manager administration message containing a request to update the queue manager rule class is received.

If a queue manager rule has already been applied to the queue manager, the existing rule is asked whether it may be replaced with a different rule. If the answer is yes, the new rule is loaded and activated. A restart of the queue manager is not required.

The `QueueManagerUpdater` command-line tool in the package `examples.administration.commandline` shows how to create such an administration message.

C example queue manager rule:

The user's rules module is loaded and initialized when the queue manager is loaded into memory. This occurs as a result of calls either to `mqeAdministrator_QueueManager_create()` or to `mqeQueueManager_new()`. The setup steps are as follows:

- The application must register a rules alias, linking the rules alias to the rules module name and entry point, by using mqeClassAlias_add(), for example:

  ```
  #define RULES_ALIAS "myAlias"
      #define MODULE_NAME "myRulesModule.dll"
      #define ENTRY_POINT "myRules_new"
      ...

      mqeString_newUtf8(pExceptBlock,
                  &rulesAlias, RULES_ALIAS);
      mqeString_newUtf8(pExceptBlock,
                  &moduleName, MODULE_NAME);
      mqeString_newUtf8(pExceptBlock,
                  &entryPoint, ENTRY_POINT);
      mqeClassAlias_add(pExceptBlock,
                  rulesAlias, moduleName, entryPoint);
  ```

- The rules alias must be included in the queue manager start-up parameters passed to either `mqeAdministrator_QueueManager_create()` or `mqeQueueManager_new()`, for example.:

```
MQeQueueManagerParms      qmParams;
    qmParams.hQueueStore = msgStore; /* String parameters for the*/
                          /*location of the msg store */
    qmParams.hQueueManagerRules = rulesAlias; /* add in rules alias */


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

    ...

    rc = mqeAdministrator_QueueManager_create(hAdmin,pExceptBlock,
                             &hQM,qmName, &qmParms, &regParms);
```

- An initialization function or entry point must be supplied by the user. The following is an example of an initialization function for a rules implementation. The members of the parameter structures are documented in the MQe C Programming Reference.

```
MQERETURN myRules_new( MQeRulesNew_in_ * pInput,MQeRulesNew_out_ * pOutput) {

    MQERETURN rc = MQERETURN_OK;
    /* declare an instance of the private data */
   /*structure passed around between rules invocations. */
   /*This holds user data which is 'global' between rules. */
    myRules * myData = NULL;

    /* allocate the memory for the structure */
    myData = malloc(sizeof(myRules));
    if(myData != NULL)    {
   /* map user rules implementations to
      function pointers in output parameter structure */
        pOutput->fPtrActivateQMgr   = myRules_ActivateQMgr;
        pOutput->fPtrCloseQMgr      = myRules_CloseQMgr;
        pOutput->fPtrDeleteMessage  = unitTestRules_DeleteMessage;
        pOutput->fPtrGetMessage     = myRules_getMessage;
        pOutput->fPtrPutMessage     = myRules_putMessage;
        pOutput->fPtrTransmitQueue  = myRules_TransmitQueue;
        pOutput->fPtrTransmitQMgr   = myRules_TransmitQMgr;
        pOutput->fPtrActivateQueue  = myRules_activateQueue;
        pOutput->fPtrCloseQueue     = myRules_CloseQueue;
        pOutput->fPtrMessageExpired = myRules_messageExpired;

        /* initialize data in the private data structure */
        mydata->carryOn = MQE_TRUE;
        mydata->hAdmin = NULL;
        mydata->hThread = NULL;
        mydata->ifp = NULL;
        mydata->triggerInterval = 15000;

        /* now assign the private data structure to */
      /*the output parameter structure variable */
        pOutput->pPrivateData = (MQEVOID *)mydata;
    }
    else {
        /* We had a problem so clear up any strings in the structure -
            none in this case */
    }

    return rc;
}
```

The rules module is unloaded when the queue manager is freed. Note that, unlike the Java code base, the rules implementation is linked to the execution life cycle of a single queue manager and may not be replaced during the course of this life cycle.

Using queue manager rules

This topic describes some examples of the use of queue manager rules.

In the Java code base, a user provides an implementation of a rule method by subclassing the `MQeQueueManagerRule` class.

In the C code base, a user maps rules functions to relevant rules function pointers. These pointers are passed into the rules initialization function, which is also the entry point to the user's rules module.

For a description of all parameters passed to rules functions in the C code base, see the MQe C Programming Reference.

Example put message rule: This first example shows a put message rule that insists that any message being put to a queue using this queue manager must contain an MQe message ID field:

Java code base

```
/* Only allow msgs containing an ID field to be placed on the Queue */

public void putMessage( String destQMgr, String destQ, MQeMsgObject msg,
                      MQeAttribute attribute, long confirmId )        {
    if ( !(msg.Contains( MQe.Msg_MsgId )) )    {
          throw new MQeException( Except_Rule, "Msg must contain an ID" );
    }
}
```

C code base

```
  MQERETURN myRules_putMessage( MQeRulesPutMessage_in_ * pInput,
                              MQeRulesPutMessage_out_ * pOutput)    {
    // Only allow msgs containing an ID field to be placed on the Queue
    MQERETURN rc = MQERETURN_OK;
    MQEBOOL contains = MQE_FALSE;

    MQeExceptBlock * pExceptBlock=(MQeExceptBlock*)(pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

    rc = mqeFields_contains(pInput->hMsg,pExceptBlock,
                   &contains, MQE_MSG_MSGID);
    if(MQERETURN_OK == rc && !contains)     {
        SET_EXCEPT_BLOCK( pExceptBlock,
                          MQERETURN_RULES_DISALLOWED_BY_RULE,
                          MQEREASON_NA);
    }
}
```

Notice the manner in which the exception block instance is retrieved from the output parameter structure and then set with the appropriate return and reason codes. This is the way in which the rule function communicates with the application, thus modifying application behavior.

Example get message rule:

The next example rule is a get message rule that insists that a password must be supplied before allowing a get message request to be processed on the queue called OutboundQueue. The password is included as a field in the message filter passed into the `getMessage()` method.

Java code base

```
/* This rule only allows GETs from 'OutboundQueue',
   if a password is  */
/* supplied as part of the filter */

public void getMessage( String destQMgr,
                String destQ, MQeFields filter,
                     MQeAttribute attr, long confirmId )         {
   super.getMessage( destQMgr, destQ, filter, attr, confirmId );
   if (destQMgr.equals(Owner.GetName()
               && destQ.equals("OutboundQueue"))  {
       if ( !(filter.Contains( "Password" ) )      {
           throw new MQeException( Except_Rule,
                        "Password not supplied" );
       }
       else    {
           String pwd = filter.getAscii( "Password" );
           if ( !(pwd.equals( "1234" )) )   {
               throw new MQeException( Except_Rule,
                            "Incorrect password" );
           }
       }
   }
}
```

C code base

```
MQERETURN myRules_getMessage( MQeRulesGetMessage_in_ * pInput,
                              MQeRulesGetMessage_out_ * pOutput)    {
    MQeStringHndl hQueueManagerName, hCompareString, hCompareString2,
                  hFieldName, hFieldValue;
    MQEBOOL isEqual = MQE_FALSE;
    MQEBOOL contains = MQE_FALSE;
    MQeQueueManagerHndl hQueueManager;

    MQERETURN rc = MQERETURN_OK;
    MQeExceptBlock * pExceptBlock =
                        (MQeExceptBlock *)
                 (pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

    /* get the current queue manager */
     rc = mqeQueueManager_getCurrentQueueManager(pExceptBlock,
                                             &hQueueManager);
    if(MQERETURN_OK == rc) {
        // if the destination queue manager is the local queue manager
              rc = mqeQueueManager_getName( hQueueManager,
                                      pExceptBlock,
                                      &hQueueManagerName );
        if(MQERETURN_OK == rc)     {
            rc = mqeString_equalTo(pInput->hQueue_QueueManagerName,
                                   pExceptBlock,
                                   &isEqual,
                                   hQueueManagerName);
            if(MQERETURN_OK == rc && isEqual)    {
                // if the destination queue name is "OutboundQueue"
                rc = mqeString_newUtf8(pExceptBlock,
                                       &hCompareString,
                                       "OutboundQueue");
                rc = mqeString_equalTo(pInput->hQueueName,
                                       pExceptBlock,
                                       &isEqual,
                                       hCompareString);
                if(MQERETURN_OK == rc && isEqual)    {
                    // password required for this queue
                    MQEBOOL contains = MQE_FALSE;
                    rc = mqeString_newUtf8(pExceptBlock,
                                           &hFieldName,
                                           "Password");
                    rc = mqeFields_contains(pInput->hFilter,
                                            pExceptBlock,
                                            &contains,
                                            hFieldName);
                    if(MQERETURN_OK == rc && contains == MQE_FALSE)     {
                        SET_EXCEPT_BLOCK(pExceptBlock,
                                         MQERETURN_RULES_DISALLOWED_BY_RULE,
                                         MQEREASON_NA);
                    }
                    else   {
                            // parse password, etc.
                    }
                }
            }
        }
    }
}
```

This previous rule is a simple example of protecting a queue. However, for more comprehensive security, you are recommended to use an authenticator. An authenticator allows an application to create access control lists, and to determine who is able to get messages from queues.

Example remove queue rule:

The next example rule is called when a queue manager administration request tries to remove a queue. The rule is passed an object reference to the proxy for the queue in question. In this example, the rule checks the name of the queue that is passed, and if the queue is named PayrollQueue, the request to remove the queue is refused.

Java code base

```
/* This rule prevents the removal of the Payroll Queue */
public void removeQueue( MQeQueueProxy queue )
throws Exception    {
    if ( queue.getQueueName().equals( "PayrollQueue" ) )    {
        throw new MQeException( Except_Rule,
                    "Can't delete this queue" );
    }
}
```

C code base

This rule is not implemented in the C code base.

Transmission rules

A message that is put to a remote queue that is defined as synchronous is transmitted immediately. Messages put to remote queues defined as asynchronous are stored within the local queue manager until the queue manager is triggered into transmitting them. The queue manager can be triggered directly by an application. The process can be modified or monitored using the queue manager's transmission rules.

The transmission rules are a subset of the queue manager rules. The two rules that allow control over message transmission are:

`triggerTransmission()`

This rule determines whether to allow message transmission at the time when the rule is called. This can be used to veto or allow the transmission of all messages, that is, either all or none are allowed to be transmitted.

`transmit()`

This rule makes a decision to allow transmission on a per queue basis for asynchronous remote queues. For example, this makes it possible only to transmit the messages from queues deemed to be high priority. The `transmit()` rule is only called if the `triggerTransmission()` rule returns successfully.

Trigger transmission rule example

MQe calls the `triggerTransmission` rule when transmission is triggered. This occurs when the queue manager `triggerTransmission` method or function is explicitly called from an application or a rule.

Additionally, in the Java code base, the rule may be invoked when a message is put onto a remote asynchronous queue. The default rule behavior in both Java and C allows the attempt to transmit pending messages to proceed. For example, this is the default Java rule in com.ibm.mqe.MQeQueueManagerRule:

```
/* default trigger transmission rule -
    always allow transmission */
public boolean triggerTransmission(int noOfMsgs,
      MQeFields msgFields ){
    return true;
}
```

The return code from this rule tells the queue manager whether or not to transmit any pending messages. A return code of true means ″transmit″, while a return code of false means ″do not transmit at this time″.

The user may override the default behavior by implementing their own triggerTransmission() rule. A more complex rule can decide whether or not to transmit immediately based on the number of messages awaiting transmission on asynchronous remote queues. The following example shows a rule that only allows transmission to continue if there are more than 10 messages pending transmission.

Java code base

```
/* Decide to transmit based on number of pending messages */
public boolean triggerTransmission( int noOfMsgs, MQeFields msgFields ) {
    if(noOfMsgs > 10)    {
        return true; /* then transmit */
    }
    else {
        return false; /* else do not transmit */
    }
}
```

C code base

```
/* The following function is mapped to the
    fPtrTransmitQMgr function pointer  */
/* in the user's initialization function output parameter structure. */

MQERETURN myRules_TransmitQMgr( MQeRulesTransmitQMgr_in_  * pInput,
                        MQeRulesTransmitQMgr_out_ * pOutput)    {
    MQeExceptBlock * pExceptBlock =
                        (MQeExceptBlock*)(pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

   /* allow transmission to be triggered only
      if the number of pending messages > 10  */
    if(pInput->msgsPendingTransmission <= 10) {
        SET_EXCEPT_BLOCK(pExceptBlock,
                     MQERETURN_RULES_DISALLOWED_BY_RULE,
        MQEREASON_NA);
    }
}
```

Transmit rule

The `transmit()` rule is only called if the `triggerTransmission()` rule allows transmission. It returns a value of true or `MQERETURN_OK`. The `transmit()` rule is called for every remote queue definition that holds messages awaiting transmission. This means that the rule can decide which messages should be transmitted on a queue by queue basis.

A sensible extension to this rule can allow all messages to be transmitted at ′off-peak′ time. This allows only messages from high-priority queues to be transmitted during peak periods.

Transmit rule - Java example 1:

The example rule below only allows message transmission from a queue if the queue has a default priority greater than 5. If a message has not been assigned a priority before being placed on a queue, it is given the queue's default priority.

```
public boolean transmit( MQeQueueProxy queue )    {
    if ( queue.getDefaultPriority() > 5 )    {
        return (true);
    }
    else    {
        return (false);
    }
}
```

Transmit rule - C example 1:

The example rule below only allows message transmission from a queue if the queue has a default priority greater than 5. If a message has not been assigned a priority before being placed on a queue, it is given the queue's default priority.

```
/* The following function is mapped to the fPtrTransmitQueue function*/
/* pointer in the user's initialization
/* function output parameter structure. */

MQERETURN myRules_TransmitQueue( MQeRulesTransmitQueue_in_ * pInput,
                        MQeRulesTransmitQueue_out_ * pOutput) {
    MQERETURN rc = MQERETURN_OK;
    MQEBYTE queuePriority;

    MQeRemoteAsyncQParms queueParms = REMOTE_ASYNC_Q_INIT_VAL;
    myRules * myData = (myRules *)(pInput->pPrivateData);

    MQeExceptBlock * pExceptBlock =
                   (MQeExceptBlock *)(pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

    /* inquire upon the default priority of the queue*/
    /* specify the subject of the inquire
      in the queue parameter structure*/
   queueParms.baseParms.opFlags =  QUEUE_PRIORITY_OP ;

    rc =  mqeAdministrator_AsyncRemoteQueue_inquire(myData->hAdmin,
                                   pExceptBlock,
                                   pInput->hQueueName,
                                   pInput->hQueue_QueueManagerName,
                                   &queueParms);
    // if the default priority is less than 6, disallow the operation
    if(MQERETURN_OK == rc
      && queueParms.baseParms.queuePriority < 6) {
        SET_EXCEPT_BLOCK(pExceptBlock,
                         MQERETURN_RULES_DISALLOWED_BY_RULE,
                  MQEREASON_NA);
    }
}
```

A more complex transmit rule example

The following example (in Java and in C) assumes that the transmission of the messages takes place over a communications network that charges for the time taken for transmission. It also assumes that there is a cheap-rate period when the unit-time cost is lower. The rules block any transmission of messages until the cheap-rate period. During the cheap-rate period, the queue manager is triggered at regular intervals.

Transmit rule - Java example 2:

The following example assumes that the transmission of the messages takes place over a communications network that charges for the time taken for transmission. It also assumes that there is a cheap-rate period when the unit-time cost is lower. The rules block any transmission of messages until the cheap-rate period. During the cheap-rate period, the queue manager is triggered at regular intervals.

```
import com.ibm.mqe.*;
import java.util.*;
/**
* Example set of queue manager
    rules which trigger the transmission
* of any messages waiting to be sent.
*
* These rules only trigger the
    transmission of messages if the current
* time is between the values defined
    in the variables cheapRatePeriodStart
* and cheapRatePeriodEnd
* (This example assumes that transmission
      will take place over a
* communication network which charges
      for the time taken to transmit)
*/
public class ExampleQueueManagerRules extends MQeQueueManagerRule
implements Runnable
{
    // default interval between triggers is 15 seconds
    private static final long
          MILLISECS_BETWEEN_TRIGGER_TRANSMITS = 15000;

    // interval between which we c
      heck whether the queue manager is closing down.
        private static final long
          MILLISECS_BETWEEN_CLOSE_CHECKS = 1000 ;

    // Max wait of ten seconds to kil off
        the background thread when
    // the queue manager is closing down.
    private static final long
        MAX_WAIT_FOR_BACKGROUND_THREAD_MILLISECONDS = 10000;

    // Reference to the control block used to
        communicate with the background thread
    // which does a sleep-trigger-sleep-trigger loop.
    // Note that freeing such blocks for garbage
        collection will not stop the thread
    // to which it refers.
    private Thread th = null;

    // Flag which is set when shutdown of
        the background thread is required.
    // Volatile because the thread using the
        flag and the thread setting it to true
    // are different threads, and it is
        important that the flag is not held in
    // CPU registers, or one thread will
        see a different value to the other.
    private volatile boolean toldToStop = false;
   //cheap rate transmission period start and end times
    protected int cheapRatePeriodStart = 18;  /*18:00 hrs */
    protected int cheapRatePeriodEnd = 9;     /*09:00 hrs */
}
```

The `cheapRatePeriodStart` and `cheapRatePeriodEnd` functions define the extent of this cheap rate period. In this example, the cheap-rate period is defined as being between 18:00 hours in the evening until 09:00 hours the following morning.

The constant `MILLISECS_BETWEEN_TRIGGER_TRANSMITS` defines the period of time, in milliseconds, between each triggering of the queue manager. In this example, the trigger interval is defined to be 15 seconds.

The triggering of the queue manager is handled by a background thread that wakes up at the end of the triggerInterval period. If the current time is inside the cheap rate period, it calls the `MQeQueueManager.triggerTransmission()` method to initiate an attempt to transmit all messages awaiting transmission. The background thread is created in the `queueManagerActivate()` rule and stopped in the `queueManagerClose()` rule. The queue manager calls these rules when it is activated and closed respectively.

```
/**
* Overrides MQeQueueManagerRule.queueManagerActivate()
* Starts a timer thread
*/
public void queueManagerActivate()throws Exception {
    super.queueManagerActivate();
    // background thread which triggers transmission
    th = new Thread(this, "TriggerThread");
    toldToStop = false;
    th.start();    // start timer thread
}


/**
* Overrides MQeQueueManagerRule.queueManagerClose()
* Stops the timer thread
*/
 public void queueManagerClose()throws Exception {
    super.queueManagerClose();

    // Tell the background thread to stop,
      as the queue manager is closing now.
    toldToStop = true ;

    // Now wait for the background thread,
      if it's not already stopped.
    if ( th != null)  {
        try {
        // Only wait for a certain time before
        giving up and timing out.
         th.join( MAX_WAIT_FOR_BACKGROUND_THREAD_MILLISECONDS );

         // Free up the thread control block for garbage collection.
            th = null ;
        } catch (InterruptedException e) {
            // Don't propogate the exception.
            // Assume that the thread will stop shortly anyway.
        }
    }
}
```

The code to handle the background thread looks like this:

```
/**
* Timer thread
* Triggers queue manager every interval until thread is stopped
*/
public void run()    {
    /* Do a sleep-trigger-sleep-trigger loop until the */
   /*  queue manager closes or we get an exception.*/
    while ( !toldToStop) {
        try {

            // Count down until we've waited enough
          // We do a tight loop with a smaller granularity because
          // otherwise we would stop a queue manager from closing quickly
          long timeToWait = MILLISECS_BETWEEN_TRIGGER_TRANSMITS ;
          while( timeToWait > 0 && !toldToStop ) {

            // sleep for specified interval
```

```
                Thread.sleep( MILLISECS_BETWEEN_CLOSE_CHECKS );

                // We've waited for some time.
              Account for this in the overall wait.
                timeToWait -= MILLISECS_BETWEEN_CLOSE_CHECKS ;
          }
   if( !toldToStop && timeToTransmit()) {
            // trigger transmission on QMgr (which is rule owner)
            ((MQeQueueManager)owner).triggerTransmission();
                      }
          } catch ( Exception e ) {
              e.printStackTrace();
          }
        }
    }
}
```

The variable owner is defined by the class `MQeRule`, which is the ancestor of `MQeQueueManagerRule`. As part of its startup process, the queue manager activates the queue manager rules and passes a reference to itself to the rules object. This reference is stored in the variable owner.

The thread loops indefinitely, as it is stopped by the `queueManagerClose()` rule, and it sleeps until the end of the `MILLISECS_BETWEEN_TRIGGER_TRANSMITS` interval period. At the end of this interval, if it has not been told to stop, it calls the `timeToTransmit()` method to check if the current time is in the cheap-rate transmission period. If this method succeeds, the queue manager's `triggerTransmission()` rule is called. The `timeToTransmit` method is shown in the following code:

```
protected boolean timeToTransmit()    {
    /* get current time */
    Calendar calendar = Calendar.getInstance();
    calendar.setTime( new Date() );
    /* get hour */
    int hour = calendar.get( Calendar.HOUR_OF_DAY );
    if ( hour >= cheapRatePeriodStart || hour
            < cheapRatePeriodEnd ) {
        return true; /* cheap rate */
    }
    else    {
        return false; /* not cheap rate */
    }
}
```

Transmit rule - C example 2:

The C example emulates the Java example. While the native C code base is entirely single-threaded, it is possible to write platform-specific code in which threads are created. In this example of a user-written queue manager activate rule, a thread is spawned which loops, sleeping for a period of time defined in a `triggerInterval` variable and then, providing it has not been asked to stop, checking that we are in a cheap rate period prior to attempting to trigger transmission. Data, which is required between rules invocations, is stored in the rule's private data structure. The queue manager's close rule function is used to provide the thread's terminating condition, setting a boolean switch, `carryOn` to `MQE_FALSE`. This switch can be initialized to `MQE_TRUE` in the rules initialization function. This function waits until the thread is suspended before passing control back to the application.

The private data structure passed between rule invocations is as follows:

```
struct myRules_st_ {
// rules instance structure
    MQeAdministratorHndl hAdmin;
// administrator handle to carry around between

// rules functions
    MQEBOOL carryOn;
// used for trigger transmission thread
```

```
    MQEINT32 triggerInterval;
// used for trigger transmission thread
    HANDLE hThread;
// handle for the trigger transmission thread
};

typedef struct myRules_st_ myRules;

The queue manager activate rule:

MQEVOID myRules_activateQueueManager( MQeRulesActivateQMgr_in_ * pInput,
                                    MQeRulesActivateQMgr_out_ * pOutput) {
    // retrieve exception block - passed from application
    MQeExceptBlock * pExceptBlock = (MQeExceptBlock *)
                          (pOutput->pExceptBlock);

    // retrieve private data structure passed
      between user's rules invocations
    myRules * myData = (myRules *)(pInput->pPrivateData);

    MQeQueueManagerHndl hQueueManager;
    MQERETURN rc = MQERETURN_OK;

    rc = mqeQueueManager_getCurrentQueueManager(pExceptBlock,
                                &queueManager);
    if(MQERETURN_OK == rc) {
        // set up the private data administrator
          handle using the retrieved
        // application queue manager handle.
          This is done here rather than in
        // the rules initialization function as the
          queue manager has not yet been
      // activated fully when the rules
      //initialization function is invoked.
        rc = mqeAdministrator_new(pExceptBlock,
                    &myData>hAdmin,hQueueManager);
    }
    if(MQERETURN_OK == rc) {
        DWORD tid;
        // Launch thread to govern calls to trigger transmission
        myData->hThread = (HANDLE) CreateThread(NULL,
                                            0,
                                            timeToTrigger,
                                            (MQEVOID *)myData,
                                            0,
                                            &tId);
                                if(myData>hThread == NULL) {
            // thread creation failed
            SET_EXCEPT_BLOCK(pExceptBlock,
                      MQERETURN_RULES_ERROR,
                      MQEREASON_NA);
        }
    }
}
```

The `timeToTrigger` function provides the equivalent functionality of the `run()` method in the Java example. Notice the use of the private data variable `carryOn`, type `MQEBOOL`, as one of the conditions for the while loop to continue. Once this variable has a value of `MQE_FALSE`, the while loop will terminate, causing the thread to terminate when the function is exited.

```
DWORD _stdcall timeToTrigger(myRules * rulesStruct) {

    MQERETURN rc = MQERETURN_OK;
    MQeQueueManagerHndl hQueueManager;
    MQeExceptBlock exceptBlock;
    myRules * myData = (myRules *)rulesStruct;
    SET_EXCEPT_BLOCK_TO_DEFAULT(&exceptBlock);
    /* retrieve the current queue manager */
    rc = mqeQueueManager_getCurrentQueueManager(&exceptBlock,
                                 &hQueueManager);
    if(MQERETURN_OK == rc) {
        /* so long as there is not a grave
        internal error and the termination
           condition has not been set */
        while(!(EC(&exceptBlock) ==
                MQERETURN_QUEUE_MANAGER_ERROR &&
                ERC(&exceptBlock) ==
                MQEREASON_INTERNAL_ERROR) &&
                myData->carryOn == MQE_TRUE) {
           /* Are we in a cheap rate transmission period? */
            if(timeToTransmit())    {
                /* if so,  attempt to trigger transmission */
                rc = mqeQueueManager_triggerTransmission(hQueueManager,
                                        &exceptBlock);

                /* wait for the duration of the trigger interval */
                Sleep(myData->triggerInterval);
            }
        }
    }
    return 0;
}
```

The `timeToTransmit()` function returns a boolean to indicate whether or not we are in a cheap transmission period:

```
MQEBOOL timeToTransmit() {

    SYSTEMTIME timeInfo;
    GetLocalTime(&timeInfo);

    if (timeInfo.wHour >= 18 || timeInfo.wHour < 9) {
        return MQE_TRUE;
    } else {
        return MQE_FALSE;
    }
}
```

It would probably be a better idea to define constants for the cheap rate interval boundary times and carry these around in the rules private data structure also but that has been not been done here for reasons of clarity.

The function returns `MQE_TRUE` to suggest that we are in a cheap rate period, that is between the hours of 18:00 and 09:00. A return value of `MQE_TRUE` is one of the prerequisites for transmission to be triggered in `timeToTrigger()`. Finally, the queue manager close rule is used to terminate the thread. Notice that one of the conditions for termination of the `timeToTrigger()` function is for the boolean variable carryOn to have a value of `MQE_FALSE`. In the close function, the value of carryOn is set to false. But, there may still be a considerable lapse of time between when this value is set to `MQE_FALSE` and when the `timeToTrigger()` function is exited. The value of triggerInterval + the time taken to perform a `triggerTransmission` operation. Also, we wait for the thread to terminate in this function. We also call `triggerTransmission()` one more time in case there are still some pending messages.

```
  MQEVOID myRules_CloseQMgr( MQeRulesCloseQMgr_in_ * pInput,
                          MQeRulesCloseQMgr_out_ * pOutput)     {
    MQERETURN rc = MQERETURN_OK;
    MQeQueueManagerHndl hQueueManager;
    myRules * myData = (myRules *)pInput->pPrivateData;
    DWORD result;
    MQeExceptBlock exceptBlock =
            *((MQeExceptBlock *)pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(&exceptBlock);
    // Effect the ending of the thread by
        setting the MQEBOOL continue to MQE_FALSE
    // This leads to a return from timeToTrigger()
          and hence the implicit call
    // to _endthread
    myData->carryOn = MQE_FALSE;

    /* wait for the thread in any case */
    result = WaitForSingleObject(myData->hThread, INFINITE);

    /* retrieve the current queue manager */
    rc = mqeQueueManager_getCurrentQueueManager(&exceptBlock,
                                 &hQueueManager);
    if(MQERETURN_OK == rc) {
        /* attempt to trigger transmission one
      /* last time to clean up queue */
        rc = mqeQueueManager_triggerTransmission(hQueueManager,
                                 &exceptBlock);
    }
}
```

Activating asynchronous remote queue definitions

The queue manager can activate its asynchronous remote queue definitions and home server queues at startup time. In the Java code base, activating asynchronous remote queue definitions results in an attempt to transmit any messages they contain, while activating home server queues results in an attempt to get any messages that are waiting on their assigned store-and-forward queue. The activateQueues() rule allows this behavior to be configured.

The default rule just returns true.

```
public boolean activateQueues()    {
    return true; /* activate queues on queue manager start-up */
}

/*As with other rules examples above,
    a check can be made to see if the current */
/* time is inside the cheap-rate transmission period.
    This information can then */
/* be used to determine whether queues should be activated or not.

public boolean activateQueues()    {
    if ( timeToTransmit() )    {
        return true;
    }
    else    {
        return false;
    }
}
```

If `activateQueues()` returns false, the remote queue definitions are only activated when a message is put onto them. Home server queues can be activated by calling the queue manager's `triggerTransmission()` method.

In the C code base, activation of home server queues and asynchronous queues does not result in any attempts to transmit or pull down pending messages. Only explicit calls to the queue manager's `triggerTransmission()` function have this result. There is no implementation of an activateQueues rule in the C code base. Activation of queues occurs at queue manager startup.

Queue rules

In the Java code base, each queue has its own set of rules. A solution can extend the behavior of these rules. All queue rules should descend from class com.ibm.mqe.MQeQueueRule.

In the C code base, only a single set of rules is loaded. A user can implement different rules for different queues by loading other rules modules from the ’master’ module. The master rules functions can then invoke the corresponding functions in any other modules as required.

Queue rules are called when:

- The queue is activated.
- The queue is closed.
- A message is placed on the queue using a put operation (Java code base only).
- A message is removed from the queue using a get operation.
- A message is deleted from the queue using a delete operation (Java code base only).
- The queue is browsed.
- An undo operation is performed on a message on the queue.
- A message listener is added to the queue (Java code base only).
- A message listener is removed from the queue (Java code base only).
- A message expires.
- An attempt is made to change a queue’s attributes, that is authenticator, cryptor, compressor (Java code base only).
- A duplicate message is put onto a queue.
- A message is being transmitted from a remote asynchronous queue.

Using queue rules

This section describes some examples of the use of queue rules.

The first example shows a possible use of the message expired rule, putting a copy of the message onto a Dead Letter Queue. Both queues and messages can have an expiry interval set. If this interval is exceeded, the message is flagged as being expired. At this point the messageExpired() rule is called. On return from this rule, the expired message is deleted.

The first example sends any expired messages to the queue manager’s dead-letter queue, the name of which is defined by the constant `MQe.DeadLetter_Queue_Name` in the Java code base and `MQE_DEADLETTER_QUEUE_NAME` in the C code base. The queue manager rejects a put of a message that has previously been put onto another queue. This protects against a duplicate message being introduced into the MQe network. So, before moving the message to the dead-letter queue, the rule must set the resend flag. This is done by adding the Java `MQe.Msg_Resend` or C `MQE_MSG_RESEND` field to the message.

The message expiry time field must be deleted before moving the message to the dead-letter queue.

Queue rules - Java example 1:

This example shows a possible use of the message expired rule, and a copy of the message is put onto a Dead Letter Queue. Both queues and messages can have an expiry interval set. If this interval is exceeded, the message is flagged as being expired. At this point the messageExpired() rule is called. On return from this rule, the expired message is deleted.

```
/* This rule puts a copy of any expired messages to a Dead Letter Queue */

public boolean messageExpired( MQeFields entry, MQeMsgObject msg )
           throws Exception   {

    /* Get the reference to the Queue Manager */
    MQeQueueManager qmgr = MQeQueueManager.getReference(
                          ((MQeQueueProxy)owner).getQueueManagerName());
    /* need to set re-send flag so that put of message
      to new queue isn't rejected */
    msg.putBoolean( MQe.Msg_Resend, true );
```

```
     /* if the message contains an expiry
        interval field - remove it */
    if ( msg.contains( MQe.Msg_ExpireTime )    {
        msg.delete( MQe.Msg_ExpireTime );
    }
    /* put message onto dead letter queue */
    qmgr.putMessage( null, MQe.DeadLetter_Queue_Name,
                msg, null, 0 );
    /* Return true. Note that no use is made
        of this return value - the message is
      always deleted but the return value is kept
      for backward compatibility */
    return (true);
}
```

Queue rules - C example 1:

This example shows a possible use of the message expired rule, and a copy of the message is put onto a Dead Letter Queue. Both queues and messages can have an expiry interval set. If this interval is exceeded, the message is flagged as being expired. At this point the messageExpired() rule is called. On return from this rule, the expired message is deleted.

```
MQEVOID myRules_messageExpired( MQeRulesMessageExpired_in_ * pInput,
                          MQeRulesMessageExpired_out_ * pOutput)  {
    MQERETURN rc = MQERETURN_OK;
    MQeExceptBlock * pExceptBlock =
              (MQeExceptBlock *)(pOutput->pExceptBlock);

    MQEBOOL contains = MQE_FALSE;
    MQeFieldsHndl hMsg;
    MQeQueueManagerHndl hQueueManager;
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

    /* Set re-send flag so that attempt to put
      message to new queue isn't rejected */
    // First, clone the message as the
   //input parameter is read-only
    rc = mqeFields_clone(pInput->hMsg, pExceptBlock,
                  &hMsg);
    if(MQERETURN_OK == rc) {
        rc = mqeFields_putBoolean(hMsg, pExceptBlock,
                              MQE_MSG_RESEND, MQE_TRUE);
        if(MQERETURN_OK == rc) {
            // if the message contains an expiry
            interval field - remove it
            rc = mqeFields_contains(hMsg, pExceptBlock,
                          &contains,
                                   MQE_MSG_EXPIRETIME);
            if(MQERETURN_OK == rc && contains) {
                rc = mqeFields_delete(hMsg, pExceptBlock,
                                     MQE_MSG_EXPIRETIME);
            }
if(MQERETURN_OK == rc) {
                // put message onto dead letter queue
                MQeStringHndl hQueueManagerName;
                rc = mqeQueueManager_getCurrentQueueManager(pExceptBlock,
                                                    &hQueueManager);
                if(MQERETURN_OK == rc) {
            rc = mqeQueueManager_getName(hQueueManager,
                                              pExceptBlock,
                                             &hQueueManagerName);
                    if(MQERETURN_OK == rc) {
                        // use a temporary exception block as don't care
                        // if dead letter queue does not exist
                        MQeExceptBlock tempExceptBlock;
                        SET_EXCEPT_BLOCK_TO_DEFAULT(&tempExceptBlock);
                        rc = mqeQueueManager_putMessage( hQueueManager,
                                                   &tempExceptBlock,
                                                   hQueueManagerName,
                                              MQE_DEADLETTER_QUEUE_NAME,
                                                    hMsg, NULL, 0 );
                       (MQEVOID)mqeString_free(hQueueManagerName,
                                  &tempExceptBlock);
                    }
                }
            }
        }
    }
}
```

Queue rules - Java example 2:

The following example shows how to log an event that occurs on the queue. The event that occurs is the creation of a message listener.

In the example, the queue has its own log file, but it is equally as valid to have a central log file that is used by all queues. The queue needs to open the log file when it is activated, and close the log file when the queue is closed. The queue rules, `queueActivate` and `queueClose` can be used to do this. The variable `logFile` needs to be a class variable so that both rules can access the log file.

```
/* This rule logs the activation of the queue */
public void queueActivate()    {
    try    {
     logFile = new LogToDiskFile( \\log.txt );
     log( MQe_Log_Information, Event_Activate, "Queue " +
     ((MQeQueueProxy)owner).getQueueManagerName() + " + " +
     ((MQeQueueProxy)owner).getQueueName() + " active" );
    }
    catch( Exception e )    {
        e.printStackTrace( System.err );
    }
}

/* This rule logs the closure of the queue */
public void queueClose()    {
    try    {
      log( MQe_Log_Information, Event_Closed, "Queue " +
          ((MQeQueueProxy)owner).getQueueManagerName() + " + " +
          ((MQeQueueProxy)owner).getQueueName() + " closed" );
        /* close log file */
        logFile.close();
    }
    catch ( Exception e )    {
        e.printStackTrace( System.err );
    }
}
```

The `addListener` rule is shown in the following code. It uses the `MQe.log` method to add an `Event_Queue_AddMsgListener` event.

```
/* This rule logs the addition of a message listener */
public void addListener( MQeMessageListenerInterface listener,
                  MQeFields filter ) throws Exception
   {
    log( MQe_Log_Information, Event_Queue_AddMsgListener,
               "Added listener on queue "
         + ((MQeQueueProxy)owner).getQueueManagerName() + "+"
         + ((MQeQueueProxy)owner).getQueueName() );
}
```

Queue rules - C example 2:

The following example shows how to log an event that occurs on the queue. The event that occurs is a put message request.

In this example, a central log is set up for all queues using the queue activate and close rules. This log is then used to keep track of all putMessage operations. Because the log is shared between rules invocations, the information needed to access the log is stored in the rules private data structure. In this case, the private data structure contains a file handle for passing between rules invocations:

```
struct myRulesData_  {
// rules instance structure
      MQeAdministratorHndl hAdmin;  /
 administrator handle to carry around between
// rules functions
      FILE * ifp;
// file handle for logging rules
};
typedef struct myRulesData_ myRules;
```

In the rules queue activate function, the file is opened and the activation of the queue logged:

```
MQEVOID myRules_activateQueue(MQeRulesActivateQueue_in_ * pInput,
                              MQeRulesActivateQueue_out_ * pOutput)  {
    MQERETURN rc = MQERETURN_OK;
    MQECHAR * qName;
    MQEINT32 size;

    // recover the private data from the input
        structure parameter pInput
    myRules * myData = (myRules *)(pInput->pPrivateData);

    MQeExceptBlock * pExceptBlock =
                (MQeExceptBlock *)(pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

    if(myData->ifp == NULL)   {
    // initialized to NULL in the rules initialization function
        myData->ifp = fopen("traceFile.txt","w");
        rc =  mqeString_getUtf8(pInput->hQueueName,
                        pExceptBlock, NULL, &size);
        if(MQERETURN_OK == rc) {
             qName = malloc(size);
             rc = mqeString_getUtf8(pInput->hQueueName,
                            pExceptBlock, qName, &size);
             if(MQERETURN_OK ==
              rc && myData->ifp != NULL) {
                 fprintf(myData->ifp,
            "Activating queue %s \n", qName);
             }
        }
    }
}
```

In the rules queue close function, the file is closed after the closure of the queue is logged:

```
MQEVOID myRules_closeQueue(MQeRulesCloseQueue_in_ * pInput,
                           MQeRulesCloseQueue_out_ * pOutput)  {
    MQERETURN rc = MQERETURN_OK;
    MQECHAR * qName;
    MQEINT32 size;

    // recover the private data from the
      input structure parameter pInput
    myRules * myData = (myRules *)(pInput->pPrivateData);

    MQeExceptBlock * pExceptBlock =
          (MQeExceptBlock *)(pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);
    if(myData->ifp != NULL)   {
        rc =  mqeString_getUtf8(pInput->hQueueName,
                pExceptBlock, NULL, &size);
        if(MQERETURN_OK == rc) {
             qName = malloc(size);
             rc = mqeString_getUtf8(pInput->hQueueName,
                  pExceptBlock, qName, &size);
             if(MQERETURN_OK == rc)  {
                 fprintf(myData->ifp,
              "Closing queue %s \n", qName);
             }
        }
        fclose(myData->ifp);
        MyData->ifp = NULL;
    }
}
```

The rules put message function ensures that each put message operation is logged:

```
MQERETURN myRules_putMessage(MQeRulesPutMessage_in_ * pInput,
                             MQeRulesPutMessage_out_ * pOutput)    {
    MQERETURN rc = MQERETURN_OK;
    MQECHAR * qName, * qMgrName;
    MQEINT32 size;

    // recover the private data from the input structure parameter pInput
    myRules * myData = (myRules *)(pInput->pPrivateData);

    MQeExceptBlock * pExceptBlock =
(MQeExceptBlock *)(pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

    if(myData->ifp != NULL)   {
        rc =  mqeString_getUtf8(pInput->hQueueName,
                        pExceptBlock, NULL, &size);
        if(MQERETURN_OK == rc) {
             qName = malloc(size);
             rc = mqeString_getUtf8(pInput->hQueueName,
                          pExceptBlock, qName,&size);
        }
        if(MQERETURN_OK == rc)   {
            rc =  mqeString_getUtf8(pInput->hQueue_QueueManagerName,
                          pExceptBlock,
                                      NULL, &size);
            if(MQERETURN_OK == rc) {
                qMgrName = malloc(size);
                rc = mqeString_getUtf8(pInput->hQueue_QueueManagerName,
                                         pExceptBlock,
                            qMgrName, &size);
            }
        }
if(MQERETURN_OK == rc)  {
                 fprintf(myData->ifp, "Putting a message
                  onto queue %s on queue
                         manager %s\n",qName, qMgrName);
        }
    }
    /* allow the operation to proceed regardless of what
      went wrong in this rule */
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);
    return EC(pExceptBlock);
}
```

Bridge rules

Whilst Queue Rules can also be applied to Bridge Queues, you can also apply the following other types of rules to the Bridge:

UndeliveredMessageRules
: These rules can be applied to the Bridge Listener and can be used to determine what action is to be performed when an MQ Message can't be delivered to the MQe Gateway. The default rule used by MQe will stop the Bridge Listener after a set number of attempts to deliver the message. Two example rules are provided:

  examples.mqbridge.rules.MQeUndeliveredMessageRule
  : Copy of the default rule

  examples.mqbridge.rules.UndeliveredMQMessageToDLQRule
  : Will either discard the message or move it to MQ's Dead Letter Queue depending on the report field of the original MQ Message

StartUp Rules
: These rules can be used to control startup of the objects held in the bridge so that, for example, the bridge is in a stopped state when the MQe Gateway is started. An example is provided: examples.mqbridge.rules.MQeStartupRule.

SyncQueuePurger Rules
: These rules can be used for administrative purposes to clear up old records that can sometimes be left on the MQ Queue manager. However, this typically only occurs if the corresponding MQe message has been deleted. Two examples are provided:

  examples.mqbridge.rules.MQeSyncQueuePurgerRule
  : Calls trace with an info statement when it discovers messages older than a specified time

  examples.mqbridge.rules.DestructiveMQSyncQueuePurgerRule
  : Deletes any message that is older than a specified time

Java Message Service (JMS)

The MQe classes for Java Message Service (JMS) are a set of Java classes that implement the Sun JMS interfaces to enable JMS programs to access MQe systems. This topic describes how to use the MQe classes for JMS.

The initial release of JMS classes for MQe Version 2.1, supports the point-to-point model of JMS, but does not support the publish or subscribe model.

The use of JMS as the API to write MQe applications has a number of benefits, because JMS is open standard:
- The protection of investment, both in skills and application code
- The availability of people skilled in JMS application programming
- The ability to write messaging applications that are independent of the JMS implementations

More information about the benefits of the JMS API is on Sun's Web site at http://java.sun.com.

Using JMS with MQe

This section describes how to set up your system to run the example programs, including the Installation Verification Test (IVT) example which verifies your MQe JMS installation.

To use JMS with MQe you must have the following jar files, in addition to MQeBase.jar, on your class path:

jms.jar
This is Sun’s interface definition for the JMS classes

MQeJMS.jar
This is the MQe implementation of JMS

Obtaining jar files

MQe does not ship with Sun’s JMS interface definition, which is contained in jms.jar, and this must be downloaded before JMS can be used. At the time of writing, this can be freely downloaded fromhttp://java.sun.com/products/jms/docs.htmlThe JMS Version 1.0.2b jar file is required.

In addition, if JMS administered objects are to be stored and retrieved using the Java Naming and Directory Interface (JNDI), the javax.naming.* classes must be on the classpath. If Java 1 is being used, for example, a 1.1.8 JRE, jndi.jar must be obtained and added to the classpath. If Java 2 is being used, a 1.2 or later JRE, the JRE might contain these classes. You can use MQe without JNDI, but at the cost of a small degree of provider dependence.MQe-specific classes must be used for the ConnectionFactory and Destination objects. You can download JNDI jar files from http://java.sun.com/products/jndi

Testing the JMS class path

You can use the example program `examples.jms.MQeJMSIVT` to test your JMS installation. Before you run this program, you need an MQe queue manager that has a `SYSTEM.DEFAULT.LOCAL.QUEUE`. In addition to the JMS jar files mentioned above, you also need the following or equivalent jar files on your class path to run `examples.jms.MQeJMSIVT`:

- MQeBase.jar
- MQeExamples.jar

You can run the example from the command line by typing:

```
  java examples.jms.MQeJMSIVT -i
    <ini file name>
```

where <ini file name> is the name of the initialization (ini) file for the MQe queue manager. You can optionally add a ″-t″ flag to turn tracing on:

```
java examples.jms.MQeJMSIVT -t -i
    <ini file name>
```

The example program checks that the required jar files are on the class path by checking for classes that they contain. It creates a *QueueConnectionFactory* and configures it using the ini file name that you passed in on the command line. It starts a connection, which:

1. Starts the MQe queue manager
2. Creates a JMS Queue representing the queue `SYSTEM.DEFAULT.LOCAL.QUEUE` on the queue manager
3. Sends a message to the JMS Queue
4. Reads the message back and compares it to the message it sent

The `SYSTEM.DEFAULT.LOCAL.QUEUE` should not contain any messages before running the program, otherwise the message read back will not be the one that the program sent. The output from the program should look like this:

```
using ini file '<.ini file name>'
    to configure the connection
checking classpath
found JMS interface classes
found MQe JMS classes
found MQe base classes
Creating and configuring QueueConnectionFactory
Creating connection
From the connection data, JMS
```

```
  provider is IBM MQe Version 2.0.0.0
Creating session
Creating queue
Creating sender
Creating receiver
Creating message
Sending message
Receiving message


HEADER FIELDS
----------------------------------------
 JMSType:          jms_text
 JMSDeliveryMode:  2
 JMSExpiration:    0
 JMSPriority:      4
 JMSMessageID:     ID:00000009524cf094000000f052fc06ca
 JMSTimestamp:     1032184399562
 JMSCorrelationID: null
 JMSDestination:   null:SYSTEM.DEFAULT.LOCAL.QUEUE
 JMSReplyTo:       null
 JMSRedelivered:   false

 PROPERTY FIELDS (read only)
-----------------------------------------
 JMSXRcvTimestamp : 1032184400133

 MESSAGE BODY (read only)
-----------------------------------------
A simple text message from the MQeJMSIVT program

Retrieved message is a TextMessage; now checking
for equality with the sent message
Messages are equal. Great!
Closing connection
connection closed
IVT finished
```

Running other MQe JMS example programs

MQe provides two other example programs for the JMS classes. The program `examples.jms.PTPSample01` is similar to the IVT examples described above, but there is a command line argument to tell it not to use the *Java Naming and Directory Interface* (JNDI) and it does not have the same checks on the class path. The program requires the same JMS and MQe jar files on the class path as `examples.jms.MQeJMSIVT`, that is jms.jar, MQeJMS.jar, MQeBase.jar, and MQeExamples.jar. It also requires the jndi.jar file, even if it does not use JNDI, because the program imports javax.naming. The section on Using JNDI provides more information on the jndi.jar file. You can run the example from the command line by typing:

```
 java examples.jms.PTPSample01 -nojndi -i <ini file name>
```

where <ini file name > is the name of the initialization (ini) file for the MQe queue manager. By default, the program will use the `SYSTEM.DEFAULT.LOCAL.QUEUE` on this queue manager. You can specify a different queue by using the -q flag:

```
java examples.jms.PTPSample01 -i <ini file name> -q <queue name>
```

You can also turn tracing on by adding the -t flag:

```
java examples.jms.PTPSample01 -t -i <ini file name> -q <queue name>
```

The `examples.jms.PTPSample02` program uses message listeners and filters. This program creates a *QueueReceiver* with a "blue" filter and creates a message listener for it. It creates a second QueueReceiver with a "red" filter and message listener. It then sends four messages to a queue, two with the filter property colour set to blue and two with the filter property colour set to red, and checks that the message listeners receive the correct messages. The program has the same command line parameters as `examples.jms.PTPSample01`.

Writing JMS programs

Introduces the JMS model and provides information on writing MQe JMS applications

This section provides information on writing MQe JMS applications. It provides a brief introduction to the JMS model and information on programming some common tasks that application programs may need to perform.

The JMS model

JMS defines a generic view of a message service. It is important to understand this view, and how it maps onto the underlying MQe system. The generic JMS model is based around the following interfaces that are defined in Sun's javax.jms package:

Connection
This provides a connection to the underlying messaging service and is used to create *Sessions*.

Session
This provides a context for producing and consuming messages, including the methods used to create *MessageProducers* and *MessageConsumers*.

MessageProducer
This is used to send messages.

MessageConsumer
This is used to receive messages.

Destination
This represents a message destination.

Note: A connection is thread safe, but sessions, message producers, and message consumers are not. While the JMS specification allows a Session to be used by more than one thread, it is up to the user to ensure that Session resources are not concurrently used by multiple threads. The recommended strategy is to use one Session per application thread.

Therefore, in MQe terms:

Connection
This provides a connection to an MQe queue manager. All the Connections in a JVM must connect to the same queue manager, because MQe supports a single queue manager per JVM. The first connection created by an application will try and connect to an already running queue manager, and if that fails will attempt to start a queue manager itself. Subsequent connections will connect to the same queue manager as the first connection.

Session
This does not have an equivalent in MQe

Message producer and message consumer
These do not have direct equivalents in MQe. The MessageProducer invokes the `putMessage()` method on the queue manager. The MessageConsumer invokes the `getMessage()` method on the queue manager.

Destination
This represents an MQe queue.

MQe JMS can put messages to a local queue or an asynchronous remote queue and it can receive messages from a local queue. It cannot put messages to or receive messages from a synchronous remote queue.

The generic JMS interfaces are subclassed into more specific versions for Point-to-point and Publish or Subscribe behavior. MQe implements the Point-to-point subclasses of JMS. The Point-to-point subclasses are:

QueueConnection
Extends Connection

QueueSession
Extends Session

QueueSender
Extends MessageProducer

QueueReceiver
Extends MessageConsumer

Queue
Extends destination

It is recommended that you write application programs that use only references to the interfaces in javax.jms. All vendor-specific information is encapsulated in implementations of:

- QueueConnectionFactory
- Queue

These are known as "administered objects", that is, objects that can be administered and stored in a JNDI namespace. A JMS application can retrieve these objects from the namespace and use them without needing to know which vendor provided the implementation. However, on small devices looking up objects in a JNDI namespace may be impractical or represent an unnecessary overhead. We, therefore, provide two versions of the `QueueConnectionFactory` and Queue classes.

The parent classes, `MQeQueueConnectionFactory.class`, `MQeJMSQueue.class`, provide the base JMS functionality but cannot be stored in JNDI, while subclasses, MQeJNDIQueueConnectionFactory.class, and the MQeJMSJNDIQueue.class, add the necessary functionality for them to be stored and retrieved from JNDI.

Building a connection:

You normally build connections indirectly using a connection factory. A JNDI namespace can store a configured factory, therefore insulating the JMS application from provider-specific information. See the section Using JNDI, below, for details on how to store and retrieve objects using JNDI.

If a JNDI namespace is not available, you can create factory objects at runtime. However, this reduces the portability of the JMS application because it requires references to MQe specific classes. The following code creates a QueueConnectionFactory. The factory uses an MQe queue manager that is configured with an initialisation (ini) file:

```
QueueConnectionFactory factory;
factory = new com.ibm.mqe.jms.MQeJNDIQueueConnectionFactory();
((com.ibm.mqe.jms.MQeJNDIQueueConnectionFactory)factory).
setIniFileName(<initialisation file>)
```

Using the factory to create a connection:

Use the createQueueConnection() to create a QueueConnection:

```
QueueConnection connection;
connection = factory.createQueueConnection();
```

Starting the connection:

Under the JMS specification, connections are not active upon creation. Until the connection starts, MessageConsumers that are associated with the connection cannot receive any messages. Use the following command to start the connection:

```
connection.start();
```

Obtaining a session:

Once a connection has been created, you can use the `createQueueSession()` method on the QueueConnection to obtain a session. The method takes two parameters:

1. A boolean that determines whether the session is ″transacted″ or ″non-transacted″.
2. A parameter that determines the ″acknowledge″ mode. This is used when the session is ″non-transacted″.

The simplest case is that where acknowledgements are used and are handled by JMS itself with `AUTO_ACKNOWLEDGE`, as shown in the following code fragment:

```
QueueSession session;
boolean transacted = false;
session = connection.createQueueSession(transacted, Session.AUTO_ACKNOWLEDGE);
```

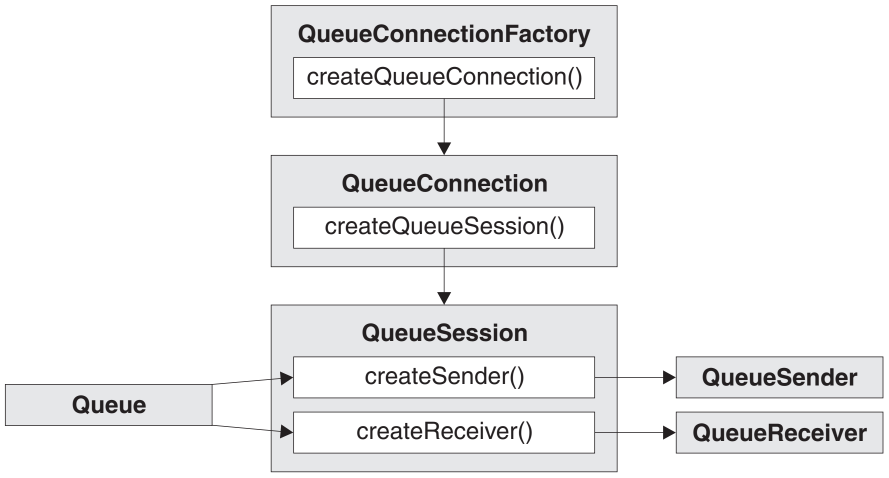


Figure 73. Obtaining a session once a connection is created

Sending a message:

Messages are sent using a MessageProducer. For point-to-point this is a QueueSender that is created using the `createSender()` method on QueueSession. A QueueSender is normally created for a specific Queue, so that all messages sent using that sender are sent to the same destination. Queue objects can be either created at runtime, or built and stored in a JNDI namespace. Refer to “Using Java Naming and Directory Interface (JNDI)” on page 149, for details on how to store and retrieve objects using JNDI.

JMS provides a mechanism to create a Queue at runtime that minimizes the implementation-specific code in the application. This mechanism uses the QueueSession.createQueue() method, which takes a string parameter describing the destination. The string itself is still in an implementation-specific format, but this is a more flexible approach than directly referencing the implementation classes.

For MQe JMS the string is the name of the MQe queue. This can optionally contain the queue manager name. If the queue manager name is included, the queue name is separated from it by a plus sign '+', for example:

```
ioQueue = session.createQueue("myQM+myQueue");
```

This will create a JMS Queue representing the MQe queue "myQueue" on queue manager "myQM". If no queue manager name is specified the local queue manager is used, i.e. the one that JMS is connected to. For example:

```
String queueName = "SYSTEM.DEFAULT.LOCAL.QUEUE";

...

ioQueue = session.createQueue(queueName);
```

This will create a JMS Queue representing the MQe queue `SYSTEM.DEFAULT.LOCAL.QUEUE` on the queue manager that the JMS Connection is using.

Message types:

JMS provides several message types, each of which embodies some knowledge of its content. To avoid referencing the implementation-specific class names for the message types, methods are provided on the Session object for message creation. In the sample program, a text message is created in the following manner:

```
System.out.println("Creating a TextMessage");
TextMessage outMessage = session.createTextMessage();
System.out.println("Adding Text");
outMessage.setText(outString);
```

The message types that can be used are:

- BytesMessage
- ObjectMessage
- TextMessage

Receiving a message:

Messages are received by using a QueueReceiver. This is created from a Session by using the `createReceiver()` method. This method takes a Queue parameter that defines where the messages are received from. See "Sending a message" above for details of how to create a Queue object. The sample program creates a receiver and reads back the test message with the following code:

```
QueueReceiver queueReceiver = session.createReceiver(ioQueue);
Message inMessage = queueReceiver.receive(1000);
```

The parameter in the receive call is a timeout in milliseconds. This parameter defines how long the method should wait if there is no message available immediately. You can omit this parameter, in which case the call blocks indefinitely. If you do not want any delay, use the `receiveNoWait()` method. The receive methods return a message of the appropriate type. For example, if a TextMessage is put on a queue, when the message is received the object that is returned is an instance of TextMessage . To extract the content from the body of the message, it is necessary to cast from the generic Message class, which is the declared return type of the receive methods, to the more specific subclass, such as TextMessage . If the received message type is not known, you can use the "instanceof" operator to determine which type it is. It is good practice always to test the message class before casting, so that unexpected errors can be handled gracefully. The following code illustrates the use of "instanceof", and extraction of the content from a TextMessage:

```
if (inMessage instanceof TextMessage){
    String replyString = ((TextMessage)inMessage).getText();
     ...
} else {
    //Print error message if Message was not a TextMessage.
    System.out.println("Reply message was not a TextMessage");
}
```

Handling errors:

Any runtime errors in a JMS application are reported by exceptions. The majority of methods in JMS throw JMSExceptions to indicate errors. It is good programming practice to catch these exceptions and handle them appropriately. Unlike normal Java Exceptions, a JMSException may contain a further exception embedded in it. For JMS, this can be a valuable way to pass important detail from the underlying transport. When a JMSException is thrown as a result of MQe raising an exception, the exception is usually included as the embedded exception in the JMSException. The standard implementation of JMSException does not include the embedded exception in the output of its `toString()` method. Therefore, it is necessary to check explicitly for an embedded exception and print it out, as shown in the following fragment:

```
try {
    ...code which may throw a JMSException
} catch (JMSException je) {
    System.err.println("caught "+je);
    Exception e = je.getLinkedException();
    if (e != null) {
        System.err.println("linked exception:"+e);
    }
}
```

Exception listener:

For asynchronous message delivery, the application code cannot catch exceptions raised by failures to receive messages. This is because the application code does not make explicit calls to `receive()` methods. To cope with this situation, it is possible to register an ExceptionListener, which is an instance of a class that implements the `onException()` method. When a serious error occurs, this method is called with the `JMSException` passed as its only parameter. Further details are in Sun's JMS documentation.

JMS messages:

JMS messages are composed of the following parts:

Header
All messages support the same set of header fields. Header fields contain values that are used by both clients and providers to identify and route messages.

Properties
Each message contains a built-in facility to support application-defined property values. Properties provide an efficient mechanism to filter application-defined messages.

Body JMS defines several types of message body which cover the majority of messaging styles currently in use. JMS defines five types of message body:

Text A message containing a `java.lang.String`

Object
A message that contains a Serializable java object

Bytes A stream of uninterpreted bytes for encoding a body to match an existing message format

Stream
A stream of Java primitive values filled and read sequentially, not supported in this version of MQe JMS

Map A set of name-value pairs, where names are Strings and values are Java primitive types.

The entries can be accessed sequentially or randomly by name. The order of the entries is undefined. Map is not supported in this version of MQe JMS.

The `JMSCorrelationID` header field is used to link one message with another. It typically links a reply message with its requesting message.

Message selectors:

A message contains a built-in facility to support application-defined property values. In effect, this provides a mechanism to add application-specific header fields to a message. Properties allow an application, via message selectors, to have a JMS provider select or filter messages on its behalf, using application-specific criteria. Application-defined properties must obey the following rules:

- Property names must obey the rules for a message selector identifier.
- Property values can be boolean, byte, short, int, long, float, double, and String.
- The JMSX and JMS_ name prefixes are reserved.

Property values are set before sending a message. When a client receives a message, the message properties are read-only. If a client attempts to set properties at this point, a MessageNotWriteableException is thrown. If `clearProperties()` is called, the properties can then be both read from, and written to.

A property value may duplicate a value in a message's body, or it may not. JMS does not define a policy for what should or should not be made into a property. However, for best performance, applications should only use message properties when they need to customize a message's header. The primary reason for doing this is to support customized message selection. A JMS message selector allows a client to specify the messages that it is interested in by using the message header. Only messages whose headers match the selector are delivered. Message selectors cannot reference message body values. A message selector matches a message when the selector evaluates to true when the message's header field and property values are substituted for their corresponding identifiers in the selector.

A message selector is a String, which can contain:

Literals

- A string literal is enclosed in single quotes. A doubled single quote represents a single quote. Examples are 'literal' and 'literal''s'. Like Java string literals, these use the Unicode character encoding.
- An exact numeric literal is a numeric value without a decimal point, such as 57, -957, +62. Numbers in the range of Java long are supported.
- An approximate numeric literal is a numeric value in scientific notation, such as 7E3 or -57.9E2, or a numeric value with a decimal, such as 7., -95.7, or +6.2. Numbers in the range of Java double are supported. Note that rounding errors may affect the operation of message selectors including approximate numeric literals.
- The boolean literals `TRUE` and `FALSE`.

Identifiers

- An identifier is an unlimited length sequence of Java letters and Java digits, the first of which must be a Java letter. A letter is any character for which the method `Character.isJavaLetter` returns `true`. This includes "_" and "$". A letter or digit is any character for which the method `Character.isJavaLetterOrDigit` returns `true`.
- Identifiers cannot be the names `NULL`, `TRUE`, or `FALSE`.
- Identifiers cannot be `NOT`, `AND`, `OR`, `BETWEEN`, `LIKE`, `IN`, and `IS`.
- Identifiers are either header field references or property references.
- Identifiers are case-sensitive.
- Message header field references are restricted to:

  - JMSDeliveryMode
  - JMSPriority
  - JMSMessageID
  - JMSTimestamp
  - JMSCorrelationID
  - JMSType

  JMSMessageID, JMSTimestamp, JMSCorrelationID, and JMSType values may be null, and if so, are treated as a NULL value.
- Any name beginning with ″JMSX″ is a JMS-defined property name
- Any name beginning with ″JMS_″ is a provider-specific property name
- Any name that does not begin with ″JMS″ is an application-specific property name
- If there is a reference to a property that does not exist in a message, its value is NULL. If it does exist, its value is the corresponding property value.

White space
This is the same as is defined for Java, space, horizontal tab, form feed, and line terminator.

Logical operators
Currently supports `AND` only.

Comparison operators
- Only equals (′=′) is currently supported.
- Only values of the same type can be compared.
- If there is an attempt to compare different types, the selector is always false.
- Two strings are equal if they contain the same sequence of characters.
- The `IS NULL` comparison operator tests for a null header field value, or a missing property value. The `IS NOT NULL` comparison operator is not supported.

Note that Arithmetic operators are not currently supported.

The following message selector selects messages with a message type of car and a colour of blue:
```
"JMSType ='car 'AND colour ='blue'"
```

When selecting Header fields MQe will interpret exact numeric literals so that they match the type of the field in question, that is a selector testing the JMSPriority or JMSDeliveryMode Header fields will interpret an exact numeric literal as an int, whereas a selector testing JMSExpiration or JMSTimestamp will interpret an exact numeric literal as a long. However, when selecting message properties MQe will always interpret an exact numeric literal as a long and an approximate numeric literal as a double. Application specific properties intended to be used for message selection should therefore be set using the `setLongProperty` and `setDoubleProperty` methods respectively.

Restrictions in this version of MQe

This version of MQe JMS implements the Point-to-Point subset of JMS with a few restrictions. It does not implement any of the optional classes:
- The application server classes ConnectionConsumer, ServerSession, and ServerSessionPool
- The XA classes:
  - XAConnection
  - XAConnectionFactory
  - XAQueueConnection
  - XAQueueConnectionFactory

– XAQueueSession
– XASession
– XATopicConnection
– XATopicConnectionFactory
– XATopicSession

It does not implement the TemporaryQueue class, which means that the QueueRequestor class will not work or the MapMessage and StreamMessage classes.

In the `QueueConnectionFactory`, the `createQueueConnection()` method that takes a username and password as parameters is not implemented, MQe does not have the concept of a user. The method with no parameters is implemented.

When a message is read from a queue but not acknowledged, the message is returned to the queue for redelivery. In this case the JMSRedelivered header field should be set in the message. MQe JMS does not set this header field.

MQe JMS can put messages to a local queue or an asynchronous remote queue and it can receive messages from a local queue. It cannot put to or receive messages from a synchronous remote queue.

Using Java Naming and Directory Interface (JNDI)

One of the advantages of using JMS is the ability to write applications which are independent of the JMS implementations, allowing you to plug in a JMS implementation which is appropriate for your environment. However, certain JMS objects must be configured in a way which is specific to the JMS implementation you have chosen. These objects are the connection factories and destinations, queues, and they are often referred to as ″administered objects″. In order to keep the application programs independent of the JMS implementation, these objects must be configured outside of the application programs. They would typically be configured and stored in a JNDI namespace. The application would lookup the objects in the namespace and would be able to use them straight away, because they have already been configured.

There may be situations, such as on a small device, where it would not be desirable to use JNDI. In these cases the objects could be configured directly in the application. The cost of not using JNDI would be a small degree of implementation-dependence in the application.

Storing and retrieving objects with JNDI

Before using JNDI to either store or retrieve objects, an ″initial context″ must be set up, as shown in this fragment taken from the MQeJMSIVT_JNDI example program:

```
import javax.jms.*;
import javax.naming.*;
import javax.naming.directory.*;

...
java.util.Hashtable environment =new java.util.Hashtable();
environment.put(Context.INITIAL_CONTEXT_FACTORY, icf);
environment.put(Context.PROVIDER_URL, url);
Context ctx = new InitialContext(environment );
```

where:

icf defines a factory class for the JNDI context. This depends upon the JNDI provider that you are using. The documentation supplied by the JNDI provider should tell you what value to use for this. See also the examples below.

url defines the location of the namespace. This will depend on the type of namespace you are using.

If you are using the file system, this will be a file url that identifies a directory in your file system. If you are using LDAP this will be a ldap url that identifies a LDAP server and location in the directory tree of that server. The documentation supplied by the JNDI provider should describe the correct format for the url.

For more details about JNDI usage, see Sun's JNDI documentation.

Note: Some combinations of the JNDI packages and LDAP service providers can result in an LDAP error 84. To resolve the problem, insert the following line before the call to InitialContext.

```
environment.put(Context.REFERRAL,"throw");
```

Once an initial context is obtained, objects can be stored in and retrieved from the namespace. To store an object, use the `bind()` method:

```
ctx.bind(entryName, object);
```

where 'entryName' is the name under which you want the object stored, and 'object' is the object to be stored, for example to store a factory under the name "ivtQCF":

```
ctx.bind("ivtQCF", factory);
```

To store an object in a JNDI namespace, the object must satisfy either the javax.naming.Referenceable interface or the java.io.Serializable interface, depending on the JNDI provider you use. The `MQeJNDIQueueConnectionFactory` and `MQeJMSJNDIQueue`classes implement both of these interfaces. To retrieve an object from the namespace, use the`lookup()` method:

```
object = ctx.lookup(entryName);
```

where `entryName` is the name under which you want the object stored , for example, to retrieve a `QueueConnectionFactory` stored under the name "ivtQCF":

```
QueueConnectionFactory factory;
factory = (QueueConnectionFactory)ctx.lookup("ivtQCF");
```

Using the sample programs with JNDI

The example program `examples.jms.MQeJMSIVT_JNDI` can be used to test your installation using JNDI. This is very similar to the `examples.jms.MQeJMSIVT` program, except that it uses JNDI to retrieve the connection factory and the queue that it uses. Before you can run this program you must store these two administered objects in a JNDI namespace:

Table 4. Administered objects for a JNDI namespace

Entry name	Java class	Description
ivtQCF	`MQeJNDIQueueConnectionFactory`	A `QueueConnectionFactory` configured to use an MQe queue manager
ivtQ	`MQeJMSJNDIQueue`	A Queue configured to represent an MQe queue which is local to the queue manager used by the ivtQCF entry

The program examples.jms.CreateJNDIEntry or the MQeJMSAdmin tool , explained in the following section, can be used to create these entries. Larger installations may have a *Lightweight Directory Access Protocol (LDAP)* directory available, but for smaller installations a file system namespace may be more appropriate. When you have decided on a namespace you must obtain the corresponding JNDI class files to support the namespace and add these to your classpath. These will vary depending on your choice of namespace and the version of Java you are using.

You must always have the javax.naming.* classes on your classpath. If you are using Java 1 (for example a 1.1.8 JRE) you must obtain a copy of the jndi.jar file and add it to your classpath. If you are using Java 2 (a 1.2 or later JRE) the JRE may contain these classes itself.

If you want to use an LDAP directory, you must obtain JNDI classes that support LDAP, for example Sun's ldap.jar or IBM's ibmjndi.jar, and add these to your classpath. Some Java 2 JREs may already contain Sun's classes for LDAP. See also the section below about LDAP support for Java classes.

If you want to use a file system directory, you must obtain JNDI classes that support the file system, for example Sun's fscontext.jar (which requires providerutil.jar as well) and add these to your classpath. The CreateJNDIEntry example program requires the MQeJMS.jar file on your classpath, in addition to the JNDI jar files. It takes the following command line arguments:

```
java examples.jms.CreateJNDIEntry -url<providerURL>
    [-icf<initialContextFactory>][-ldap]
     [-qcf<entry name><MQe queue manager ini file>]
    [-q<entry name><MQe queue name>]
```

An alternative argument to use is:

```
java  examples.jms.CreateJNDIEntry -h
```

In the previous two examples:

-url<providerURL>
The URL of the JNDI initial context (obligatory parameter)

-icf<initialContextFactory>
The initialContextFactory for JNDI that defaults to the file system:

```
com.sun.jndi.fscontext.RefFSContextFactory
```

-ldap This should be specified if you are using an LDAP directory

-qcf<entry name><MQe queue manager ini file>
The name of a JNDI entry to be created for a JMS `QueueConnectionFactory` and the name of an initialisation (ini) file for an MQe queue manager to be used to configure it

-h Displays a help message

The url, -url, must be specified and either a QueueConnectionFactory (-qcf) or a Queue (-q), or both, must be specified. The context factory, -icf, is optional and defaults to a file system directory. The LDAP flag, -ldap, should be specified if an LDAP directory is being used, this prefixes the entry name with ″cn=″, which is required by LDAP.

For example, if a queue manager with the initialization file *d:\MQe\exampleQM\exampleQM.ini* exists, and you are using a JNDI directory based in the file system at *d:\MQe\data\jndi\*, type (all on one line):

```
  java  examples.jms.CreateJNDIEntry -url file://d:/MQe/data/jndi -qcf  ivtQCF
  d:\MQe\exampleQM\exampleQM.ini
```

Note that forward slashes are used in the url, even if the file system itself uses back slashes. The url directory must already exist. To add an entry for the queue you would type (all on one line):

```
  java  examples.jms.CreateJNDIEntry -url file://
  d:/MQe/data/jndi -q ivtQ SYSTEM.DEFAULT.LOCAL.QUEUE
```

You could use another local queue instead of the SYSTEM.DEFAULT.LOCAL.QUEUE.

You could also specify the queue name as `exampleQM+SYSTEM.DEFAULT.LOCAL.QUEUE`, where `exampleQM` is the name of the queue manager. If the name of the queue manager is not specified, the local queue manager is used.

Both entries could be added at the same time by typing:

```
java  examples.jms.CreateJNDIEntry
        -url file://d:/MQe/data/jndi -qcf ivtQCF
d:\MQe\exampleQM\exampleQM.ini -q ivtQ
                SYSTEM.DEFAULT.LOCAL.QUEUE
```

Again, you should type all of this command on one line. A maximum of one connection factory and one queue can be added at a time.

When the JNDI entries have been created, you can run the `example .jms.MQeJMSIVT_JNDI` program. This requires the same jar files on the classpath as the MQeJMSIVT program, that is:

- `jms.jar`, Sun's interface definition for the JMS classes
- `MQeJMS.jar`, the MQe implementation of JMS
- `MQeBase.jar`
- `MQeExamples.jar`

It also requires the JNDI jar files, as used for the `CreateJNDIEntry` example program. The example can be run from the command line by typing:

```
java  examples.jms.MQeJMSIVT_JNDI
     -url<providerURL>
```

where <providerURL> is the specified URL of the JNDI initial context. By default the program uses the file system context for JNDI:

```
com.sun.jndi.fscontext.RefFSContextFactory
```

If necessary you can specify an alternative context:

```
java  examples.jms.MQeJMSIVT_JNDI -url<providerURL>
    -icf<initialContextFactory>
```

You can optionally add a `-t` flag to turn tracing on:

```
java  examples.jms.MQeJMSIVT_JNDI -url<providerURL>
     -icf<initialContextFactory> -t
```

To use the entries in the file system directory created in the CreateJNDIEntry example above, type:

```
java examples.jms.MQeJMSIVT_JNDI -url file://d:/MQe/data/jndi
```

The example program checks that the required jar files are on the classpath by checking for classes that they contain. It looks up the QueueConnectionFactory and the Queue in the JNDI directory. It starts a connection, which starts the MQe queue manager, sends a message to the Queue, reads the message back and compares it to the message it sent. The queue should not contain any messages before running the program, otherwise the message read back will not be the one that the program sent. The first lines of output from the program should look like this:

```
using context factory
      'com.sun.jndi.fscontext.RefFSContextFactory' for the directory
using directory url 'file://d:/MQe/data/jndi'
checking classpath
found JMS interface classes
found MQe JMS classes
found MQe base classes
found jndi.jar classes
found com.sun.jndi.fscontext.RefFSContextFactory classes
Looking up connection factory in jndi
Looking up queue in jndi
Creating connection
```

The rest of the output should be similar to that from the example without JNDI. You can also run the two other example programs, `examples.jms.PTPSample01` and `example .jms.PTPSample02`, using JNDI. These programs requires the same JMS and MQe jar files on the classpath as the `MQeJMSIVT_JNDI` program, that is:

- `jms.jar`
- `MQeJMS.jar`
- `MQeBase.jar`
- `MQeExamples.jar`

They also require the jndi.jar file and the jar files for the JNDI provider you are using, for example, file system or LDAP. The examples can be run from the command line by typing:

```
 java  examples.jms.PTPSsample01 -url<providerURL>
```

As in the previous example, `providerURL` is the URL of the JNDI initial context. By default, the program uses the file system context for JNDI, that is `com.sun.jndi.fscontext.RefFSContextFactory`. If necessary you can specify an alternative context:

```
java examples.jms.PTPSsample01 -url<providerURL>
              -icf<initialContextFactory>
```

You can optionally add a ″-t″ flag to turn tracing on: java examples.jms. PTPSsample01 -url <providerURL><-icf initialContextFactory> -t . To use the entries in the file system directory created in the CreateJNDIEntry example above, you would type:

```
java examples.jms.PTPSample01 -url file://d:/MQe/data/jndi
```

The program `examples.jms.PTPSample02` uses message listeners and filters. It creates a `QueueReceiver` with a filter ″colour=′blue′″ and creates a message listener for it. It creates a second `QueueReceiver` with a filter ″colour=′red′″ and also creates a message listener. It sends four messages to a queue, two with the property ″colour″ set to ″red″ and two with the property ″colour″ set to ″blue″, and checks that the message listeners receive the correct messages. The program has the same command line parameters as the `PTPSample01` program and can be run in the same way. Simply substitute `PTPSample02` for `PTPSample01`.

Mapping JMS messages to MQe messages

This section describes how the JMS message structure is mapped to an MQe message. It is of interest to programmers who wish to transmit messages between JMS and traditional MQe applications.

As described earlier, the JMS specification defines a structured message format consisting of a header, three types of property and five types of message body, while MQe defines a single free-format message object, `MQeMsgObject`. MQe defines some constant field names that messaging applications require, for example UniqueID, MessageID, and Priority, while applications can put data into an MQe message as `<name, value>` pairs.

To send JMS messages using MQe, we define a constant format for storing the information contained in a JMS message within an `MQeMsgObject`. This adds three top-level fields and four `MQeFields` objects to an `MQeMsgObject`, as shown in the following example.

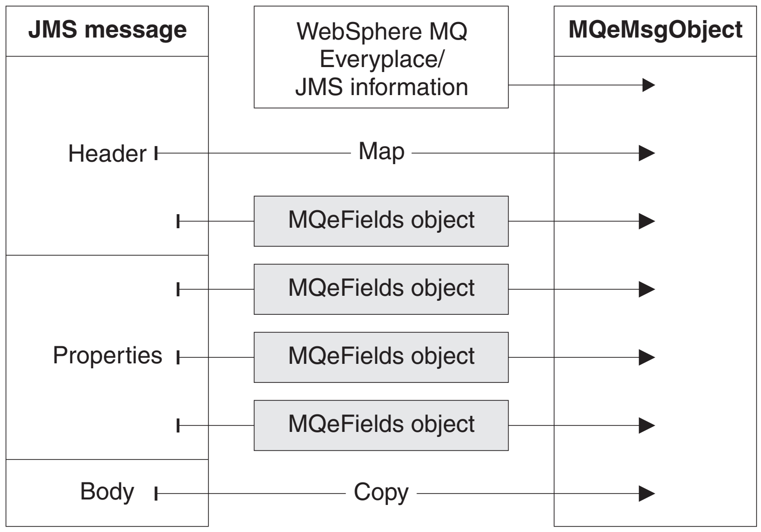


Figure 74. Mapping a JMS message to an MQeMQeMsgObject

The following sections describe the contents of these fields:

Naming MQeMsgObject fields

An `MQeMsgObject` stores data as a `<name, value>` pair. The field names used to map JMS message data to the MQeMsgObject are defined in com.ibm.mqe.MQe and com.ibm.mqe.jms.MQeJMSMsgFieldNames:

MQeJMS field names

```
MQe.MQe_JMS_VERSION
MQeJMSMsgFieldNames.MQe_JMS_CLASS
```

JMS message field names

```
MQeJMSMsgFieldNames.MQe_JMS_HEADER
MQeJMSMsgFieldNames.MQe_JMS_PROPERTIES
MQeJMSMsgFieldNames.MQe_JMS_PS_PROPERTIES
MQeJMSMsgFieldNames.MQe_JMSX_PROPERTIES
MQeJMSMsgFieldNames.MQe_JMS_BODY
```

JMS header field names

```
MQeJMSMsgFieldNames.MQe_JMS_DESTINATION
MQeJMSMsgFieldNames.MQe_JMS_DELIVERYMODE
MQeJMSMsgFieldNames.MQe_JMS_MESSAGEID
MQeJMSMsgFieldNames.MQe_JMS_TIMESTAMP
MQeJMSMsgFieldNames.MQe_JMS_CORRELATIONID
MQeJMSMsgFieldNames.MQe_JMS_REPLYTO
MQeJMSMsgFieldNames.MQe_JMS_REDELIVERED
MQeJMSMsgFieldNames.MQe_JMS_TYPE
MQeJMSMsgFieldNames.MQe_JMS_EXPIRATION
MQeJMSMsgFieldNames.MQe_JMS_PRIORITY
```

MQe JMS information

Two <name, value> pairs holding information required for MQe to recreate the JMS message are added directly to the `MQeMsgObject`:

`MQe.MQe_JMS_VERSION`

This contains a *short* describing the version number of the MQe JMS implementation used to store the message. The current version number is 1. The presence or absence of a field named `MQe.MQe_JMS_VERSION` is used to determine if an `MQeMsgObject` contains an MQe JMS message.

MQeJMSMsgFieldNames.MQe_JMS_CLASS
This contains a *String* describing the type of JMS message body stored in the MQeMsgObject. It defines the strings in the following table:

Table 5. Strings in MQeJMSMsgFieldNames.MQe_JMS_CLASS

JMS message type	**MQe.MQe_JMS_CLASS**
Bytes message	jms_bytes
Map message	jms_map
Null message	jms_null
Object message	jms_object
Stream message	jms_stream
Text message	jms_text

JMS header files

JMS Header fields are stored within an MQeMsgObject using the following rules:
1. If a JMS header field is identical to a defined MQeMsgObject field then the header value is mapped directly to the appropriate field in the MQeMsgObject.
2. If a JMS header field does not map directly to a defined field but can be represented using existing fields defined by MQe then the JMS header value is converted as appropriate and then set in the MQeMsgObject.
3. If MQe has not defined an equivalent field by then, the header field is stored within an MQeFields object, which is then embedded in the MQeMsgObject. This ensures that the JMS header field in question can be restored when the JMS message is recreated.

The header fields that map directly to MQeMsgObject fields are:

Table 6. Header fields that map directly to MQeMsgObject fields

JMS header field	**MQeMsgObject defined field**
JMSTimestamp	MQe.Msg_Time
JMSCorrelationID	MQe.Msg_CorrelID
JMSExpiration	MQe.Msg_ExpireTime
JMSPriority	MQe.Msg_Priority

Two JMS header fields, JMSReplyTo and JMSMessageID, are converted prior to being stored in MQeMsgObject fields.

JMSReplyTo is split between MQe.Msg_ReplyToQMgr and MQe.Msg_ReplyToQ, while JMSMessageID is the String "ID:" followed by a 24-byte hashcode generated from a combination of MQe.Msg_OriginQMgr and MQe.Msg_Time.

The remaining four JMS header fields, JMSDeliveryMode, JMSRedelivered, and JMSType have no equivalents in MQe. These fields are stored within an MQeFields object in the following manner:
- As an int field named MQe.MQe_JMS_DELIVERYMODE
- As a boolean field named MQe.MQe_JMS_REDELIVERED
- As a String field named MQe.MQe_JMS_JMSTYPE

This MQeFields object is then stored within the MQeMsgObject as MQe.MQe_JMS_HEADER. Finally, JMSDestination is recreated when the message is received and, therefore does not need to be stored in the MQeMsgObject.

JMS properties

When storing JMS property fields in an MQeMsgObject, the <name, value> format used by the JMS properties corresponds very closely to the format of data in an MQeFields object:

Table 7. JMS property fields and the MQeFields object

Property type	Corresponding MQeFields object
Application-specific	MQe.MQe_JMS_PROPERTIES
Standard (JMSX_name)	MQe.MQe_JMSX_PROPERTIES
Provider-specific (JMS_provider_name)	MQe.MQe_JMS_PS_PROPERTIES

Three MQeFields objects, corresponding to the three types of JMS property, application-specific, standard, and provider-specific are used to store the <name, value> pairs stored as JMS message properties.

These three MQeFields objects are then embedded in the MQeMsgObject with the following names:

- MQe.MQe_JMS_PROPERTIES, application-specific
- MQe_MQe_JMSX_PROPERTIES, standard properties
- MQe.MQe_JMS_PS_PROPERTIES, provider-specific

Note that MQe does not currently set any provider specific properties. However, this field is used to enable MQe to handle JMS messages from other providers, for example MQ.

The following code fragment creates an MQe JMS text message by adding the required fields to an MQeMsgObject:

```
// create an MQeMsgObject
  MQeMsgObject msg = new MQeMsgObject();

  // set the JMS version number
  msg.putShort(MQe.MQe_JMS_VERSION, (short)1);
  // and set the type of JMS message this MQeMsgObject contains
  msg.putAscii(MQeJMSMsgFieldNames.MQe_JMS_CLASS, "jms_text");

  // set message priority and exipry time - these are mapped to
    JMSPriority and JMSExpiration
  msg.putByte(MQe.Msg_Priority, (byte)7);
  msg.putLong(MQe.Msg_ExpireTime, (long)0);

  // store JMS header fields with no MQe
    equivalents in an MQeFields object
  MQeFields headerFields = new MQeFields();
  headerFields.putBoolean(MQeJMSMsgFieldNames.MQe_JMS_REDELIVERED,
                  false);
  headerFields.putAscii(MQeJMSMsgFieldNames.MQe_JMS_TYPE,
                "testMsg");
  headerFields.putInt(MQeJMSMsgFieldNames.MQe_JMS_DELIVERYMODE,
  Message.DEFAULT_DELIVERY_MODE);
  msg.putFields(MQeJMSMsgFieldNames.MQe_JMS_HEADER,
            headerFields);

  // add an integer application-specific property
  MQeFields propField = new MQeFields();
  propField.putInt("anInt", 12345);
  msg.putFields(MQeJMSMsgFieldNames.MQe_JMS_PROPERTIES,
            propField);

  // the provider-specific and JMSX properties are blank
  msg.putFields(MQeJMSMsgFieldNames.MQe_JMSX_PROPERTIES,
          new MQeFields());
  msg.putFields(MQeJMSMsgFieldNames.MQe_JMS_PS_PROPERTIES,
          new MQeFields());

  // finally add a text message body
  String msgText =
      "A test message to MQe JMS";
  byte[] msgBody = msgText.getBytes("UTF8");
  msg.putArrayOfByte(MQeJMSMsgFieldNames.MQe_JMS_BODY,
                   msgBody);

  // send the message to an MQe Queue
  queueManager.putMessage(null,
                  "SYSTEM.DEFAULT.LOCAL.QUEUE",
                  msg, null, 0);
```

Now, you use JMS to receive the message and print it:

```
// first set up a QueueSession, then...
  Queue queue = session.createQueue
    ("SYSTEM.DEFAULT.LOCAL.QUEUE");
  QueueReceiver receiver = session.createReceiver(queue);

  // receive a message
  Message rcvMsg = receiver.receive(1000);

  // and print it out
  System.out.println(rcvMsg.toString());
```

This gives:

```
   HEADER FIELDS
  -----------------------------
   JMSType:          testMsg
   JMSDeliveryMode:  2
   JMSExpiration:    0
   JMSPriority:      7
   JMSMessageID:     ID:00000009524cf094000000f07c3d2266
   JMSTimestamp:     1032876532326
  JMSCorrelationID: null
  JMSDestination:   null:SYSTEM.DEFAULT.LOCAL.QUEUE
   JMSReplyTo:       null
   JMSRedelivered:   false

   PROPERTY FIELDS (read only)
  ------------------------------
   JMSXRcvTimestamp : 1032876532537
   anInt : 12345

   MESSAGE BODY (read only)
------------------------------------------------------------------
  A test message to MQe JMS
```

Note that JMS sets some of the JMS message fields, for example `JMSMessageID`, `JMSXRcvTimestamp` internally.

JMS message body:

Regardless of the JMS message type, MQe stores the JMS message body internally as an array of bytes. For the currently supported message types, this byte array is created as follows:

Table 8. JMS message body

JMS message type	Conversion
Bytes message	ByteArrayOutputStream.toByteArray();
Object message	<serialized object>.toByteArray();

Table 8. JMS message body (continued)

JMS message type	Conversion
Text message	String.getBytes("UTF-8");

When the JMS message body is stored in an `MQeMsgObject`, this *byte* array is added directly to the `MQeMsgObject` with the name `MQe.MQe_JMS_BODY`.

MQe JMS classes

MQe classes for Java Message Service consist of a number of Java classes and interfaces that are based on the Sun javax.jms package of interfaces and classes. They are contained in the com.ibm.mqe.jms package. The following classes are provided:

Table 9. MQe JMS classes

Class	Implements
MQeBytesMessage	BytesMessage
MQeConnection	Connection
MQeConnectionFactory	ConnectionFactory
MQeConnectionMetaData	ConnectionMetaData
MQeDestination	Destination
MQeJMSEnumeration	Java.util.Enumeration from QueueBrowser
MQeJMSJNDIQueue	Queue
MQeJMSQueue	Queue
MQeMessage	Message
MQeMessageConsumer	MessageConsumer
MQeMessageProducer	MessageProducer
MQeObjectMessage	ObjectMessage
MQeQueueBrowser	QueueBrowser
MQeQueueConnection	QueueConnection
MQeJNDIQueueConnectionFactory	QueueConnectionFactory
MQeQueueConnectionFactory	QueueConnectionFactory
MQeQueueReceiver	QueueReceiver
MQeQueueSender	QueueSender
MQeQueueSession	QueueSession
MQeSession	Session
MQeTextMessage	TextMessage

Note that MessageListener and ExceptionListener are implemented by applications.

Errors and error handling

Overview of errors and error handling in Java and C

This chapter describes what happens if an error occurs within the Java and C code bases.

Error handling in Java

Errors within the Java code base are handled using exceptions. The MQe Java Programming Reference documents all of the exception codes that the MQe Java code can return in the following classes:

- com.ibm.mqe.MQeExceptionCodes
- com.ibm.mqe.mqbridge.MQeBridge.ExceptionCodes

Error handling in C

The C code base indicates errors using *Return* and *Reason* codes. The C code does not have any exception handling mechanism, as in C++. MQe does not use the operating system error handling functions. An `MQeExceptBlock` handles errors and returns values from the functions. An application is free to install any operating system exception handlers that it requires.

The specific nature of an error condition is returned using two values, `MQERETURN` and `MQEREASON`. `MQERETURN` determines the general area in which the application failed, and distinguishes between warnings and errors. You can ignore warnings, but you must not ignore errors. With errors, your application needs to solve the problem in order to continue safely.

`MQERETURN` and `MQEREASON` are both returned in the `MQeExceptBlock`. The `MQERETURN` value is also the return value from the function.

Code structure

The MQe_nativeReturnCodes.h header file lists all of the return and reason codes. They are divided into function area and then by error or warning. For example, `MQERETURN_QUEUE_MANAGER_ERROR` and `MQERETURN_QUEUE_MANAGER_WARNING`. Warnings indicate that a situation can be ignored.

Exception block

The `MQeExceptBlock` structure is used to pass the return code and reason code, generated by a function call, back to the user. If a function call does not return `MQERETURN_OK`, use the ERC macro to get the reason code.

MQe ships two macros:

EC This macro resolves to the return code in the exception block structure.

ERC This macro resolves to the reason code in the exception block structure.

The convention within MQe is that a pointer to an exception block is passed first on a new function. A pointer to the object handle is passed second, followed by any additional parameters. On subsequent calls, the object handle is the first parameter passed, and the pointer to the exception block is second, followed by any additional parameters.

The structure of the exception block, as shown in the following example, is MQeExceptBlock_st.

```
struct MQeExceptBlock_st
  {
     MQERETURN    ec;
           /* return code*/
     MQEREASON    erc;
           /* reason code*/
     MQEVOID*      reserved;
           /* reserved for internal use only*/
  }
```

It is recommended that you allocate the Exception Block on the stack, rather than the heap. This simplifies possible memory allocations, although there are no restrictions on allocating space on the heap. The following code demonstrates how to do this:

```
MQERETURN rc
MQeExceptBlock exceptBlock;
/*.....initialisation*/
rc = mqeFunction_anyFunction(&exceptBlock,
/*parameters go here*/);
if (MQERETURN_OK ! = rc) {
printf("An error has occured, return code =
      %d, reason code =%d \n",
      exceptBlock.ec exceptBlock.erc);
}else {
}
```

All API calls need to take exception blocks. The C Bindings code base permits NULL to be passed to an API call. However, this feature is deprecated in the C code base and, therefore, not recommended.

You should use a different exception block for each thread in the application.

Note: If an error is not corrected, subsequent API calls can put the system in an unpredictable state.

Useful macros

A number of macros help to access the exception block:

SET_EXCEPT_BLOCK
Sets the return and reason codes to specific values, for exampe:

```
MQeExceptBlock exceptBlock;
SET_EXCEPT_BLOCK(&exceptBlock,
        MQERETURN_OK,
        MQEREASON_NA);
```

SET_EXCEPT_BLOCK_TO_DEFAULT
Sets return and reason codes to non-error values, for example:

```
MQeExceptBlock exceptBlock;
SET_EXCEPT_BLOCK_TO_DEFAULT(&exceptBlock);
```

EC Accesses the return code, for example:

```
MQeExceptBlock exceptBlk;
/*MQe API call */
MQERETURN returncode;
returnCode = EC(&exceptBlock);
```

ERC Accesses the reason code, for example:

```
MQeExceptBlock exceptBlk;
/*MQe API call*/
MQEREASON reasoncode;
MQEREASON reasonCode = ERC(&exceptBlock);
```

NEW_EXCEPT_BLOCK
Can create a temporary exception block. This is useful for temporary clean-up operations.

Index