WebSphere MQ Everyplace V2.0.2

Contenido

Diseño de una aplicación real

Mensajería

Visión general de la mensajería de MQe

El modelo de programación de MQe utiliza varias entidades, por ejemplo, mensajes, colas y gestores de colas que funcionan juntos como un kit de herramientas flexible. Cada entidad tiene un propósito específico y funciona en colaboración con otras entidades para proporcionar soluciones para las topologías de mensajes.

¿Qué son los mensajes de MQe?

Introducción al uso de los mensajes de MQe

Los mensajes son colecciones de datos que envía una aplicación y que van dirigidos a otra aplicación. Los mensajes de MQe contienen contenido definido por aplicaciones. Cuando se almacenan estos mensajes, se retienen en una cola y se pueden desplazar a través de una red MQe.

Los mensajes de MQe son un tipo especial de elementos MQeFields, como se describe en "MQeFields" en la página 6. Por lo tanto, se pueden utilizar los métodos aplicables a MQeFields con mensajes.

De este modo, los mensajes son objetos de campos con más campos especiales. Java ofrece una subclase de MQeFields, MQeMsgObject, que proporciona métodos para gestionar estos campos. La base de código en C no facilita una subclase como esta. En lugar de eso, proporciona varias funciones mqeFieldsHelper_operation. Los siguientes campos constituyen el *ID exclusivo* de un mensaje de MQe:

- En Java, la indicación de la hora se genera cuando el mensaje se crea por primera vez o, en C, cuando el mensaje se coloca por primera vez en una cola.
- El nombre del gestor de colas en el que el mensaje se coloca por primera vez.

El `ID exclusivo` identifica un mensaje dentro de una red de MQe, siempre que todos los gestores de colas de la red MQe tengan nombres exclusivos. Sin embargo, MQe no comprueba ni impone la exclusividad de los nombres de los gestores de colas.

En Java, el mensaje se crea cuando se crea una instancia de MQeMsgObject. En C, el mensaje se ″crea″, es decir, se añaden campos de ID exclusivo, cuando el mensaje se transmite a una cola.

El método mqeMsg_getMsgUIDFields() o la función mqeFieldsHelpers_getMsgUidFields() accede al `ID exclusivo` de un mensaje, por ejemplo:

Código Java

Código C

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

MQe añade información relacionada con propiedades a un mensaje (y posteriormente la elimina) para implementar operaciones de mensajería y de gestión de colas. Cuando se envía un mensaje entre gestores de colas, se puede añadir un reenvío de la información para indicar que los datos se están retransmitiendo.

Los mensajes habituales basados en aplicaciones tienen propiedades adicionales de acuerdo con su finalidad. Algunas de estas propiedades adicionales son genéricas y comunes a muchas aplicaciones, como es el caso del nombre del gestor de colas de respuestas.

Propiedades de mensajes

Tabla de propiedades de los mensajes de MQe

MQe da soporte a las siguientes propiedades de mensajes:

Tabla 1. Propiedades de mensajes

Nombre de la propiedad	Tipo Java	Tipo C	Descripción
Acción	int (entero)	MQEINT32	Lo utiliza la administración para indicar acciones como, por ejemplo, consulta, creación y supresión
ID de correlación	byte[]	MQEBYTE[]	Serie de bytes habitualmente utilizada para asociar una respuesta al mensaje original
Errores	MQeFields	MQeFieldsHndl	Lo utiliza la administración para devolver información de error
Caducidad	int (entero) o long (entero largo)	MQEINT32 o MQEINT64	Tiempo tras el cual el mensaje se puede suprimir (aunque no se haya entregado)
ID de bloqueo	long (entero largo)	MQEINT64	La clave necesaria para desbloquear un mensaje
ID de mensaje	byte[]	MQEBYTE[]	Un identificador exclusivo para un mensaje
Gestor de colas de origen	string (serie)	MQeStringHndl	Nombre del gestor de colas que ha enviado el mensaje
Parámetros	MQeFields	MQeFieldsHndl	Lo utiliza la administración para pasar detalles de administración
Prioridad	byte	MQEBYTE	Orden de prioridades relativas para la transmisión de mensajes
Motivo	string (serie)	MQeStringHndl	Lo utiliza la administración para devolver información de error
Cola de respuestas	string (serie)	MQeStringHndl	Nombre de la cola a la que se debe dirigir la respuesta de un mensaje
Gestor de colas de respuestas	string (serie)	MQeStringHndl	Nombre del gestor de colas al que se debe dirigir la respuesta de un mensaje
Reenviar	boolean (booleano)	MQEBOOL	Indica que el mensaje es el reenvío de un mensaje anterior
Código de retorno	byte	MQEBYTE	Lo utiliza la administración para devolver el estado de una operación de administración
Estilo	byte	MQEBYTE	Distingue mandatos de petición/respuesta, por ejemplo
Recortar mensaje	byte[]	MQEBYTE[]	Mensaje empaquetado para asegurar la protección de los datos

Nombres simbólicos:

Tabla de nombres simbólicos correspondientes a propiedades de mensajes de MQe

En la tabla siguiente se enumeran los nombres simbólicos correspondientes a las propiedades de los mensajes de MQe:

Tabla 2. Nombres simbólicos que corresponden a nombres de propiedades de mensajes

Nombre de la propiedad	**Constante Java**	**Constante C**
Acción	MQeAdminMsg.Admin_Action	MQE_ADMIN_ACTION
ID de correlación	MQe.Msg_CorrelID	MQE_MSG_CORRELID
Errores	MQeAdminMsg.Admin_Errors	MQE_ADMIN_ERRORS
Caducidad	MQe.Msg_ExpireTime	MQE_MSG_EXPIRETIME
ID de bloqueo	MQe.Msg_LockID	MQE_MSG_LOCKID
ID de mensaje	MQe.Msg_MsgID	MQE_MSG_MSGID
Gestor de colas de origen	MQe.Msg_OriginQMgr	MQE_MSG_ORIGIN_QMGR
Parámetros	MQeAdminMsg.Admin_Params	MQE_ADMIN_PARAMS
Prioridad	MQe.Priority	MQE_MSG_PRIORITY
Motivo	MQeAdminMsg.Admin_Reason	MQE_ADMIN_REASON
Cola de respuestas	MQe.Msg_ReplyToQ	MQE_MSG_REPLYTO_Q
Gestor de colas de respuestas	MQe.Msg_ReplyToQMgr	MQE_MSG_REPLYTO_QMGR
Reenviar	MQe.Msg_Resend	MQE_MSG_RESEND
Código de retorno	MQeAdminMsg.Admin_RC	MQE_ADMIN_RC
Estilo	MQe.Msg_Style	MQE_MSG_STYLE
Recortar mensaje	MQe.Msg_WrapMsg	MQE_MSG_WRAPMSG

Ejemplos:

Propiedades de los mensajes: ejemplos

En todos los casos se dispone de una constante definida que permite transportar el nombre de propiedad en un solo byte. Por ejemplo, la prioridad (si está presente) afecta al orden en el que se transmiten los mensajes, el ID de correlación activa la indexación de una cola para la recuperación rápida de información, el tiempo de caducidad activa la caducidad del mensaje y así sucesivamente. Asimismo, el mandato de vuelco de mensajes predeterminados minimiza el tamaño de la serie de bytes generada para obtener una transmisión y un almacenamiento de mensajes eficaz.

Los *ID de mensaje* e *ID de correlación* de MQe permiten a la aplicación proporcionar una identidad para un mensaje. Estos también se utilizan en interacción con el resto de la familia de MQ:

Java

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

C

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

La prioridad (*Prioridad*) contiene los valores de prioridad de los mensajes. La prioridad de los mensajes se define como en otros miembros de la familia MQ. Su valor oscila entre 9 (el más alto) y 0 (el más bajo):

Java

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

C

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

Las aplicaciones pueden crear campos para sus propios datos dentro de los mensajes:

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);
```

La prioridad del mensaje se utiliza, en parte, para controlar el orden en el que los mensajes se eliminan de la cola. Si el mensaje no especifica ninguno, se utiliza la prioridad predeterminada de la cola. Ésta, a no ser que se cambie, es 4. Sin embargo, la aplicación debe interpretar los diferentes niveles de prioridad.

En Java, se puede ampliar MQeMsgObject para incluir algunos métodos que ayudan a crear mensajes, como se muestra en el siguiente ejemplo:

```
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");
  }
```

Para saber cuál es la longitud de un mensaje, se puede enumerar el mensaje, puesto que cada tipo de datos dispone de métodos para poder obtener esta longitud.

Filtros de mensajes

Introducción a los filtros de mensajes de MQe

Los filtros permiten a MQe realizar búsquedas eficaces de mensajes. La mayor parte de las operaciones del gestor de colas dan soporte al uso de filtros. Se pueden crear filtros utilizando la clase MQeFields.

El uso de un filtro, por ejemplo, en una llamada getMessage(), hace que una aplicación devuelva el primer mensaje disponible que contiene los mismos campos y valores que el filtro. En el siguiente ejemplo, se crea un filtro que obtiene el primer mensaje con el ID de mensaje 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);

Este filtro se puede utilizar como un parámetro de entrada para varias llamadas API, por ejemplo `getMessage`.

Caducidad de los mensajes

Visión general de la caducidad de los mensajes en las colas

Las colas se pueden definir con un intervalo de caducidad. Si un mensaje ha permanecido en una cola durante un período de tiempo superior a este intervalo, el mensaje se suprimirá automáticamente. Cuando un mensaje se suprime, se llama a una norma de colas. Esta norma no puede afectar a la supresión del mensaje, pero facilita una oportunidad para crear una copia del mensaje.

Los mensajes también pueden tener un intervalo de caducidad que prevalezca sobre el intervalo de caducidad de la cola. Éste se puede definir añadiendo un campo C `MQE_MSG_EXPIRETIME` o un campo Java `MQe.Msg_ExpireTime` al mensaje. El período de caducidad es relativo (caduca 2 días después de la creación del mensaje), o absoluto (caduca el 25 de Noviembre de 2000 a las 08:00 horas). Los períodos de caducidad relativa son campos de tipo Int o MQEINT32, y los períodos de caducidad absoluta son campos de tipo Long o MQEINT64.

En el siguiente ejemplo, el mensaje caduca a los 60 segundos de haberse creado (60000 milisegundos = 60 segundos).

```
/* 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 );
```

En el ejemplo siguiente, el mensaje caducará el 15 de mayo de 2001, a las 15:25.

```
/* 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 );
```

Para establecer un tiempo de caducidad relativo utilice lo siguiente en un manejador de mensaje:

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

Para establecer un tiempo de caducidad absoluto utilice lo siguiente:

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

Todos los tiempos se expresan en milisegundos.

Comprobación de mensajes caducados:

Explicación sobre cuándo MQe comprueba si existen mensajes caducados

Se comprueba si un mensaje ha caducado cuando:

Se añade a una cola
Un mensaje puede caducar cuando se añade de la API local, extraído a través de una cola de servidor local o empujado a una cola.

Se elimina de una cola
Un mensaje puede caducar cuando se elimina de la API local o cuando se extrae de forma remota.

Una cola se activa
Cuando se activa una cola, se crea una referencia a la cola en la memoria. Cualquier mensaje que haya caducado se elimina. El estado del mensaje es importante para esta operación.

Una cola se suprime
Si llega un mensaje de administración para suprimir una cola, la cola primero debe estar vacía. Por lo tanto, antes de que se realice la operación, los mensajes caducados se eliminan de la cola. El estado del mensaje es importante para esta operación.

Se comprueba el tamaño de una cola
Si llega un mensaje de administración para consultar el tamaño de una cola, se depuran primero los mensajes de administración de la cola.

Puede añadir una norma para recibir notificaciones cuando los mensajes caduquen. Sin embargo, en algunas situaciones entre dos gestores de colas, puede parecer que un mensaje caduca dos veces. Esto no se debe a que el mensaje se haya duplicado, sino a lo que se señala en el párrafo siguiente.

Suponga que una cola asíncrona tiene un mensaje que va a caducar el 1 de enero de 2005 a las 10:00. Todos los mensajes de esas colas se transmite con un proceso de fase 2. Este proceso es equivalente a un par de operaciones putMessage y confirmPutMessage. Suponga que la primera etapa de transmisión tiene lugar a las 09:55. En el gestor de colas remoto aparece una referencia al mensaje. No obstante, aún no está disponible para una aplicación en ese gestor de colas. Por lo tanto, si la red falla hasta las 10:05, se salta la caducidad del mensaje y el mensaje caduca, pues, en la cola remota y la norma de caducidad se transmite. Así mismo, normalmente, la norma de caducidad de la cola se transmite en el gestor de colas de destino.

Garantía de caducidad:

Explica cómo garantizar la caducidad de los mensajes

Se puede calcular la hora de caducidad hasta el milisegundo. Para un funcionamiento correcto, los relojes de las máquinas que ejecuten los gestores de colas deben estar totalmente sincronizados. Si no se sincronizan con la precisión que determine la selección de los tiempos de caducidad, los mensajes aparecerán activos en un gestor de colas, mientras que, en otros, habrán caducado. Asegúrese de utilizar el tipo de campo correcto para el valor de caducidad. Se utiliza un campo int (32 bits) para los tiempos de caducidad relativos y se utiliza un campo long (64 bits) para los tiempos absolutos. El nombre del campo es el mismo en los dos casos.

MQeFields

Visión general de la estructura de contenedores de MQeFields

MQeFields es una estructura de datos de contenedor utilizada ampliamente en MQe. Se pueden colocar varios tipos de datos en el contenedor. Es particularmente útil para representar datos que se tienen que transportar, como el caso de los mensajes. El siguiente código crea una estructura de MQeFields:

Código Java

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

Código C

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

MQeFields contiene una recopilación de campos desordenados. Cada uno de los campos consiste en un trío que consta de un nombre de entrada, valor de entrada y tipo de entrada. MQeFields constituye la base de todos los mensajes de MQe.

Utilice el nombre de la entidad para recuperar y actualizar los valores. Se aconseja utilizar nombres cortos porque los nombres se incluyen en los datos cuando el elemento MQeFields se transmite.

El nombre debe:

- Tener una longitud mínima de un carácter
- Ajustarse al conjunto de caracteres ASCII (caracteres con valores 20 < valor < 128)
- Excluir cualquiera de los caracteres { } [ ] # ( ) : ; , ' " =
- Ser único dentro de MQeFields

Almacenamiento y recuperación de valores en MQeFields

Ejemplos de almacenamiento de valores en un elemento de MQeFields y recuperación de valores de un elemento MQeFields

En el siguiente ejemplo se muestra cómo almacenar valores en un elemento MQeFields:

Código Java

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

Código C

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

En el siguiente ejemplo se muestra cómo recuperar valores desde un elemento MQeFields:

Código Java

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

Código C

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

MQe proporciona métodos para almacenar y recuperar los siguientes tipos de datos:

- Las matrices de longitud fijas se manejan con los métodos putArrayOf*tipo* y getArrayOf*tipo*, donde *tipo* puede ser Byte, Short, Int, Long, Float o Double.
- La función de guardar matrices de longitud variable es posible, pero ya no se utiliza en este release. Puede acceder a estas matrices mediante las llamadas de tipo Java putArray y get**Array** o las llamadas de tipo C put.
- La base de código en Java tiene un formato de operaciones un poco especial para los tipos Float y Double. Proporciona compatibilidad con MicroEdition. Los tipos Float se transfieren utilizando una representación Int y los tipos Double se transfieren utilizando una representación Long. Utilice `Float.floatToIntBits()` y `Double.doubleToLongBits()` para realizar la conversión.No obstante, esto no es necesario en la API de C.

Inclusión de elementos de MQeFields

Descripción sobre cómo se incluye un elemento de MQeFields dentro de otro elemento de MQeFields

Un elemento de MQeFields se puede incluir dentro de otro elemento de MQeFields mediante los métodos putFields y getFields.

El contenido de un elemento MQeFields se puede volcar con uno de los siguientes formatos:

binario

El formato binario se utiliza normalmente para enviar un objeto MQeFields o MQeMsgObject a través de la red. El método dump convierte los datos en binarios. Este método devuelve una matriz de bytes binarios con un formato codificado del contenido del elemento.

Nota: No se trata de una serialización en Java.

Cuando se vuelca una matriz de longitud fija y la matriz no contiene elementos (su longitud es cero), su valor se restaura como nulo.

serie de caracteres codificada (sólo en Java)

El formato de serie de caracteres utiliza el método dumpToString del elemento de MQeFields. Necesita dos parámetros, una plantilla y un título. La plantilla es una serie de caracteres de tipo patrón que muestra cómo se deben convertir los datos del elemento MQeFields, tal y como se indica en el ejemplo siguiente:

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

donde

#0 es el tipo de datos (ascii o short, por ejemplo)

#1 es el nombre de campo

#2 es la representación del valor en la serie

Los demás caracteres se copian sin modificar en la serie de salida. El método vuelca correctamente objetos incluidos de MQeFields en una serie de caracteres, pero debido a restricciones, es posible que los datos incrustados de MQeFields no se restauren con el método restoreFromString.

Colas

Visión general de las colas de MQe

¿Qué son las colas de MQe?

Introducción a las colas de MQe

Las colas de MQe almacenan mensajes. Una aplicación no puede ver directamente las colas y todas las interacciones con las colas tienen lugar a través de los gestores de colas. Todo gestor de colas puede poseer y gestionar colas. Estas colas se denominan colas *locales*. MQe también permite que las aplicaciones accedan a mensajes de las colas que pertenecen a otro gestor de colas. Estas colas se denominan colas *remotas*. Hay conjuntos similares de operaciones que están disponibles en las colas locales y remotas, a excepción de la definición de escuchas de mensajes. Consulte el apartado “Escuchas de mensajes” en la página 46 para obtener más información. En el apartado Tipos de colas se proporciona más información sobre los diferentes tipos de colas que se pueden tener.

Los mensajes se retienen en el almacenamiento permanente de la cola. Una cola accede a su almacenamiento permanente a través de un *adaptador de almacenamiento de la cola*. Dichos adaptadores son interfaces entre MQe y los dispositivos de hardware, como por ejemplo discos o redes, o

almacenamientos de software, como una base de datos. Los adaptadores están diseñados para actuar como componentes ″conectables″, lo que permite modificar fácilmente los protocolos disponibles para las comunicaciones con el dispositivo.

Es posible que las colas presenten características como la autentificación, compresión y cifrado. Estas características se utilizan cuando un objeto de mensaje está almacenado en una cola.

Nombres de colas

Limitaciones de los nombres de colas de MQe

Los nombres de colas de MQe pueden contener los caracteres siguientes:

- Numéricos del 0 al 9
- Minúsculas de la ″a″ a la ″z″
- Mayúsculas de la ″A″ a la ″Z″
- Símbolo de subrayado (_)
- Punto (.)
- Tanto por ciento (%)

No existe ninguna limitación de longitud de nombre inherente en MQe.

Las colas se configuran mediante la utilización de mensajes de administración.

Propiedades de las colas

Tabla de propiedades de colas de MQe

En la siguiente tabla se muestran las propiedades de las colas. No todas las propiedades indicadas se aplican a todos los tipos de colas:

Nombre de campo proporcionado como serie estática de caracteres	**Serie estática de caracteres en C: MQe_Queue_Constants.h**	**Descripción**	**Clase que se tiene que utilizar para el valor de MQeField**	**Valor de la serie estática de caracteres para el nombre del campo**
Admin_Class		Clase de cola	String (serie)	admtype
Admin_Name		Nombre de cola ASCII	String (serie)	admname
Queue_Active	MQE_QUEUE_ACTIVE	Cola en estado activo/inactivo	boolean (booleano)	qact
Queue_AttRule		Clase de norma que controla las operaciones de seguridad	String (serie)	qar
Queue _Authenticator	MQE_QUEUE_AUTHENTICATOR	Clase de autentificador	String (serie)	qau
Queue_BridgeName		Propiedad del nombre de puente de MQ; sólo puente	String (serie)	q-mq-bridge
Queue_Client-Connection		Nombre de la conexión de cliente; sólo puente	String (serie)	q-mq-client-con

Nombre de campo proporcionado como serie estática de caracteres	Serie estática de caracteres en C: MQe_Queue_Constants.h	Descripción	Clase que se tiene que utilizar para el valor de MQeField	Valor de la serie estática de caracteres para el nombre del campo
Queue_CloseIdle		Cierra la conexión con el gestor de colas remoto cuando todos los mensajes se hayan transmitido	boolean (booleano)	qcwi
Queue_CreationDate	MQE_QUEUE_CREATIONDATE	Fecha en la que se ha creado la cola	long (entero largo)	qcd
Queue_Compressor	MQE_QUEUE_COMPRESSOR	Clase de compresor	String (serie)	qco
Queue_Cryptor	MQE_QUEUE_CRYPTOR	Clase de cifrador	String (serie)	qcr
Queue_CurrentSize	MQE_QUEUE_CURRENTSIZE	Número de mensajes en la cola	int (entero)	qcs
Queue_Description	MQE_QUEUE_DESCRIPTION	Descripción Unicode	String (serie)	qd
Queue_Expiry	MQE_QUEUE_EXPIRY	Fecha de caducidad para los mensajes		qe
Queue_FileDesc	MQE_QUEUE_FILEDESC	Descriptor de archivo: especifica el tipo de almacén de mensajes	String (serie)	qfd
Queue_MaxIdletime		Tiempo máximo durante el que se debe mantener una conexión inactiva; sólo puente	int (entero)	q-mq-max-idle-time
Queue_MaxMsgSize	MQE_QUEUE_MAXMSGSIZE MQE_QUEUE_NOLIMIT	Longitud máxima de los mensajes permitidos en la cola	int (entero)	qms
Queue_MaxQSize	MQE_QUEUE_MAXQSIZE MQE_QUEUE_NOLIMIT	Número máximo de mensajes permitidos	int (entero)	qmqs
Queue_Mode	MQE_QUEUE_MODE MQE_QUEUE_SYNCHRONOUS MQE_QUEUE_ASYNCHRONOUS	Síncrono o asíncrono	byte Queue_Synchron-ous Queue_Asynchro-nous	qm
Queue_MQQMgr		Proxy de gestor de colas de MQ; sólo puente	String (serie)	q-mq-q-mgr
Queue_Priority	MQE_QUEUE_PRIORITY	Prioridad que se debe utilizar para los mensajes (a no ser que se altere temporalmente por el valor de un mensaje)	byte	qp
Queue_QAlias-NameList	MQE_QUEUE_QALIASNAMELIST	Nombres alternativos de la cola	String[]	qanl

Nombre de campo proporcionado como serie estática de caracteres	Serie estática de caracteres en C: MQe_Queue_Constants.h	Descripción	Clase que se tiene que utilizar para el valor de MQeField	Valor de la serie estática de caracteres para el nombre del campo
Queue_QMgrName	MQE_QUEUE_QMGRNAME	Gestor de colas poseedor de la cola real	String (serie)	qqmn
Queue_QMgr-NameList	MQE_QUEUE_QMGRNAMELIST: sólo para administración; C no da soporte a las colas de almacenes.	Destinos del gestor de colas: se utiliza en colas de almacenes	String[] -	qqmnl
Queue_Remote-QName		Nombre de campo de MQ remoto; sólo puente	String (serie)	q-mq-remote-q
Queue_Rule		Clase de norma para las propiedades de colas	String (serie)	qr
Queue_QTimer-Interval		Intervalo de tiempo antes de procesar los mensajes de tiempo en la cola del servidor local: utilice una norma para la transmisión de desencadenantes en su lugar *	long (entero largo)	qti
Queue_Target-Registry	MQE_QUEUE_TARGETREGISTRY	Tipo de registro de destino	Valores posibles de String[]: Queue_Registry-None Queue_Registry-QMgr Queue_Registry-Queue	qtr
Queue_Transporter	MQE_QUEUE_TRANSPORTER MQE_QUEUE_DEFAULT-TRANSPORTER	Clase de transportador	String; utilice Queue_Default-Transporter	qtc
Queue_Transporter-XOR		Transportador para utilizar la compresión XOR	boolean (booleano)	qtxor
Queue_Transformer		Clase de transformador	String (serie)	q-mq-transformer

* Si se utiliza un intervalo de temporizador en la cola del servidor local en el caso de que se produzca un error, la aplicación no sabe que la hebra se ha detenido y, por lo tanto, no puede hacer nada al respecto. En su lugar, el intervalo de temporizador debe establecerse en cero y se debe utilizar una norma en el gestor de colas para crear un bucle y llamar de forma explícita a triggerTransmission(). Es aconsejable no establecer un bucle demasiado ajustado, sino establecer el temporizador del bucle en un valor adecuado para que los mensajes se sigan enviando/recuperando sin que se deba utilizar una CPU ajena.

Tipos de colas

Introducción a los tipos de colas de MQe

Existen varios tipos de *colas* distintos que se pueden utilizar en un entorno de MQe.

Cola local

El tipo más sencillo de cola es una cola local. Este tipo de colas es local para un gestor de colas específico que es su propietario. Se trata del destino final de todos los mensajes. Las aplicaciones del gestor de colas propietario pueden interactuar directamente con la cola para almacenar mensajes de forma segura, evitando los casos de anomalías de hardware o de pérdidas del dispositivo.

Se pueden utilizar colas locales en línea o fuera de línea, conectadas o no a una red. Las colas también pueden disponer de un conjunto de atributos de seguridad, de un modo muy similar para proteger mensajes con atributos.

El acceso a mensajes en colas locales es siempre síncrono, lo cual significa que la aplicación espera hasta que se producen las devoluciones de MQe después de haber completado la operación, por ejemplo, put (de transferencia), get (de obtención) o browse (de examen).

La cola posee acceso y seguridad y podrá permitir a un gestor de colas remoto la utilización de estas características cuando esté conectada a una red. Esto permitirá que otros usuarios envíen o reciban mensajes de la cola.

Cola remota

Una cola remota es una cola local que pertenece a otro gestor de colas. Esta definición de cola remota intercambia mensajes con la cola local remota.

MQe puede establecer colas remotas automáticamente. Si intenta acceder a una cola ubicada en otro gestor de colas, por ejemplo, para enviar un mensaje a esa cola, MQe buscará una definición de cola remota. Si existe alguna, la utilizará. Si no es el caso, se producirá un *descubrimiento de cola*.

Nota: El concepto de descubrimiento de cola no se aplica a la base de código en C.

MQe descubre las características de compresión, cifrado y autentificación de la cola real y crea una definición de cola remota. Este descubrimiento de cola depende de si se puede acceder o no al destino. Si no se puede acceder al destino, la definición remota se debe suministrar de otro modo. Cuando se produce un descubrimiento de cola, MQe establece el modo de acceso en síncrono porque ya se sabe que la cola está disponible síncronamente.

Las colas remotas *síncronas* son colas a las que se puede acceder sólo cuando están conectadas a una red que se comunica con el gestor de colas propietario. Si la red no está establecida, las operaciones devuelven un error. La cola propietaria controla los permisos de acceso y los requisitos de seguridad necesarios para acceder a la cola. La aplicación se encarga de manejar cualquier error o reintento que se produzca al enviar mensajes porque, en ese caso, MQe deja de hacerse responsable de la entrega asegurada en una sola vez y sólo en una.

Las colas remotas *asíncronas* son colas que se utilizan para enviar mensajes a colas remotas y que pueden almacenar mensajes pendientes de transmisión. Estas colas no pueden recuperar mensajes de forma remota. Si la conexión de red se ha establecido, los mensajes se envían a la cola y al gestor de colas propietario. No obstante, si la red no está conectada, los mensajes se almacenan localmente hasta que hay una conexión de red y, entonces, los mensajes se transmiten. Esto permite a las aplicaciones operar en la cola cuando el dispositivo está fuera de línea. El resultado es que estas colas almacenan temporalmente mensajes en el gestor de colas emisor, mientras esperan que se lleve a cabo la transmisión.

Cola de almacenamiento y reenvío

Nota: Las colas de almacenamiento y reenvío no se implementan en la base de código en C.

Una cola de almacenamiento y reenvío almacena mensajes en nombre de uno o más gestores de colas remotos hasta que estén preparados para recibirlos. Esto se puede configurar para realizar cualquiera de los pasos siguientes:

- Enviar mensajes al gestor de colas de destino o a otro gestor de colas situado entre los gestores de colas emisores y de destino.
- Esperar a que el gestor de colas de destino extraiga los mensajes destinados a él.

Una cola de almacenamiento y reenvío almacena mensajes asociados a uno o más destinos del gestor de colas de destino. Los mensajes dirigidos a un gestor de colas de destino o específico se colocan en la cola de almacenamiento y reenvío. La cola de almacenamiento y reenvío puede disponer opcionalmente de un nombre de gestores de colas de reenvío establecido. Si este nombre está establecido, la cola intenta enviar todos sus mensajes a ese gestor de colas mencionado. Si el nombre no está establecido, la cola simplemente retiene los mensajes.

Nota: Una cola de almacenamiento y reenvío y una *cola de servidor local* no deben tener el mismo gestor de colas de destino. Una cola de almacenamiento y reenvío con una cola QueueManagerName que no es la misma que su sistema principal QueueManagerName, intentará empujar mensajes al gestor de colas remoto. Si este gestor de colas remoto tiene una cola de servidor local, es posible que intente extraer el mismo mensaje simultáneamente, provocando el bloqueo del mensaje.

Las colas para almacenar y reenviar pueden alojar mensajes para muchos gestores de colas de destino, y puede que haya una cola para almacenar y reenviar para cada gestor de colas de destino.

Este tipo de colas está definido normalmente, pero no necesariamente, en un servidor o en una pasarela únicamente en Java. En un único gestor de colas puede haber múltiples colas de almacenamiento y reenvío, pero los nombres de destino no se deben duplicar. El contenido de una cola de almacenamiento y reenvío no está disponible para programas de aplicación. Asimismo, una aplicación de envío de mensajes desconoce la presencia o la función de las colas de almacenamiento y reenvío en la transmisión de mensajes.

Colas de mensajes no entregados

MQe tiene un concepto de cola de mensajes no entregados parecido al de MQ. Estas colas almacenan mensajes que no se pueden entregar. No obstante, existen diferencias importantes en la forma en que se utilizan.

- En MQ, si un mensaje se traslada del gestor de colas `A` al gestor de colas `B` y, a continuación, no se encuentra la cola de destino en el gestor de colas `B`, el mensaje se puede colocar en la cola de mensajes no entregados del *gestor de colas receptor* (el `B`).
- En MQe, si la cola del servidor local de un cliente extrae un mensaje de un servidor y no puede entregar el mensaje a una cola local y el cliente dispone de una cola de mensajes no entregados, el mensaje se colocará en la cola de mensajes no entregados del cliente.

  Nota: en C, la cola de mensajes no entregados es una cola local con un nombre específico.

  La utilización de colas de mensajes no entregados con un puente de MQ necesita especial atención. Para obtener más información, consulte “Cola de puente de MQ” en la página 14.

Cola de administración

La cola de administración es una cola especializada que procesa los mensajes de administración.

Los mensajes que se transfieren a la cola de administración se procesan internamente. Debido a esto, las aplicaciones no pueden obtener mensajes directamente de la cola de administración. Sólo se procesa un mensaje cada vez, los demás mensajes que llegan mientras se está procesando un mensaje se colocan en cola y se procesan según el orden de llegada.

Cola del servidor inicial

Normalmente las colas de este tipo residen en un cliente y apuntan a una cola para almacenar y reenviar de un servidor denominado *servidor local*. Esta cola extrae los mensajes de la cola para almacenar y reenviar del servidor local cuando el cliente se conecta a la red.

En Java, las colas del servidor local normalmente tienen establecido un intervalo de sondeo que hace que se compruebe la existencia de mensajes pendientes en el servidor mientras se está conectado a la red.

Cuando esta cola extrae un mensaje del servidor, utiliza la entrega asegurada de mensajes para transferir el mensaje al gestor de colas local. A continuación, el mensaje se almacena en la cola de destino.

Las colas del servidor local tienen una función importante al permitir a los clientes recibir mensajes a través de conexiones entre cliente y servidor.

Cola de puente de MQ

Nota: la base de código en C no da soporte a las colas puente de MQ.

Este tipo de colas siempre está definido en un gestor de colas de pasarela de MQe y proporciona una vía de acceso del entorno de MQe al entorno de MQ. La cola puente MQ es una definición de cola remota que hace referencia a una cola ubicada en un gestor de colas de MQ.

Las aplicaciones pueden utilizar las operaciones **put**, **get** y **browse** en este tipo de colas, como si fuera una cola de MQe local.

Almacenamiento permanente de cola

Visión general de almacenes de mensajes de MQe

Las colas locales y las colas remotas asíncronas almacenan mensajes y, por lo tanto, tienen propiedades para determinar cómo y dónde se almacenan los mensajes.

El almacenamiento de mensajes determina cómo se correlacionan los mensajes al medio de almacenamiento. Las versiones en C y Java del soporte de MQe dan soporte a un almacén de mensajes predeterminado, lo que permite la utilización de nombres de archivo largos. La versión en Java de MQe tiene dos almacenes de mensaje adicionales, `MQeShortFilenameMessageStore` que asegura que el nombre de archivo no supere los ocho caracteres y `MQe4690ShortFilenameMessageStore` que da soporte al sistema de archivos predeterminado en un 4690. El adaptador de almacenamiento proporciona el acceso al almacén de mensajes al soporte de almacenamiento, la versión en Java y en C de MQe proporcionan adaptadores de disco y con la versión en Java también se proporciona un adaptador que no distingue entre mayúsculas y minúsculas y un adaptador de memoria.

El almacén de copia de seguridad que utiliza una cola se puede modificar mediante un mensaje de administración de MQe. Sin embargo, no está permitido modificar el almacenamiento de copia de seguridad si la cola está activa o si contiene mensajes. Si el almacenamiento de copia de seguridad que utiliza la cola permite recuperar mensajes en caso de una anomalía del sistema, MQe puede asegurar la entrega de los mensajes.

Utilización de alias de colas

Presenta el uso de los alias de colas.

Pueden asignarse alias para las colas de MQe a fin de proporcionar un nivel de direccionamiento indirecto entre la aplicación y las colas reales. De este modo, los atributos de una cola a la que hace referencia un alias se pueden modificar sin que sea necesario modificar la aplicación. Por ejemplo, se puede dar a una cola un número de alias y los mensajes enviados a cualquiera de estos nombres serán aceptados por la cola.

Ejemplos de asignación de alias de cola

Muestra algunas de las maneras en las que se puede utilizar la asignación de alias con las colas.

Los ejemplos siguientes muestran algunas de las maneras en que se puede utilizar la asignación de alias con las colas:

Fusión de aplicaciones:

Utilización de la asignación de alias de colas para fusionar aplicaciones

Imagine que tiene la siguiente configuración:

- Una aplicación cliente que coloca los datos en la cola `Q1`
- Una aplicación de servidor que toma los datos de `Q1` para su proceso
- Una aplicación cliente que coloca los datos en la cola `Q2`
- Una aplicación de servidor que toma los datos de `Q2` para su proceso

Poco después, las dos aplicaciones de servidor se fusionan en una aplicación que da soporte a las peticiones de las dos aplicaciones cliente. Entonces podría ser el momento adecuado para que las dos colas se convirtieran en una. Por ejemplo, podría suprimir `Q2` y añadir un alias de la cola `Q1`, denominándola `Q2`. Los mensajes de la aplicación cliente que anteriormente utilizaron `Q2` se envían de manera automática a `Q1`.

Actualización de aplicaciones:

Utilización de la asignación de alias de colas para actualizar aplicaciones

Imagine que tiene la siguiente configuración:

- Una cola `Q1`
- Una aplicación que obtiene mensajes de `Q1`
- Una aplicación que coloca mensajes en `Q1`

A continuación, desarrolle una nueva versión de la aplicación que obtiene los mensajes. Puede hacer que la nueva aplicación funcione con una cola llamada `Q2`. Puede definir una cola llamada `Q2` y utilizarla para practicar con la nueva aplicación. Cuando desee entrar en operaciones, deje que la versión anterior borre todo el tráfico de la cola `Q1` y, a continuación, cree un alias de `Q2` denominado `Q1`. La aplicación que coloca los mensajes en `Q1` todavía funcionará, pero los mensajes acabarán en `Q2`.

Utilización de modalidades de transferencia distintas a una sola cola:

Utilización de modalidades de transferencia distintas a una sola cola mediante la asignación de alias de colas

Imagine que tiene una cola `MY_Q_ASYNC` en el gestor de colas `MQE1`. Los mensajes pasan a `MY_Q_ASYNC` por otro gestor de colas `MQE2`, utilizando una definición de cola remota que se define como cola asíncrona. Ahora imagine que periódicamente su aplicación desea obtener mensajes de manera síncrona de la cola `MY_Q_ASYNC`.

Para que así suceda, es recomendable añadir un alias a la cola `MY_Q_ASYNC`, denominada quizás `MY_Q_SYNC`. A continuación, defina una definición de cola remota en su gestor de colas `MQE2`, que hace referencia a la

cola MY_Q_SYNC. Esto le proporciona dos definiciones de cola remota. Si utiliza la definición MY_Q_ASYNC, los mensajes son transportados de manera asíncrona. Si utiliza la definición MY_Q_SYNC, se usa la transferencia de mensajes síncrona.

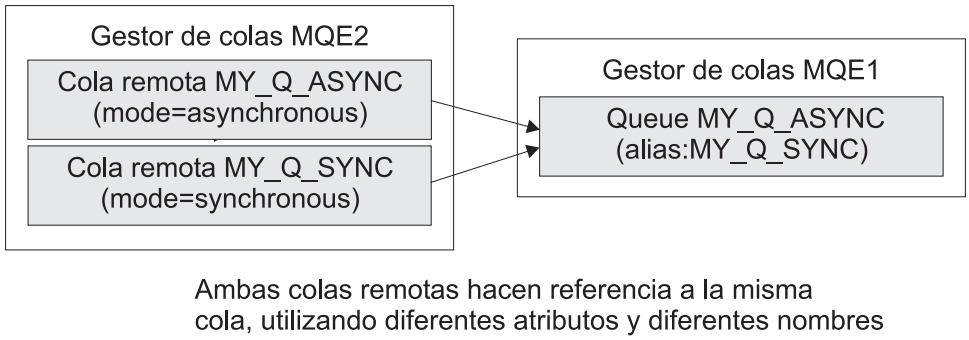

Figura 1. Dos modalidades de transferencia para una cola única

Definiciones de conexiones de MQe

Explica cómo se establecen conexiones lógicas entre gestores de colas.

MQe da soporte a un método de establecimiento de conexiones lógicas entre gestores de colas con el fin de enviar o recibir datos.

Los clientes y servidores de MQe se comunican a través de conexiones denominadas *canales de cliente/servidor*.

Los **canales de cliente/servidor** tienen los atributos siguientes:

- Son *dinámicos*, puesto que se crean bajo demanda. Esto los diferencia de las conexiones de MQ que tienen que crearse de forma explícita.
- Sólo puede establecer la conexión desde el lado cliente.
- Un cliente se puede conectar con muchos servidores y cada conexión utiliza un canal independiente.
- El gestor de colas de lado servidor puede aceptar muchas conexiones de forma simultánea de muchos clientes distintos, con un escucha para cada protocolo.
- Funcionan a través de un cortafuegos si el lado de servidor de la conexión se encuentra detrás del cortafuegos. Sin embargo, eso depende de la configuración del cortafuegos.
- Son *unidireccionales* y dan soporte a toda la gama de funciones que proporciona MQe, incluidas la mensajería síncrona y asíncrona.

  Nota: unidireccional significa que el cliente puede enviar datos al servidor o solicitarle datos, pero el área del servidor no puede iniciar peticiones del cliente.

Las conexiones estándar, utilizadas para el estilo de conexión de cliente/servidor, son unidireccionales, pero dependen de un escucha en el servidor, puesto que los servidores no pueden iniciar una transferencia de datos. El cliente inicia la petición de conexión y el servidor responde. Un servidor puede manejar normalmente múltiples peticiones de entrada de los clientes. En una conexión estándar, el cliente tiene acceso a los recursos del servidor. Si una aplicación en el servidor necesita un acceso síncrono a los recursos del cliente, se tendrá que establecer una segunda conexión en la que las funciones se inviertan. Sin embargo, puesto que las conexiones estándar son en sí mismas bidireccionales, los mensajes

destinados a un cliente desde la cola de transmisión de su servidor se entregan a ese mismo cliente mediante una conexión estándar (cliente/servidor) que éste ha iniciado.

Un cliente puede ser un cliente de muchos servidores simultáneamente. El estilo de la conexión de cliente/servidor está generalmente adaptado para poderla utilizar a través de cortafuegos, ya que el destino de la conexión de entrada se identifica normalmente como aceptable para el cortafuegos.

Nota: Supongamos que hay dos gestores de colas de servidor: SQM1 y SQM2. SQM2 dispone de un sistema principal de direcciones de escuchas 2: 8082. Asimismo, supongamos que SQM1 tiene una conexión a SQM2 y un sistema principal de direcciones de escuchas 1:8081. Si se crea una definición de conexión en un gestor de colas de cliente llamado SQM2, con sistema principal de direcciones 1: 8081, éste transportará mandatos para SQM2 a SQM1, que después los transportará a SQM2. Evite esta estructura porque no es eficaz.

Debido a la forma en que funciona la seguridad de canales, cuando una norma de atributo determinada se especifica para una cola de destino, se obliga al gestor de colas local a crear una instancia de la misma norma de atributo, de modo que se trata a la norma `examples.rules.AttributeRule` y a la norma `com.ibm.mqe.MQeAttributeRule` como si fueran la misma. Si no desea que se produzca este comportamiento, se puede especificar una norma no válida para la cola de destino. En este caso, la norma com.ibm.mqe.`MQeAttributeDefaultRule` es válida.

Las conexiones pueden tener varios atributos o características, como la autentificación, el cifrado, la compresión o el protocolo de transmisión que se va a utilizar. Las diferentes conexiones pueden utilizar distintas características. Cada conexión puede tener su propio conjunto de valores para cada uno de los siguientes atributos:

Autentificador
Este atributo hace que se lleve a cabo la autentificación. Se trata de una función de seguridad que reta al entorno de aplicación de colocación o al usuario a demostrar su identidad. Tiene un valor de `NULL` (nulo) o un *autentificador* que puede llevar a cabo la autentificación de conexiones o usuarios.

Cifrador
Este atributo hace que se lleve a cabo el cifrado y descifrado de los mensajes que pasan a través del canal. Se trata de una función de seguridad que codifica los mensajes durante su tránsito, de modo que no se pueden leer sin la información de descodificación. Tiene un valor de NULL (nulo) o un *cifrador* que puede ejecutar el cifrado y descifrado.

El tipo más simple de cifrador es MQeXorCryptor, que cifra los datos que se envían realizando una operación exclusiva OR de los datos. Este cifrado no es seguro, pero modifica los datos, de modo que no se pueden ver. Por el contrario, el cifrador MQe3DESCryptor implementa el cifrado DES triple, un método de cifrado de clave simétrica.

Canal
Clase que proporciona los servicios de transporte.

Compresor
Este atributo hace que se lleve a cabo la compresión y descompresión de los mensajes que pasan a través del canal. Esto tiene como finalidad reducir el tamaño de los mensajes mientras se transmiten y almacenan. Tiene un valor de NULL (nulo) o un *compresor* que puede ejecutar la compresión y descompresión de datos. El tipo de compresor más simple es el MQeRleCompressor, que comprime los datos sustituyendo los caracteres repetidos por un contador.

Destino
Servidor y número de puerto para la conexión. El destino para esta conexión, por ejemplo SERVER.XYZ.COM

Habitualmente, la autentificación sólo se produce cuando se prepara una conexión. Todos los flujos de comunicaciones utilizan normalmente compresores y cifradores.

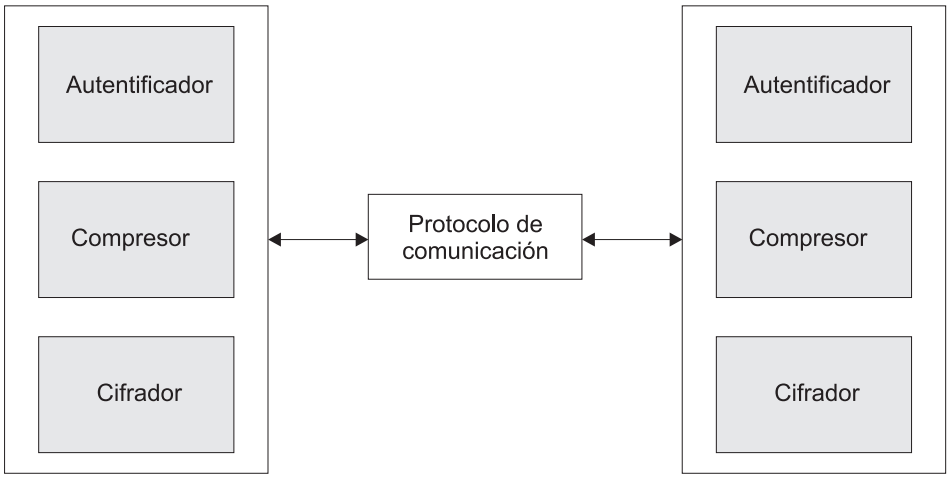

Figura 2. Conexión de MQe

Se pueden establecer conexiones de MQeQe utilizando una variedad de protocolos que les permita conectarse de diferentes maneras, por ejemplo:

- Conexión permanente, por ejemplo, una LAN o una línea alquilada
- Conexión de marcación, por ejemplo, un módem estándar para conectarse con un Proveedor de servicios de Internet (ISP)
- Conexión de marcación y respuesta, por ejemplo, utilizando CellPhone o ScreenPhone

MQe implementa los protocolos de comunicación como un conjunto de adaptadores, con un adaptador para cada uno de los protocolos a los que se da soporte. Esto permite añadir nuevos protocolos.

Operaciones de los gestores de colas

Explicación de las operaciones de mensajería que se pueden realizar en un gestor de colas.

En este tema se explica detalladamente cuáles son las operaciones de mensajería que se pueden realizar en un gestor de colas. Se describen los servicios, las funciones y los usos de los gestores de colas en los siguientes apartados:

¿Qué es un gestor de colas de MQe?

Introducción a las funciones y a la utilización de los gestores de colas

El gestor de colas de MQe es el punto focal del sistema MQe. Proporciona:

- Un punto central de acceso a una red de mensajería o gestión de colas para aplicaciones de MQe.
- Gestión de colas opcional desde el lado del cliente
- Funciones opcionales de administración
- Entrega segura de una sola vez de mensajes
- Recuperación en caso de anomalía
- Comportamiento basado en normas ampliable

A diferencia del producto base MQ, MQe tiene un solo tipo de gestor de colas. Sin embargo, los gestores de colas de MQe se pueden programar para que actúen como clientes o servidores tradicionales. Asimismo, se puede personalizar el comportamiento del gestor de colas utilizando normas. El gestor de colas de MQe se incorpora en los programas escritos por el usuario y estos programas se pueden ejecutar en cualquier dispositivo o plataforma soportado por MQe.

Se pueden configurar gestores de colas de distintas formas y las principales son las de cliente, servidor y pasarela. También se puede actualizar el almacén de colas de un gestor de colas mediante la utilización de mensajes de administración. Si desea ver más información sobre mensajes de administración, consulte la publicación MQe Configuration Guide.

La comunicación con otros gestores de colas en la red de mensajería de MQe puede ser síncrona o asíncrona. Si desea utilizar comunicaciones síncronas, los gestores de colas de MQe de origen y de destino deben estar disponibles en la red. La comunicación asíncrona permite que una aplicación de MQe envíe mensajes aunque el gestor de colas remoto esté fuera de línea.

Ciclo de vida del gestor de colas

Visión general del ciclo de vida de un gestor de colas

Normalmente, una aplicación crea un nuevo gestor de colas, lo configura con una cantidad de colas y, a continuación, lo libera. Una aplicación también abre un gestor de colas existente, lo inicia, efectúa operaciones de mensajería y a continuación se detiene. Otro programa de administración puede volver a abrir el gestor de colas, eliminar todas sus colas y, a continuación, detenerse. En el siguiente diagrama se muestra esta información:

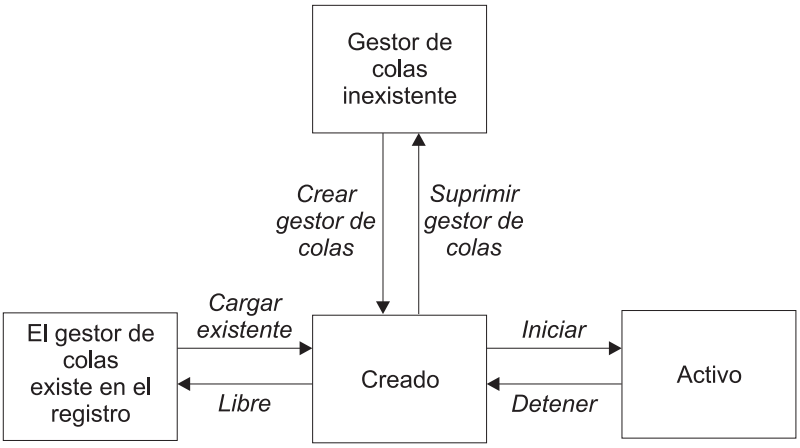

Figura 3. Ciclo de vida del gestor de colas

Creación de gestores de colas

Un gestor de colas requiere como mínimo lo siguiente:

- Un registro
- Una definición de gestor de colas
- Definiciones de colas locales predeterminadas

Una vez ubicadas las definiciones, se puede ejecutar el gestor de colas y se puede realizar cualquier configuración adicional como, por ejemplo, añadir más colas, utilizando la interfaz de administración.

En la clase MQeQueueManagerConfigure se proporcionan métodos para crear estos objetos iniciales.

Los programas de instalación de ejemplo `examples.install.SimpleCreateQM` y `examples.install.SimpleDeleteQM` utilizan esta clase.

Nombres del gestor de colas

Los nombres de los gestores de colas de MQe pueden contener los siguientes caracteres:

- Numéricos del 0 al 9

- Minúsculas de la ″a″ a la ″z″
- Mayúsculas de la ″A″ a la ″Z″
- Símbolo de subrayado (_)
- Punto (.)
- Tanto por ciento (%)

No existe ninguna limitación de longitud de nombre inherente en MQe.

Creación de un gestor de colas - paso a paso

Los pasos básicos necesarios para crear un gestor de colas son:

1. Crear y activar una instancia de MQeQueueManagerConfigure
2. Establecer las propiedades del gestor de colas y crear la definición del gestor de colas
3. Crear definiciones para las colas predeterminadas
4. Cerrar la instancia de MQeQueueManagerConfigure

Crear y activar una instancia de MQeQueueManagerConfigure:

Para crear el objeto MQeQueueManagerConfigure debe invocar el método mqeQueueManagerConfigure_new. Aparte de *ExceptionBlock* y del nuevo descriptor de contexto MQeQueueManagerConfigure , este método incluye dos parámetros adicionales.

El método de funcionamiento depende de estos parámetros. Se puede pasar ″NULL″ para estos parámetros, en cuyo caso se invoca mqeQueueManagerConfigure_activate inmediatamente después de mqeQueueManagerConfigure_new. De forma alternativa, se pueden pasar parámetros de inicio.

Se puede activar la clase MQeQueueManagerConfigure de cualquiera de las formas siguientes:

1. Invoque al constructor vacío seguido de activate():

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

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

2. Invoque al constructor con los parámetros:

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

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

El primer parámetro es un objeto MQeFields que contiene los parámetros de inicialización del gestor de colas. Deben contener, como mínimo lo siguiente:

- Un objeto MQeFields incrustado (*Name*) que contiene el nombre del gestor de colas.

- Un objeto MQeFields incrustado, que contiene la ubicación del almacén de colas locales como el tipo de registro (*LocalRegType*) y el nombre del directorio de registro (*DirName*). Si se utiliza un registro de archivo base, éstos son los parámetros necesarios. Si se utiliza un registro privado, también serán necesarios un *PIN* y *KeyRingPassword*.

El nombre del directorio se almacena como parte de la definición del gestor de colas y se utiliza como valor predeterminado para el almacenamiento de las colas en las definiciones de colas posteriores. No es necesario que exista el directorio y se creará cuando sea necesario.

Si utiliza un alias para cualquiera de los parámetros de inicialización o si desea utilizar un alias para establecer el nombre de la norma de los atributos de conexión, los alias se deberán definir antes de activar 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)
{ ... }
```

El código de ejemplo incluye la creación de una instancia de MQeQueueManagerConfigure.

Establecer las propiedades del gestor de colas:

Cuando haya activado MQeQueueManagerConfigure, pero antes de crear la definición del gestor de colas, podrá establecer parte o todas las propiedades siguientes del gestor de colas:

- Puede añadir una descripción al gestor de colas con mqeQueueManagerConfigure_setDescription().
- Puede establecer un valor de tiempo de espera de conexión con mqeQueueManagerConfigure_setChannelTimeout().
- Puede establecer el nombre de la norma de atributos de conexión con mqeQueueManagerConfigure_setChnlAttributeRuleName().

Invoque a mqeQueueManagerConfigure_defineQueueManager() para crear la definición del gestor de colas. Esto creará una definición del registro para el gestor de colas que incluye alguna de las propiedades que se han establecido anteriormente.

```
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)
{ ... }
```

En este punto puede invocar `close()` y `free()` MQeQueueManagerConfigure y ejecutar el gestor de colas, aunque éste no puede hacer mucho al respecto porque no tiene colas. Ni siquiera puede añadir colas utilizando la interfaz de administración, porque no posee una cola de administración con la que dar servicio a los mensajes de administración.

Los apartados siguientes muestran cómo crear colas y hacer que el gestor de colas resulte útil.

Crear definiciones para las colas predeterminadas:

MQeQueueManagerConfigure le permite definir cuatro colas estándar para el gestor de colas:

defineDefaultAdminQueue()mqeQueueManagerConfigure_
: Esta cola de administración es necesaria para que el gestor de colas pueda responder a los mensajes de administración, por ejemplo, para crear nuevas colas y definiciones de conexión.

defineDefaultAdminReplyQueue()mqeQueueManagerConfigure_
: Esta cola de respuestas de administración es una cola local que utilizan las conexiones como destino de los mensajes de respuesta generados por la administración.

defineDefaultDeadLetterQueue()mqeQueueManagerConfigure_
: Esta cola de mensajes no entregados se puede utilizar, en función de las normas vigentes, para almacenar mensajes que no se pueden entregar a su destino correcto.

defineDefaultSystemQueue()mqeQueueManagerConfigure_
: Esta cola local predeterminada, SYSTEM.DEFAULT.LOCAL.QUEUE, no tiene una especial relevancia dentro del propio MQe, pero es útil cuando MQe se utiliza con el servicio de mensajería de MQ porque está presente en cada uno de los gestores de colas de mensajería de MQ.

Todos los métodos generan una excepcióndevuelven un error si la cola ya existe.

```
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)
{ ... }
```

Cerrar la instancia MQeQueueManagerConfigure:

Una vez definidos el gestor de colas y las colas necesarias, se puede ejecutar close() MQeQueueManagerConfigure y ejecutar el gestor de colas.

El ejemplo completo es similar al siguiente:

```
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)
{ ... }
```

Inmediatamente, se crean las definiciones del registro para el gestor de colas y las colas necesarias. Las colas no se crean hasta que no se hayan activado.

Datos de configuración permanentes

Los gestores de colas de MQe, independientemente de su rol en la red MQe, exigen información para poderlos mantener en el almacenamiento permanente. Esto es responsabilidad de MQe. Si hay información adicional que debe permanecer entre invocaciones de una aplicación, esto es responsabilidad de la aplicación.

La información que se mantiene en el registro contiene los detalles de configuración del gestor de colas, por ejemplo:

- Información sobre dónde se mantienen mensajes, colas, definiciones de colas remotas, tiempo de espera de canales, alias, adaptadores y almacén de mensajes y sobre cómo acceder a ellos
- Definiciones de conexiones
- Información de seguridad
- Objetos diversos relacionados con puentes

En este manual la información permanente siguiente, útil para una aplicación, se denomina datos de entorno:

- Información de registro, clase, vía de acceso, clase de adaptador de almacenamiento y tipo de registro. Esta información se utiliza para localizar un registro existente, lo que permite a MQe iniciar un gestor de colas existente o crear un nuevo registro de gestor de colas.

- Información de gestor de clases, por ejemplo la clase y el nombre.
- Tipo de gestor de colas.

Creación de gestores de colas sencillos

El gestor de colas de MQe más sencillo es un gestor de colas que utiliza un registro basado en los valores predeterminados internos. El gestor de colas se podría crear sin colas, pero sus funciones quedarían muy limitadas. En nuestro ejemplo hay cuatro colas estándar:

- Cola de administración: para que se puede realizar la administración.
- Cola de respuesta de administración: ubicación estándar en la que almacenar las respuestas de las acciones de administración.
- Cola predeterminada del sistema: cola local general de gran utilidad.
- Cola de mensajes no entregados: ubicación para los mensajes no entregados.

El gestor de colas más sencillo no tiene seguridad y tiene un registro almacenado en el sistema de archivos local. Los pasos que se deben llevar a cabo son:

- Crear un registro en el disco
- Crear y reiniciar un gestor de colas con el registro
- Detener el gestor de colas

Estas acciones se describen tanto para la base de código en Java como la base de código en C, con código de ejemplo para cada uno. El código de ejemplo en Java se proporciona como examples.config.CreateQueueManager. Para obtener un código de ejemplo en C, consulte el apartado de compilación de HelloWorld y el archivo transport-c en el ejemplo de intermediario.

Creación de un gestor de colas sencillo en Java:

Los registros se crean en Java mediante la clase com.ibm.mqe.MQeQueueManagerConfigure. Se crea una instancia de esta clase, que se activa pasándole algunos parámetros de inicialización. Los parámetros se proporcionan en formato de objeto MQeFields. Estos MQeFields contienen dos subcampos, uno que contiene información sobre el registro y el otro que contiene información acerca del gestor de colas que se va a crear. Como estamos creando un gestor de colas muy sencillo, sólo tenemos que pasar dos parámetros: el nombre del gestor de colas, en los parámetros del gestor de colas, y la ubicación del registro, en los parámetros de registro. A continuación, podremos utilizar MQeQueue ManagerConfigure para crear las colas estándar.

En primer lugar, cree tres objetos de campo: uno para los parámetros de QueueManager, otro para los parámetros de registro y el tercero, parms, se utiliza para contener tanto los objetos de campo del gestor de colas como del registro.

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

Debe establecerse el nombre del gestor de colas. Utilice MQeQueueManager.Name como constante de etiqueta de campo.

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

Se debe especificar la ubicación del registro permanente. Hágalo en el objeto de campo de parámetros de registro. Utilice MQeRegistry.DirName como constante de etiqueta de campo.

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

Los parámetros del gestor de colas y del registro ahora se pueden incorporar en los objetos de campo principales.

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

Ahora se puede crear una instancia de MQeQueueManagerConfigure. Esto exige el objeto de campo de parámetros, más una serie de caracteres que identifique los detalles del almacén de colas que se debe utilizar.

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

Se pueden crear los cuatro tipos comunes de colas a través de cuatro métodos prácticos, de la manera siguiente:

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

Finalmente el objeto MQeQueueManagerConfigure se puede cerrar.

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

Creación de un gestor de colas sencillo en C:

Fase 1: crear los componentes de administración

Todas las acciones de administración local se realizan mediante MQeAdministrator. Esto permite crear nuevos gestores de colas y nuevas colas y realizar muchas otras acciones. Para todas las invocaciones, se exige un puntero hacia bloque de excepción, junto con un puntero para el descriptor de contenido del gestor de colas.

Fase 2: crear un gestor de colas

Para crear un gestor de colas, se exigen dos estructuras de parámetros. Una contiene los parámetros para el gestor de colas y la otro para el registro. En este caso sencillo, los valores predeterminados son adecuados, con la adición de la ubicación del registro y del almacén de colas.

La invocación al administrador creará el gestor de colas. Tenga en cuenta que el nombre del gestor de colas se pasa a la invocación. Se devuelve el descriptor de contexto del gestor de colas.

```
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);

}
```

Figura 4. Ejemplo de creación de un gestor de colas en C

Iniciar gestores de colas

Antes de utilizar los gestores de colas, se tienen que crear. En los pasos de creación se utiliza la clase QueueManagerConfigure Java o la API de administración en C para generar datos de gestor de colas permanentes en un registro. A continuación, el gestor de colas utiliza el registro cada vez que se inicia.

Inicio de gestores de colas en Java

Normalmente, la creación y el inicio de un gestor de colas puede requerir un gran conjunto de parámetros. Por lo tanto, los parámetros necesarios se suministran como una instancia de MQeFields, almacenando los valores como campos de tipo y nombre correctos.

Los parámetros se dividen en dos categorías, los parámetros del gestor de colas y los parámetros de registro. Cada una de estas categorías se representa con su propia instancia MQeFields y las dos se incluyen en una instancia MQeFields. En el siguiente ejemplo de Java se explica este concepto, pasando el nombre de los gestores de colas, "ExampleQM", y la ubicación de un registro "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);
```

Siempre que vea "initialize the parameters" (inicializar los parámetros) en fragmentos de código, prepare un conjunto de parámetros tal como se muestra en el ejemplo, incluidas las opciones apropiadas. Sólo son obligatorios un nombre de gestor de colas y una ubicación de registro.

Inicio de un gestor de colas sencillo en Java:

Para iniciar el gestor de colas más sencillo, sólo tiene que proporcionar el nombre del gestor de colas y la ubicación del registro al constructor del gestor de colas. Con ello se iniciará y se activará el gestor de colas y, cuando se devuelva el constructor, el gestor de colas se estará ejecutando.

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

Existen otros modos de iniciar un gestor de colas que permiten pasar más parámetros a fin de sacar partido de algunas funciones avanzadas.

Inicio de gestores de colas en C

La función mqeQueueManager_new carga un gestor de colas para un registro establecido. Si desea que se efectúe esta tarea, necesitará información que suministra una estructura de parámetros del gestor de colas y una estructura de parámetros de registro.

En el siguiente ejemplo se muestra cómo se pueden establecer los valores predeterminados de estas estructuras, proporcionando sólo los directorios del registro y del almacén de colas:

```
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);
```

Esto crea un gestor de colas y carga su información permanente del registro y crea colas. Sin embargo, debe iniciar el gestor de colas para:

- Crear mensajes
- Obtener y transferir mensajes
- Procesar mensajes de administración, utilizando la cola de administración

Nota: En C, las colas se activan al iniciar el gestor de colas.

Para iniciar el gestor de colas, utilice

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

Cuando el gestor de colas se haya iniciado, se podrán llevar a cabo operaciones de mensajería y cualquier cola que tenga mensajes se cargará.

Para detener el gestor de colas, utilice

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

Una vez se haya detenido, podrá reiniciar el gestor de colas según sea necesario.

Cuando la aplicación haya finalizado su proceso, deberá liberar el gestor de colas para que, a su vez, libere cualquier recurso que utilice, por ejemplo, memoria. En primer lugar, detenga el gestor de colas y, a continuación, utilice:

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

Inicio de un gestor de colas sencillo en C:

Este proceso consta de dos pasos:

1. Creación del elemento del gestor de colas.
2. Inicio del gestor de colas.

La creación del gestor de colas precisa dos conjuntos de parámetros, uno para el gestor de colas y otro para el registro. Ambos conjuntos de parámetros se inicializan. Es necesario que haya directorios para el *almacén de colas* y el registro.

Nota: Todas las llamadas necesitan un puntero a ExceptBlock y un puntero al manejador del gestor de colas.

```
 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);
}
```

Ahora se puede iniciar el gestor de colas y llevar a cabo operaciones de mensajería:

```
     /* Start the queue manager  */

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

Parámetros de los gestores de colas

Lista de los nombres de parámetros que se pueden pasar al gestor de colas y al registro.

A continuación se listan los nombres de parámetros que se pueden pasar al gestor de colas y al registro:

Parámetros del gestor de colas

MQeQueueManager.Name(ascii)
Éste es el nombre del gestor de colas que se está iniciando.

Parámetros de registro

MQeRegistry.LocalRegType(ascii)
Éste es el tipo de registro que se está abriendo. MQe da soporte actualmente a:

registro de archivos
Establezca este parámetro en `com.ibm.mqe.registry.MQeFileSession`.

registro privado
Establezca este parámetro en `com.ibm.mqe.registry.MQePrivateSession`.

Asimismo, se necesita un registro privado para algunas características de seguridad.

MQeRegistry.DirName(ascii)
Éste es el nombre del directorio que contiene los archivos de registro. Debe transferir este parámetro a un registro de archivos.

MQeRegistry.PIN(ascii)
Este PIN es necesario para un registro privado.

Nota: por razones de seguridad, MQe suprime el PIN y la KeyRingPassword, si se han suministrado, de los parámetros de inicio tan pronto como se activa el gestor de colas.

MQeRegistry.CAIPAddrPort(ascii)
Esta dirección y este número de puerto de un servidor de minicertificados son necesarios para efectuar un registro automático, de modo que el gestor de colas pueda obtener sus credenciales del servidor de minicertificados.

MQeRegistry.CertReqPIN(ascii)
Éste es el número de petición de certificado asignado por el administrador de minicertificados para permitir que el registro obtenga sus credenciales. Este número es necesario para llevar a cabo un registro automático, de modo que el gestor de colas pueda obtener sus credenciales del servidor de minicertificados.

MQeRegistry.Separator(ascii)
Se utiliza para especificar un separador que no sea predeterminado. Un separador es el carácter utilizado entre los componentes de un nombre de entrada, por ejemplo `<QueueManager><Separator><Queue>`. Aunque este parámetro esté especificado como una serie de caracteres, debe contener un solo carácter. Si contiene más de uno, se utilizará únicamente el

primero. Utilice el mismo separador para cada registro abierto y no lo modifique cuando ya se esté utilizando un registro. Si no se especifica este parámetro, el separador toma el valor predeterminado "+".

MQeRegistry.RegistryAdapter (ascii)
Ésta es la clase, o un alias que se resuelve en una clase, del adaptador que utiliza el registro para almacenar sus datos. Debe incluir esta clase si desea que el registro utilice un adaptador que no sea el MQeDiskFieldsAdapter predeterminado. Se puede utilizar cualquier clase de adaptador de almacenamiento válida.

Siempre se necesitan los dos primeros parámetros. Los dos últimos son para el registro automático del registro si se desea obtener sus credenciales del servidor de minicertificados.

MQeRegistry.RegistryAdapter (ascii)
La clase (o un alias que se resuelve en una clase) del adaptador que el registro utiliza para almacenar sus datos. Este valor debe especificarse si se desea que el registro utilice un adaptador que no sea el MQeDiskFieldsAdapter predeterminado. Puede emplearse cualquier clase de adaptador válida.

Un gestor de colas se puede ejecutar:
- Como cliente
- Como servidor
- En un servlet

En los apartados siguientes se describen el cliente, los servidores y el servlet de ejemplo que se proporcionan en el paquete examples.queuemanager.se hace referencia al código de ejemplo para ilustrar cómo iniciar los gestores de colas. Todos los gestores de colas se crean desde los mismos componentes base de MQe, con algunas adiciones que les otorgan sus propiedades exclusivas. MQe proporciona una clase de ejemplo, `MQeQueueManagerUtils`, que encapsula muchas de las funciones comunes.

En el inicio, todos los ejemplos necesitan parámetros. Estos parámetros se almacenan en archivos ini estándar. Se leen los archivos ini y los datos se convierten en un objeto MQeFields. El método `loadConfigFile()` de la clase MQeQueueManagerUtils realiza esta función.

Parámetros de registro de un gestor de colas

Descripción de los datos relacionados con el gestor de colas del registro

El registro es el almacén principal de la información relacionada con el gestor de colas. Hay uno para cada gestor de colas. Cada gestor de colas utiliza el registro para guardar:
- Datos de configuración del gestor de colas
- Definiciones de los recursos del escucha de comunicaciones
- Definiciones de colas
- Definiciones de colas remotas
- Definiciones de gestores de colas remotos
- Datos de usuario, incluida la información de seguridad que depende de la configuración
- Definiciones de recursos de puente opcionales

Tipo de registro

MQE_REGISTRY_LOCAL_REG_TYPE
El tipo de registro que se abrirá. Normalmente, se dará soporte al *registro de archivos* y al *registro privado*. Para algunas de las características de seguridad, se necesitará un registro privado.

Para un registro de archivos, este parámetro se debe establecer en:
`com.ibm.mqe.registry.MQeFileSession`

Para un registro privado, se debe establecer en:
`com.ibm.mqe.registry.MQePrivateSession`

Se pueden utilizar alias para representar estos valores.

Gestores de colas de cliente

Normalmente, un cliente se ejecuta en una plataforma de dispositivo y proporciona un gestor de colas que las aplicaciones del dispositivo pueden utilizar. Puede abrir muchas conexiones a otros gestores de colas.

A diferencia de un servidor, que normalmente se ejecuta durante largos períodos de tiempo, los clientes se inician y se detienen cuando lo solicitan las aplicaciones que los utilizan. Si varias aplicaciones van a compartir un cliente, las aplicaciones deben coordinar el inicio y detención del cliente.

Ejemplo: inicio de un gestor de colas de cliente:

El inicio de un gestor de colas de cliente implica:

1. Comprobar que no haya ningún cliente en ejecución. Sólo se permite un cliente por Java Virtual Machine).
2. Añadir los alias al sistema
3. Habilitar el rastreo si es necesario
4. Iniciar el gestor de colas

El siguiente fragmento de código inicia un gestor de colas de cliente:

```
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 );
}
```

Una vez haya iniciado el cliente, podrá obtener una referencia al objeto del gestor de colas mediante invocación a API mqeQueueManager_getReference(queueManagerName) desde la variable de clase estática `MQeClient.queueManager` o utilizando el método estático `MQeQueueManager.getReference(queueManagerName)`.

El siguiente fragmento de código carga los alias en el sistema:

```
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               */
    }
  }
}
```

Utilice el método `processAlias` para añadir cada alias al sistema. MQe y las aplicaciones pueden utilizar los alias una vez se hayan cargado los mismos.

Para iniciar un gestor de colas es necesario:

1. Crear una instancia de un gestor de colas. El nombre de la clase del gestor de colas que debe cargarse se especifica en el alias `QueueManager`. Utilice el cargador de clases de MQe para cargar la clase y llamar al constructor nulo.
2. Activar el gestor de colas. Utilice el método `activate`, pasando la representación del objeto MQeFields del archivo ini. El gestor de colas sólo utiliza las secciones `[QueueManager]` y `[Registry]` de los parámetros de arranque.

El siguiente fragmento de código inicia un gestor de colas:

```
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      */
}
```

Ejemplo - MQePrivateClient:

MQePrivateClient es una extensión de MQeClient que además configura el gestor de colas y el registro para permitir que haya colas seguras. Para un cliente seguro, se ha ampliado la sección `[Registry]` de los parámetros de inicio, del modo siguiente:

```
(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
```

Para que funcionen MQePrivateClient y MQePrivateServer, los parámetros de arranque *no* deben contener `CertReqPIN`, *`KeyRingPassword`* ni `CAIPAddrPort`.

Gestores de colas de servidor

Generalmente, un servidor se ejecuta en una plataforma de servidor. Un servidor puede ejecutar aplicaciones de lado servidor pero también puede ejecutar aplicaciones de lado cliente. Al igual que con los clientes, un servidor puede abrir conexiones con muchos otros gestores de colas tanto en los servidores como en los clientes. Una de las características principales que diferencia un servidor de un cliente es que puede manejar muchas peticiones de entrada al mismo tiempo. A menudo, un servidor actúa como un punto de entrada para muchos clientes en una red de MQe. MQe proporciona los siguientes ejemplos de servidor:

MQeServer
Servidor basado en una consola.

MQePrivateServer
Servidor basado en una consola con seguridad ampliada.

AwtMQeServer
Un componente frontal gráfico para MQeServer.

MQBridgeServer
Además de las funciones normales del servidor de MQe, este servidor puede enviar y recibir mensajes a y desde otros miembros de la familia MQ. Este servidor se encuentra en el paquete examples.mqbridge.queuemanager.

Ejemplo - MQeServer:

MQeServer es la implementación de servidor más sencilla.

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

Debe proporcionar el parámetro *-p* si el gestor de colas utiliza un registro privado. De lo contrario. el registro del gestor de colas se trata como un registro de archivos. El programa activa el gestor de colas (incluido un escucha activado en el puerto 8081) y entra un estado de latencia indefinido.

Utilice `Control-C` para concluir el servidor.

Para suprimir el gestor de colas creado, utilice la acción qm_delete de ejemplo.

Cuando dos gestores de colas se comunican entre sí, MQe abre una conexión entre los dos gestores de colas. La conexión es una entidad lógica que se utiliza como un conducto entre gestores de colas. Puede haber varias conexiones abiertas en cualquier momento.

Los gestores de colas de servidor, a diferencia de los gestores de colas de cliente, pueden disponer de uno o más escuchas. Un escucha espera a que se establezcan comunicaciones desde otros gestores de colas y procesa las peticiones de llegada, normalmente reenviándolas a su gestor de colas propietario. Un escucha tiene un adaptador especificado que define el protocolo de comunicaciones de entrada y también especifica cualquier dato adicional necesario.

Cree escuchas en el gestor de colas local con los mensajes de administración de forma remota y local. Sin embargo, un gestor de colas remoto debe tener un escucha para recibir un mensaje.

Un escucha que se acaba de crear enviando mensajes de administración al gestor de colas no se inicia, pues. Para iniciarlo puede enviar un mensaje de administración de forma explícita para iniciar el escucha o puede reiniciar el gestor de colas. No obstante, los escuchas son permanentes en el registro. Esto significa que, una vez creados, los escuchas que existen en el arranque del gestor de colas se inician de forma automática.

En este ejemplo se muestra cómo crear e iniciar un escucha con los mensajes de administración:

```
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();

.
.
```

Una vez iniciado el escucha, el servidor estará listo para aceptar las peticiones de red.

Cuando el servidor está desactivado:

1. El escucha se detiene, lo que impide la llegada de nuevas peticiones de entrada
2. Se cierra el gestor de colas

Ejemplo - MQePrivateServer:

MQePrivateServer es una ampliación de MQeServer que además puede configurar el gestor de colas y el registro para permitir que haya colas seguras.

Relación de entornos

En este tema se describen algunos requisitos para ejecutar las implementaciones en Java y C de MQe.

Java

Java

El gestor de colas java se ejecuta dentro de una instancia de una JVM. Sólo se puede tener un gestor de colas por JVM. Sin embargo, se pueden invocar múltiples instancias de JVM.

Cada uno de estos gestores de colas debe tener un nombre exclusivo. Las aplicaciones en Java se ejecutan en la misma JVM que el gestor de colas que utilizan.

Código C

Sólo se puede ejecutar un gestor de colas dentro de un proceso en C nativo. Se necesitan múltiples procesos para ejecutar múltiples gestores de colas. Cada uno de estos gestores de colas debe tener un nombre exclusivo.

Detención de los gestores de colas

Visión general sobre la detención de los gestores de colas en Java y C

Detención de un gestor de colas en Java

Existen dos maneras de cerrar un gestor de colas y las aplicaciones de MQe deben llamar a uno de los métodos de cierre cuando hayan acabado mediante el gestor de colas:

- closeQuiese
- closeImmediate

closeQuiesce:

Detención de un gestor de colas con el método closeQuiesce

Este método cierra un gestor de colas y especifica un intervalo para permitir que los procesos internos existentes finalicen con normalidad. Tenga en cuenta que este intervalo sólo se implementa como una serie de ciclos de pausas de 100 ms y reintentos. La invocación de este método impide que se inicie cualquier nueva actividad, como la transmisión de un mensaje, pero permite que las actividades que ya estén en proceso finalicen. El intervalo sólo es una sugerencia y varios factores de planificación de hebras

dependientes pueden hacer que el intervalo sea superior. Si las actividades que están en proceso actualmente acaban antes de tiempo, el método se devuelve antes de que se supere el período de detención.

Si la cola no se ha cerrado cuando finalice este período, se fuerza su cierre.

Después de haber invocado este método, no se enviarán más notificaciones de sucesos a los escuchas de mensajes. Puede ser que los mensajes lleguen después de que se haya invocado este método (y antes de que finalice). Estos mensajes no se notificarán. Los programadores de aplicaciones deben tener en cuenta esto y no presuponer que cada llegada de mensaje generará un suceso de mensaje.

```
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:

Detención de un gestor de colas con el método closeImmediate

Esto cierra el gestor de colas de forma inmediata.

Después de haber invocado este método, no se enviarán más notificaciones de sucesos a los escuchas de mensajes. Puede que los mensajes lleguen después de que se haya invocado esté método y antes de que finalice. Estos mensajes no se notifican y, por lo tanto, la llegada de mensajes no genera un suceso de mensaje.

```
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
```

Detención de un gestor de colas en C

Después de eliminar el mensaje de la cola, el gestor de colas se puede detener y liberar. Asimismo, puede liberar las series de caracteres que se han creado. Finalmente, termine la sesión:

```
      (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);
```

Supresión de los gestores de colas

En este apartado se ofrece información detallada sobre cómo suprimir un gestor de colas en Java y en C.

Java

Pasos necesarios para suprimir los gestores de colas en Java

Para suprimir un gestor de colas, siga los pasos que se indican a continuación:

1. Utilice la interfaz de administración para suprimir las definiciones
2. Crear y activar una instancia de `MQeQueueManagerConfigure`
3. Suprima la definición del gestor de colas y de la cola estándar
4. Cierre la instancia `MQeQueueManagerConfigure`

Una vez realizados estos pasos, se habrá suprimido el gestor de colas y ya no se podrá ejecutar más. Las definiciones de las colas se habrán suprimido, pero no las colas propiamente dichas. No se podrá acceder a ningún mensaje que haya quedado en las colas.

Nota: Los mensajes que quedan en las colas no se suprimen automáticamente. Los programas de aplicación deberían incluir un código que compruebe y gestione los mensajes restantes antes de suprimir el gestor de colas.

1. Supresión de las definiciones

Se puede utilizar MQeQueueManagerConfigure para suprimir las colas estándar que haya creado con MQeQueueManagerConfigure. Utilice la interfaz de administración para suprimir el resto de colas antes de invocar aMQeQueueManagerConfigure.

2. Creación y activación de una instancia de MQeQueueManagerConfigure

Este proceso es el mismo que se realiza cuando se crea un gestor de colas. Consulte el apartado “Creación de gestores de colas” en la página 19.

3. Supresión de la cola estándar y las definiciones de gestor de colas

Suprima las colas predeterminadas llamando a:

- mqeQueueManagerConfigure_deleteAdminQueueDefinition() para suprimir la cola de administración
- mqeQueueManagerConfigure_deleteAdminReplyQueueDefinition() para suprimir la cola de respuestas de administración
- mqeQueueManagerConfigure_deleteDeadLetterQueueDefinition() para suprimir la cola de mensajes no entregados
- mqeQueueManagerConfigure_deleteSystemQueueDefinition() para suprimir la cola local predeterminada

Estos métodos funcionarán correctamente aunque las colas no existan.

Suprima la definición del gestor de colas llamando a 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)
{ ... }
```

Puede suprimir las definiciones de la cola predeterminada y el gestor de colas de forma conjunta llamando a mqeQueueManagerConfigure_deleteStandardQMDefinitions(). Este método se proporciona para su comodidad y equivale a:

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

4. Cierre de la instancia de MQeQueueManagerConfigure

Cuando haya suprimido la cola y las definiciones del gestor de colas, podrá cerrar la instancia MQeQueueManagerConfigure.

El ejemplo completo es similar al siguiente:

```
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

Pasos obligatorios para suprimir los gestores de colas en C

Los pasos para suprimir un gestor de colas son:

1. Eliminar todas las definiciones de conexión.
2. Eliminar todas las colas, incluidas las colas del "sistema", por ejemplo, la cola de mensajes no entregados. Asegúrese de que todas las colas estén vacías.
3. Elimine el gestor de colas.

Necesita un administrador para realizar estas funciones. También recomendamos detener el gestor de colas en primer lugar.

Nota: si se suprime el gestor de colas, se liberará de forma automática el descriptor de contenido del gestor de colas.

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);
}
```

Ciclo de vida de mensajería

Descripción de la serie de estados a través de los cuales avanza un mensaje cuando se coloca en una cola

Cuando un mensaje se transfiere a una cola, progresa pasando por una serie de estados. En este apartado se describen estos estados y los sucesos o mandatos relacionados en los siguientes apartados:

Estados de los mensajes

La mayoría de los tipos de colas contienen mensajes en un almacén permanente, por ejemplo, un disco duro. Mientras se encuentra en el almacén, el estado del mensaje cambia a medida que se transfiere dentro y fuera del almacén, tal como se muestra en la Figura 5 en la página 39:

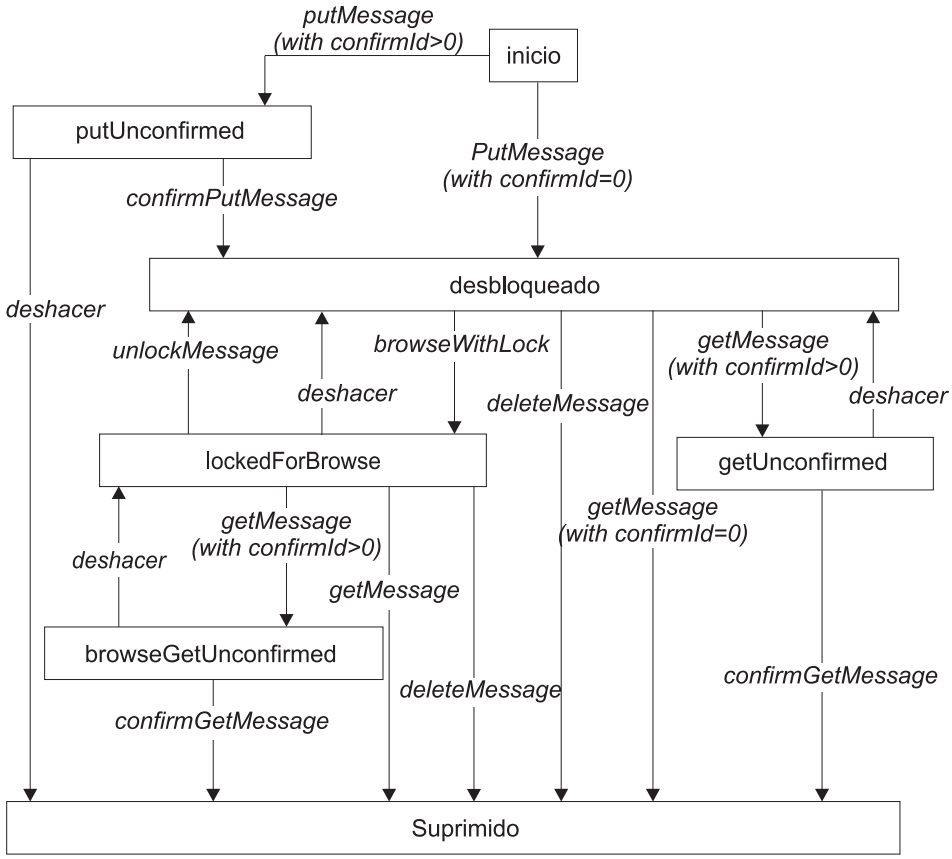

Figura 5. Flujo de estado del mensaje almacenado

En este diagrama, ″start″ y ″deleted″ no son estados de mensaje reales. Se trata de los puntos de entrada y salida del modelo de estado. Los estados del mensaje son:

Put unConfirmed
Un mensaje se coloca en el almacén de mensajes de una cola con un ID de confirmación (`confirmID`). El mensaje está realmente oculto a todas las acciones, excepto `confirmPutMessage` o `undo`.

Unlocked
Un mensaje se ha colocado en una cola y está disponible para todas las operaciones.

Locked for Browse
Un examen con bloqueo recupera mensajes. Los mensajes se encuentran ocultos a todas las consultas, excepto `getMessage`, `unlock`, `delete`, `undo` y `unlockMessage`. Una operación de examen devuelve un `lockID` (ID de bloqueo). Se debe suministrar este ID de bloqueo (`lockID`) a todas las demás operaciones.

Get Unconfirmed
Se ha producido una llamada a `getMessage` con un ID de confirmación (`confirmID`), pero la operación get no se ha confirmado. El mensaje es invisible para todas las consultas excepto para `confirmGetMessage`, `confirm` y `undo`. Cada una de estas acciones requiere que se incluya el ID de confirmación (`confirmID`) correspondiente para confirmar la obtención.

Browse Get Unconfirmed
Se ha obtenido un mensaje mientras su examen estaba bloqueado. Esto sólo puede suceder si se pasa el ID de bloqueo (`lockID`) correcto a la función `getMessage`.

En una cola remota asíncrona, existen otros estados en los que un mensaje se transmite a otra máquina. Estos estados se especifican como ″unlocked″ (desbloqueados), es decir, sólo se transmiten mensajes confirmados.

Sucesos de mensajes

Los mensajes pasan de un estado a otro como resultado de un suceso. Una llamada API genera habitualmente estos sucesos. Los posibles sucesos de mensajes, tal como se muestra en la Figura 5 en la página 39, son:

putMessage
Coloca un mensaje en una cola. No es necesario un ID de confirmación (`confirmID`).

getMessage
Recupera un mensaje de una cola. No es necesario un ID de confirmación (`confirmID`).

putMessage with confirmId>0
Coloca un mensaje en una cola. Es necesario un ID de confirmación>0 (`confirmID`). Sin embargo, los mensajes no llegan al extremo de recepción en orden de envío, sino en orden de confirmación.

confirmPutMessage
Confirmación para una operación `putMessage` anterior con un ID de confirmación (`confirmID`) >0.

getMessage with confirmId>0
Recupera un mensaje de una cola. Es necesario un ID de confirmación>0 (`confirmID`).

confirmGetMessage
Confirmación para una operación `getMessage` anterior con `confirmID`>0.

browseWithLock
Examina mensajes y bloquea aquéllos que coinciden. Evita que los mensajes cambien mientras se realiza el examen.

unlockMessage
Desbloquea un mensaje bloqueado con un mandato `browsewithLock`.

undo Desbloquea un mensaje bloqueado con un examen, deshace una operación `getMessage` con un ID de confirmación>0 (`confirmID`>0) o deshace una operación putMessage con un ID de confirmación>0 (`confirmID`>0).

deleteMessage
Elimina un mensaje de una cola.

Campos de índice de mensajes

Debido a las restricciones de tamaño de la memoria, los mensajes completos no se mantienen en la memoria, sino que, para permitir una búsqueda de mensajes más rápida, MQe contiene campos específicos de cada mensaje en un *índice de mensajes*. Los campos contenidos en el índice son:

Java En Java, el índice contiene los campos siguientes:

UniqueID
`MQe.Msg_OriginQMgr + MQe.Msg_Time`

MessageID
`MQe.Msg_ID`

CorrelationID
`MQe.Msg_CorrelID`

Prioridad
`MQe.Msg_Priority`

C En C, los siguientes campos se contienen en el índice:

UniqueID
MQE_MSG_ORIGIN_QMGR + MQE_MSG_TIME

MessageID
MQE_MSG_MSGID

CorrelationID
MQE_MSG_CORRELID

Prioridad
MQE_MSG_PRIORITY

Con estos campos en un filtro, la búsqueda es más eficaz, ya que puede que MQe no tenga que cargar todos los mensajes disponibles en la memoria.

Operaciones de mensajería

En la siguiente tabla se muestra cuáles son los tipos de operaciones de mensajería válidos en colas locales, colas remotas síncronas y colas remotas asíncronas. Tenga en cuenta que las operaciones `Listen` y `Wait` sólo reciben soporte en Java.

Tabla 3. Operaciones de mensajería en colas de MQe

Operación	Cola local	Cola remota síncrona	Cola remota asíncrona
"Transferencia"	Sí	Sí	Sí
"Get" en la página 42	Sí	Sí	No
"Browse" en la página 42	Sí	Sí	No
"confirmPut" en la página 43	Sí	Sí	Sí
"confirmGet" en la página 43	Sí	Sí	No
"Delete" en la página 43	Sí	Sí	No
"Listen" en la página 44	Sí	No	No
"Wait" en la página 44	Sí	Sí	No

Nota:
1. La operación de espera remota síncrona se implementa mediante un sondeo de la cola remota, de modo que el tiempo de espera real es un múltiplo del tiempo de sondeo
2. El puente de MQe que se suministra con MQe sólo da soporte a una operación de transferencia asegurada o no asegurada, de obtención no asegurada y de examen no asegurado (sin bloqueo).

Transferencia

Esta operación coloca los mensajes especificados en una cola especificada. La cola puede pertenecer a un gestor de colas remoto o local. Las transferencias a colas remotas pueden producirse inmediata o posteriormente, en función de cómo se defina la cola remota en el gestor de colas local.

Si una cola remota está definida como síncrona, la transmisión del mensaje se realizará de forma inmediata. Si una cola remota está definida como asíncrona, el mensaje se almacenará dentro del gestor de colas local. El mensaje permanecerá allí hasta que se transmita. La llamada del mensaje de transferencia puede finalizar antes de que el mensaje se transfiera. Consulte el apartado "Entrega de mensajes" en la página 51 para obtener más información.

Nota: En Java, si el gestor de colas local no contiene una definición de la cola remota, intenta ponerse en contacto con la cola de manera sincrónica.Esto no es aplicable a la base de código en C.

La entrega asegurada depende del valor del parámetro ID de confirmación (confirmID). Si se pasa un valor distinto a cero, el mensaje se transmite normalmente, pero se bloquea en la cola de destino hasta que se recibe una posterior confirmación. Si se pasa un valor de cero, se transmite el mensaje sin necesidad de una confirmación posterior. No obstante, la entrega del mensaje no está asegurada. Consulte el apartado “Entrega de mensajes” en la página 51, para obtener más información sobre la entrega asegurada y no asegurada de mensajes.

Mediante la seguridad de nivel de mensajes puede proteger un mensaje.

Get

Esta operación devuelve un mensaje disponible de una cola determinada y lo elimina de la cola. La cola puede pertenecer a un gestor de colas local o remoto de MQe, pero no puede ser una cola remota asíncrona.

Si no especifica ningún filtro, se devolverá el primer mensaje disponible. Si especifica un filtro, se devolverá el primer mensaje disponible que coincida con el filtro. Si se incluye un ID de bloqueo (lockID) válido en el filtro del mensaje, permitirá obtener mensajes que se hayan bloqueado mediante una operación de examen previa. Si no hay ningún mensaje disponible, la operación de obtención devolverá un error.

La utilización de la entrega asegurada de mensajes depende del valor del parámetro ID de confirmación (confirmID). Si se pasa un valor distinto a cero, se devolverá el mensaje normalmente. Sin embargo, el mensaje se bloqueará y no se eliminará de la cola de destino hasta que reciba una confirmación posterior. Se puede emitir una confirmación utilizando el método confirmGetMessage(). No obstante, la entrega del mensaje no está asegurada. Consulte el apartado “Entrega de mensajes” en la página 51, para obtener más información sobre la entrega asegurada y no asegurada de mensajes.

Browse

Se pueden examinar colas para mensajes mediante la utilización de un filtro, por ejemplo ID del mensaje (message ID) o prioridad (priority). El examen permite recuperar todos los mensajes que coinciden con el filtro, pero los deja en la cola. La cola puede pertenecer a un gestor de colas local o remoto.

MQe también da soporte al *examen bajo bloqueo*. Esto permite bloquear los mensajes coincidentes en la cola. Se pueden bloquear mensajes individualmente o en grupos identificados a través de un filtro y la operación de bloqueo devuelve un ID de bloqueo (lockID). Utilice el ID de bloqueo (lockID) para obtener o suprimir mensajes. Una opción de browse permite devolver los mensajes completos o sólo los ID exclusivos

```
   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);
}
```

La devolución de una recopilación completa de mensajes puede consumir muchos recursos del sistema. Al establecer el parámetro justUID en true se devuelve el uniqueID (ID exclusivo) de cada mensaje que coincide con el filtro.

Los mensajes devueltos en la recopilación siguen siendo visibles para otras API de MQe. Por lo tanto, cuando se realizan operaciones posteriores en los mensajes que contiene la enumeración, la aplicación debe saber que otra aplicación puede procesar esos mensajes una vez se haya devuelto la recopilación. Para evitar que otras aplicaciones procesen los mensajes, utilice el método `browseMessagesAndLock` para bloquear mensajes contenidos en la enumeración.

Delete

Este método suprime un mensaje de una cola. No devuelve el mensaje a la aplicación que lo ha llamado. Se debe especificar el ID exclusivo (`UniqueID`) y sólo se podrá suprimir un mensaje por cada operación.

La cola puede pertenecer a un gestor de colas de MQe remoto síncrono o local. Si se incluye un ID de bloqueo (`lockID`) válido en el filtro del mensaje, se podrán suprimir mensajes que se hayan bloqueado mediante una operación previa, por ejemplo, una operación de examen. Si un mensaje no está disponible, la aplicación devolverá un 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

Este método realiza la confirmación de una operación `putMessage()` anteriormente satisfactoria.

confirmGet

Este método confirma la recepción satisfactoria de un mensaje recuperado de un gestor de colas mediante una operación `getMessage()` anterior. El mensaje permanece bloqueado en la cola de destino hasta que recibe un flujo de comunicaciones de confirmación.

Listen

Las aplicaciones pueden permanecer nuevamente a la escucha de sucesos de mensajes de MQe con un filtro opcional. Sin embargo, para hacer eso, se debe añadir un escucha a un gestor de colas. Los escuchas reciben notificaciones cuando los mensajes llegan a una cola.

Wait

Este método implementa el sondeo de mensajes. Permite especificar un período de tiempo para que los mensajes lleguen a una cola. Java implementa una función de ayuda para esta tarea. Puesto que la base de código en C no tiene hebras, debe implementar una función en el código de la capa de aplicación. En el ejemplo siguiente se muestra el método `Wait`:

Java El sondeo de mensajes utiliza el método `waitForMessage()`. Este mandato emite un mandato `getMessage()` a la cola remota, en intervalos regulares. En cuanto un mensaje que coincide con el filtro suministrado pasa a estar disponible, se devuelve a la aplicación que realiza la llamada:

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

El método `waitForMessage()` sondea la cola remota durante el tiempo especificado en el parámetro final. El tiempo se especifica en milisegundos. Por lo tanto, en el ejemplo, el sondeo dura 6 segundos. Esto bloquea la hebra en la que se ejecuta el mandato durante 6 segundos, a no ser que se devuelva antes un mensaje. El sondeo de mensajes funciona tanto en las colas locales como en las colas remotas.

Nota: Si se utiliza esta técnica, se envían múltiples peticiones a través de la red.

Clasificación en las colas

Visión general de la clasificación de mensajes en una cola

El orden de los mensajes de una cola depende principalmente de su prioridad. La prioridad de los mensajes es un valor de 9 (el más alto) a 0 (el más bajo). Los mensajes con el mismo valor de prioridad se clasifican por su hora de llegada a la cola, y los mensajes que han estado durante más tiempo en la cola están al principio del grupo de prioridad.

Lectura de mensajes de una cola

Si emite un mandato `getMessage` cuando una cola está vacía, la cola genera una excepción de la base de código Java `Except_Q_NoMatchingMsg` o devuelve`MQERETURN_QUEUE_ERROR, MQEREASON_NO_MATCHING_MSG` de la base de código en C. Esto permite crear una aplicación para leer todos los mensajes que hay disponibles en una cola.

Java

Si se encierra la llamada a `getMessage()` dentro de un bloque `try..catch`, se puede comprobar el código de la excepción obtenida. Esto se lleva a cabo mediante el método `code()` de la clase MQeException. Se puede comparar el resultado del método `code()` con una lista de constantes de excepciones publicadas por la clase de MQe. Si la excepción no es del tipo `Except_Q_NoMatchingMsg`, vuelva a generar la excepción.

El código siguiente muestra esta técnica:

```
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;
}
```

Por lo tanto, se pueden leer todos los mensajes desde una cola obteniendo los mensajes de forma iterativa hasta que se devuelva `MQe.Except_Q_NoMatchingMsg`.

C

Se pueden leer todos los mensajes de una cola mediante una repetición en bucle hasta que el código de retorno sea `MQERETURN_QUEUE_WARNING` y el código de razón sea `MQEREASON_NO_MATCHING_MSG`.

Browse y Lock

Si se efectúa `BrowseAndLock` en un grupo de mensajes, se permite a una aplicación asegurar que no hay ninguna otra aplicación que pueda procesar mensajes cuando estén bloqueados. Los mensajes permanecen bloqueados hasta que la aplicación los desbloquea. Ninguna otra aplicación puede desbloquear los mensajes. Ningún mensaje que llegue a la cola después de que se haya realizado la operación `BrowseAndLock` estará bloqueado.

Una aplicación puede llevar a cabo una operación de obtención (`get`) o de supresión (`delete`) en los mensajes para eliminarlos de la cola. Para ello, la aplicación debe suministrar el ID de bloqueo (`lockID`) que se devuelve con la enumeración de mensajes.

El hecho de especificar el ID de bloqueo (`lockID`) permite a las aplicaciones trabajar con mensajes bloqueados sin tener que desbloquearlos primero.

En lugar de suprimir los mensajes de la cola, también se pueden desbloquear. Esta acción hace que vuelvan a ser visibles para todas las aplicaciones de MQe. Para ello, utilice el método `unlockMessage`.

Nota: Consulte la publicación MQe Configuration Guide para conocer las consideraciones especiales sobre las colas puente de MQ.

Ejemplo - Java:

Ejemplo de examen y bloqueo (Java)

El objeto MQeMessageEnumerationMQeEnumeration contiene todos los mensajes que coinciden con el filtro proporcionado para el examen. MQeEnumeration se puede utilizar del mismo modo que la enumeración Java estándar. Se pueden enumerar todos los mensajes examinados del modo siguiente:

Nota: Se debe proporcionar un ID de confirmación (confirmID), en caso de que la acción de localización de los mensajes no se ejecute correctamente. Se puede deshacer la acción de localización, para lo cual es necesario el ID de confirmación (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 ) );
}
```

El código siguiente lleva a cabo una operación de supresión (`delete`) en todos los mensajes que devuelve la enumeración. El ID exclusivo (`UniqueID`) y el ID de bloqueo (`lockID`) del mensaje se utilizan como filtro en la operación `delete`:

```
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);
}
```

Ejemplo: C:

Ejemplo de examen y bloqueo (C)

En el ejemplo de la base de código en C se obtiene el mensaje que se muestra a continuación. Tenga en cuenta los parámetros adicionales, un ID de confirmación (`confirmID`) en caso de que la operación se tenga que deshacer y el ID de bloqueo (`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);
}
```

Escuchas de mensajes

Nota: Este apartado no se aplica a la base de código en C.

MQe permite a una aplicación *escuchar* los sucesos que tienen lugar en las colas. La aplicación puede especificar filtros de mensajes para identificar los mensajes en los que está interesada, tal como se muestra en el siguiente ejemplo en Java:

```
/* 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 );
```

Los siguientes parámetros se pasan al método addMessageListener():

- El nombre de la cola en la que se está a la escucha de las operaciones de los mensajes
- Un objeto de *devolución de llamada* que implementa MQeMessageListenerInterface
- Un objeto MQeFields que contiene un filtro de mensaje

Cuando llega un mensaje a una cola que tiene asociado un escucha, el gestor de colas llama al objeto de `devolución de llamada` que se ha especificado cuando se ha creado el escucha de mensajes.

A continuación, se facilita un ejemplo de cómo una aplicación maneja normalmente los sucesos de los mensajes en 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() es un método implementado en MQeMessageListenerInterface. El parámetro `msgEvent` contiene información sobre el mensaje, incluidos:

- El nombre de la cola a la que ha llegado el mensaje
- El `UID` del mensaje
- El ID de mensaje (`messageID`)
- El ID de correlación (`correlationID`)
- La prioridad (`priority`) del mensaje

Los filtros de mensajes sólo funcionan en colas locales. Una técnica diferente conocida como sondeo, permite obtener los mensajes en cuanto llegan a las colas remotas.

Sondeo de mensajes

Nota: Este apartado no se aplica a la base de código en C.

El sondeo de mensajes utiliza el método mqeQueueManager_waitForMessage(). Este mandato emite un mandato mqeQueueManager_getMessage() a la cola remota, en intervalos regulares. En cuanto un mensaje que coincide con el filtro suministrado pasa a estar disponible, se devuelve a la aplicación que realiza la llamada.

Normalmente, una espera a una llamada de mensaje es similar a la siguiente:

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

El método mqeQueueManager_waitForMessage() sondea la cola remota durante el tiempo especificado en el parámetro final. El tiempo se especifica en milisegundos, de modo que en el ejemplo anterior el sondeo dura 60 segundos. Durante este período de tiempo, la hebra en la que se ejecuta el mandato está bloqueada, a menos que se devuelva un mensaje antes de que transcurra este período de tiempo.

El sondeo de mensajes funciona tanto en las colas locales como en las colas remotas.

Nota: Al utilizar esta técnica se envían varias peticiones a través de la red.

Desencadenamiento de transmisión

Este método intenta transmitir mensajes pendientes. Sólo se transmiten los mensajes desbloqueados.

Las colas remotas asíncronas y las colas del servidor local responden al proceso de desencadenamiento de transmisión. Transfiera mensajes sin ID de confirmación (`confirmID`) o transfiéralos y confírmelos antes de llamar a este método. Sólo los mensajes que se hayan transferido (con 'put') completamente se podrán transmitir.

Normas de desencadenamiento de transmisión

Existe un número de normas, que pueden controlar el proceso desencadenante de transmisión, si se lleva a cabo el proceso. Consulte el tema sobre normas para obtener más información.

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

Servlet

Visión general de los gestores de colas de servlets que se ejecutan dentro de un servidor web

Además de ejecutarse como un servidor autónomo, un gestor de colas puede encapsularse en un servlet para ejecutarse en un servidor web. Un gestor de colas de servlet tiene casi todas las mismas capacidades que un gestor de colas de servidor. MQeServlet proporciona una implementación de ejemplo de un servlet. Al igual que con el servidor, los servlets utilizan los archivos ini para contener los parámetros de inicio. Un servlet utiliza muchos de los mismos componentes de MQe que el servidor.

El principal componente no necesario en un servlet es el escucha de conexión, cuya función la maneja el propio servidor web. Los servidores web sólo manejan corrientes de datos http de modo que cualquier cliente de MQe que desee comunicarse con un servlet de MQe debe utilizar el adaptador http (com.ibm.mqe.adapters.MQeTcpipHttpAdaper). Cuando se configuran conexiones para los gestores de colas que se ejecutan en los servlets, se deberá especificar el nombre del servlet en el campo de parámetros de la conexión.

Ejemplo: configuración de una conexión en un servlet

Las siguientes definiciones configuran una conexión en el servlet /servlet/MQe con el gestor de colas `PayrollQM`:

Nombre de la conexión
`PayrollQM`

Canal com.ibm.mqe.communications.MQeChannel

Nota: La clase com.ibm.mqe.MQeChannel se ha trasladado y ahora se conoce como com.ibm.mqe.communications.MQeChannel. Cualquier referencia que se haga al nombre anterior de la clase en los mensajes de administración se sustituirá automáticamente por el nuevo nombre de clase.

Adaptador de canal
com.ibm.mqe.adapters.MQeTcpipAdapter:192.168.0.10:80

Parámetros
/servlet/MQe

Opciones

Ejemplo: configuración de una conexión en un servlet mediante alias

Si se han configurado los alias correspondientes, puede configurar la conexión de la manera siguiente:

Nombre de la conexión
PayrollQM

Canal DefaultChannel

Adaptador
Network:192.168.0.10:80

Parámetros
/servlet/MQe

Opciones

Diferencias entre el inicio de un servidor y el de un servlet

Las principales diferencias con el inicio de un servidor son:

- El servlet altera temporalmente el método init de la superclase. El servidor web llama a este método para iniciar el servlet. Generalmente, esto sucede cuando llega la primera petición para el servlet.
- El nombre del archivo ini de parámetros de inicio no se puede pasar desde la línea de mandatos. El ejemplo espera obtener el nombre utilizando el getInitParameter() del servlet, que toma el nombre de un parámetro y devuelve un valor. El servlet de MQe utiliza un parámetro *Startup* que espera que contenga un nombre de archivo ini. El mecanismo para configurar los parámetros en un servidor web depende de éste.
- Un escucha no se inicia ya que el servidor web maneja todas las peticiones de red en nombre del servlet.
- Puesto que no hay ningún escucha, es necesario un mecanismo para cerrar conexiones que han estado inactivas durante más tiempo del establecido para el tiempo de espera. Para realizar esta función, se crea una instancia de la clase de temporizador sencillo MQeChannelTimer. El valor de *TimeInterval* es el único parámetro que se utiliza de la sección [Listener] del archivo ini.

Ejemplo: inicio de un servlet

El servlet de MQe amplía el nombre de servlet en C javax.servlet.http.HttpServlet y altera temporalmente los métodos para iniciar, detener y manejar nuevas peticiones. El siguiente fragmento de código inicia un 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() );
  }
}
```

Ejemplo: manejo de peticiones de entrada

Un servlet espera que el servidor web acepte y maneje las peticiones de entrada. Cuando el servidor web haya decidido que la petición es para un servlet de MQe, pasará la petición a MQe utilizando el método doPost(). El código siguiente maneja esta petición:

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 );
  }
}
```

Este método:

1. Lee la corriente de datos de entrada http como una *matriz de bytes*. La corriente de datos de entrada se puede almacenar en el almacenamiento intermedio, de forma que el método read() se utilice para asegurarse de que se ha leído la corriente de datos completa antes de continuar.

   Nota: MQe sólo maneja peticiones con el método doPost(), no acepta peticiones con el método doGet().
2. La petición se pasa a MQe a través de un gestor de conexiones. A partir de este punto, todo el proceso de la petición lo manejan las clases núcleo de MQe, como el gestor de colas.
3. Una vez que MQe ha terminado de procesar la petición, devuelve el resultado incluido en las cabeceras http como matriz de bytes. Dicha matriz vuelve a pasar al servidor web y se vuelve a transmitir al cliente que ha creado la petición.

Ejecución de varios servlets en un servidor web

Los servidores web pueden ejecutar varios servlets. Es posible ejecutar varios servlets de MQe en un servidor web, con las limitaciones siguientes:

- Cada servlet debe tener un nombre exclusivo
- Sólo se permite un gestor de colas por servlet
- Cada servlet de MQe debe ejecutarse en una JVM (Java Virtual Machine) diferente.

Entrega de mensajes

Detalles de los distintos tipos de procesos de entrega de mensajes

Las redes de MQe están compuestas por gestores de colas conectados y pueden incluir pasarelas. Pueden abarcar muchas redes físicas y direccionar mensajes entre ellas. En general, proporcionan acceso síncrono y asíncrono a las colas con un modelo de programación independiente de la ubicación de las colas.

Entrega de mensajes asíncrona

Una transferencia asíncrona a una cola remota coloca el mensaje en el almacén de reserva asociado a la definición local de esa cola, junto con su nombre del gestor de colas de destino, nombre de cola y características de compresor, autentificador y cifrador que coinciden con el destino final del mensaje. Se llama al método de vuelco del mensaje, puesto que se guarda en un almacén permanente en un formato seguro definido por su cola de destino. El gestor de colas controla la entrega de mensajes. Identifica o establece una conexión con características apropiadas al gestor de colas para el siguiente punto de conexión y, a continuación, crea o vuelve a utilizar un transportador para el gestor de colas de destino. El transportador vuelca el mensaje y transmite la serie de bytes obtenida. El gestor de colas de destino y el nombre de cola no forman parte de ese flujo de mensajes.

Si es conveniente, el mensaje se cifra y se comprime en la conexión. Si ha llegado a su gestor de colas de destino, se descifra y se descomprime. Se crea un nuevo mensaje utilizando el método de restauración y el mensaje que se obtiene se coloca en la cola de destino. Si el mensaje no ha llegado a su gestor de colas de destino, se descifra y se descomprime. A continuación, se vuelve a cifrar y comprimir y se coloca en una cola de almacenamiento y reenvío con el fin de efectuar una transmisión progresiva, si existe una cola de almacenamiento y reenvío. En ambos casos se retiene en la cola correspondiente con un formato seguro, tal y como define su cola de destino.

Una característica de la entrega de mensajes asíncrona es que los mensajes se pasan al gestor de colas en puntos de conexión intermedios y se colocan en las colas para realizar una transmisión progresiva. Los mensajes se sacan de las colas intermedias primero por orden de prioridad y, a continuación, por orden de llegada a la cola. Los mensajes duplicados, creados cuando se reenvía un mensaje, también se sacan de las colas intermedias por orden de llegada a la cola.

Entrega de mensajes síncrona

La entrega de mensajes síncrona es parecida a la entrega asíncrona descrita anteriormente, pero el nivel de participación del gestor de colas en los puntos de conexión intermedios es mucho más bajo, afectando así al transportador y a las conexiones. Para identificar el siguiente enlace, se establece una conexión de extremo a extremo, utilizando los adaptadores definidos en las especificaciones del protocolo en cada nodo intermedio. Al final del último enlace, donde no hay más descriptores de archivos relevantes, el mensaje se pasa a las capas superiores del gestor de colas para iniciar su proceso. Por lo tanto, el nodo emisor no coloca el mensaje en la cola, sino que lo pasa por la conexión, a través de puntos de conexión intermedios y, a continuación, se lo da al gestor de colas de destino para colocarlo en la cola de destino.

El enlace con MQ utiliza una cola puente en la pasarela, que transforma el mensaje en un formato de MQ. Este mecanismo significa que la mensajería síncrona del estilo de MQe de un dispositivo es posible para MQ, con la terminación de la conexión en la pasarela. El mensaje se entrega en tiempo real desde la pasarela, a través de un canal de cliente, hasta un servidor de MQ. A partir de ese punto, puede ser necesario direccionar de forma asíncrona su destino junto con los canales de mensajes de MQ.

De forma similar, un dispositivo únicamente capaz de enviar mensajes síncronos puede enviar mensajes a una cola asíncrona de MQe, siempre que haya disponible un intermediario apropiado.

Entrega asegurada y no asegurada de mensajes

La entrega de mensajes mediante la transmisión de mensajes síncrona puede estar asegurada o no asegurada.

Entrega asegurada de mensajes

La transmisión asíncrona introduce el concepto de *entrega asegurada de mensajes*. Cuando se entregan mensajes de forma asíncrona, MQe entrega cada mensaje en una única vez a su cola de destino. Sin

embargo, sólo puede asegurarlo de forma válida si la definición de la cola remota y del gestor de colas remoto coinciden con las características actuales de la cola remota y el gestor de colas remoto. Si una definición de cola remota y la cola remota no coinciden, entonces es posible que el mensaje no se pueda entregar. En este caso, el mensaje no se pierde, pero se queda almacenado en el gestor de cola local.

Entrega no asegurada de mensajes

La entrega no asegurada de un mensaje se lleva a cabo en un solo flujo de red. El gestor de colas que envía el mensaje crea o reutiliza un canal con el gestor de colas de destino.

Se realiza un vuelco del mensaje que se ha de enviar para crear una corriente de bytes, que se entrega al canal para su transmisión. Una vez el canal devuelve el control del programa, el gestor de colas emisor sabe que el mensaje se ha entregado correctamente al gestor de colas de destino determinado, que el destino ha anotado el mensaje en una cola y que las aplicaciones de MQe pueden visualizar el mensaje.

Sin embargo, se puede producir un problema si, a través del canal, el emisor recibe una excepción procedente del destino. El emisor no tiene modo alguno de saber si la excepción se ha producido antes o después de que el mensaje se haya anotado y se haya podido visualizar. Si la excepción se ha producido antes de que el mensaje se haya podido visualizar, no hay peligro en que el emisor vuelva a enviar el mensaje. Sin embargo, si la excepción se ha producido después de que el mensaje se haya podido visualizar, hay peligro de que el emisor vuelva a enviar el mensaje al sistema puesto que una aplicación de MQe puede haber procesado el mensaje antes de que el emisor vuelva a enviarlo.

La solución a este problema requiere transmitir un flujo de confirmación adicional. Si la aplicación del emisor recibe una respuesta satisfactoria a este flujo, entonces sabrá que el mensaje se ha entregado una sola vez.

Entrega de mensajes asegurada síncrona

Se puede realizar una entrega asegurada de mensajes utilizando la transmisión síncrona de mensajes.

Transferencia (put) de mensaje: transferencia (put) asegurada

Se puede realizar una entrega asegurada de mensajes utilizando la transmisión síncrona de mensajes, sin embargo, la aplicación será la responsable del manejo de los errores.

El parámetro `confirmID` del método `putMessage` dicta si se espera o no un flujo de confirmación. Un valor cero (`zero`) significa que la transmisión del mensaje se produce en un flujo de comunicaciones, mientras que un valor mayor que cero significa que se espera un flujo de confirmación. El gestor de colas de destino anota el mensaje en la cola de destino de forma habitual, pero el mensaje queda bloqueado e invisible para las aplicaciones de MQe hasta que se recibe un flujo de confirmación. Cuando transfiera mensajes con el ID de confirmación (`confirmID`), los mensajes se ordenarán por tiempo de confirmación y no por tiempo de llegada.

Una aplicación de MQe puede emitir una confirmación de mensaje `put` mediante el método mqeQueueManager_`confirmPutMessage`. Una vez el gestor de colas de destino recibe un flujo generado por este mandato, desbloquea el mensaje y permite que se visualice en las aplicaciones de MQe. Sólo se puede confirmar un mensaje cada vez. No se puede confirmar un lote de mensajes.

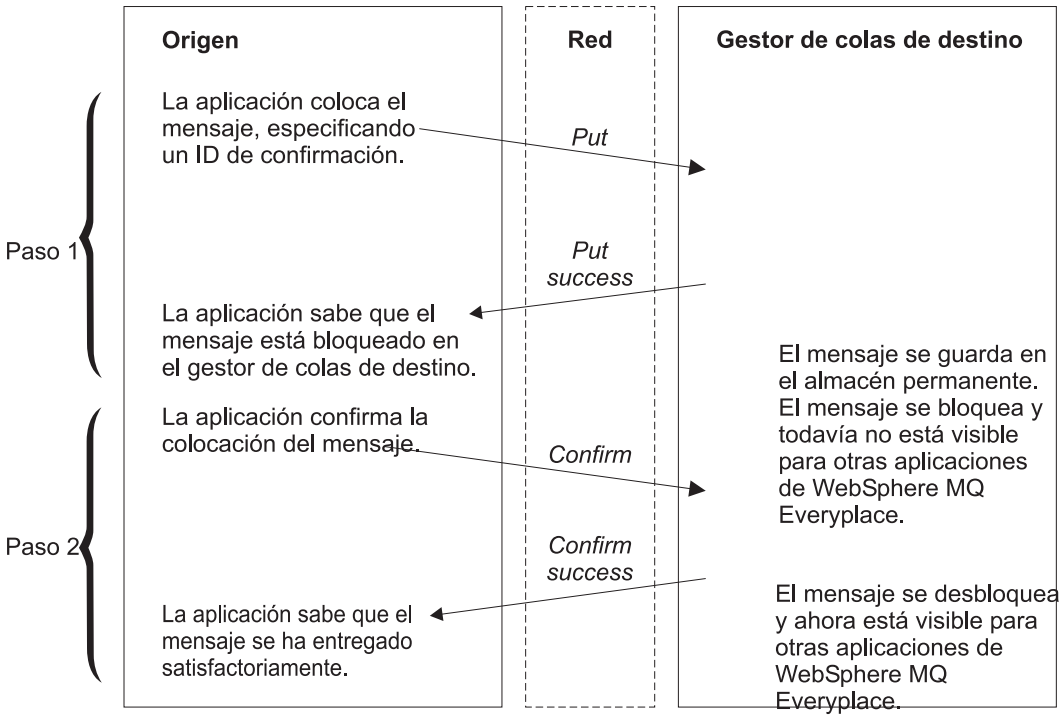

Figura 6. Transferencia de mensajes asegurada síncrona

El método mqeQueueManager_confirmPutMessage() exige que se especifique el ID exclusivo (UniqueID) del mensaje, no el ID de confirmación (confirmID) que se utiliza en el mandato de mensaje de transferencia anterior. El ID de confirmación (confirmID) se utiliza para restaurar mensajes que permanecen bloqueados tras haberse producido una anomalía en la transmisión.

Ejemplo (Java) - put asegurado:

A continuación, se muestra una versión esquematizada del código requerido para una transferencia put asegurada:

```
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     */
}
```

Ejemplo (C): transferencia (put) asegurada:

A continuación, se muestra una versión esquematizada del código requerido para una transferencia put asegurada:

```
/* 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);
          }
    }
}
```

Manejos de excepciones: transferencia (put) de mensaje:

Si se produce una anomalía durante el paso 1 de la "Transferencia (put) de mensaje: transferencia (put) asegurada" en la página 53, la aplicación debe volver a transmitir el mensaje. No hay peligro de introducir mensajes por duplicado en la red de MQe puesto que el mensaje del gestor de colas de destino no se visualizará en las aplicaciones hasta que el flujo de confirmación se haya procesado correctamente.

Si la aplicación de MQe vuelve a transmitir el mensaje, también deberá informar al gestor de colas de destino al respecto. El gestor de colas de destino suprime cualquier copia duplicada del mensaje que ya posee. La aplicación establece el campo `MQE_MSG_RESENDMQe.Msg_Resend` para tal fin.

Si se produce una anomalía durante el paso 2 de la "Transferencia (put) de mensaje: transferencia (put) asegurada" en la página 53, la aplicación debería enviar el flujo de confirmación de nuevo. No hay peligro al realizar esta operación ya que el gestor de colas de destino ignora cualquier flujo de confirmación que reciba de los mensajes que ya ha confirmado. Esto se muestra en el siguiente ejemplo, extraído de examples.application.example6.

Ejemplo - Java:

Este ejemplo se ha extraído de la aplicación de ejemplo examples.application.example6:

```
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 ++;
  }
}
```

Ejemplo - C:

Este ejemplo se ha extraído de la aplicación de ejemplo examples.application.example6:

```
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 mensaje - get asegurado

El mensaje asegurado `get` funciona de un modo parecido a `put`. Si un mandato de mensaje `get` se emite con un parámetro ID de confirmación (`confirmId`) mayor que cero, el mensaje se queda bloqueado en la cola en la que reside hasta que el gestor de colas de destino procesa un flujo de confirmación. Cuando se recibe un flujo de confirmación, el mensaje se suprime de la cola. En Figura 7 en la página 58 se describe una obtención de mensajes síncrona:

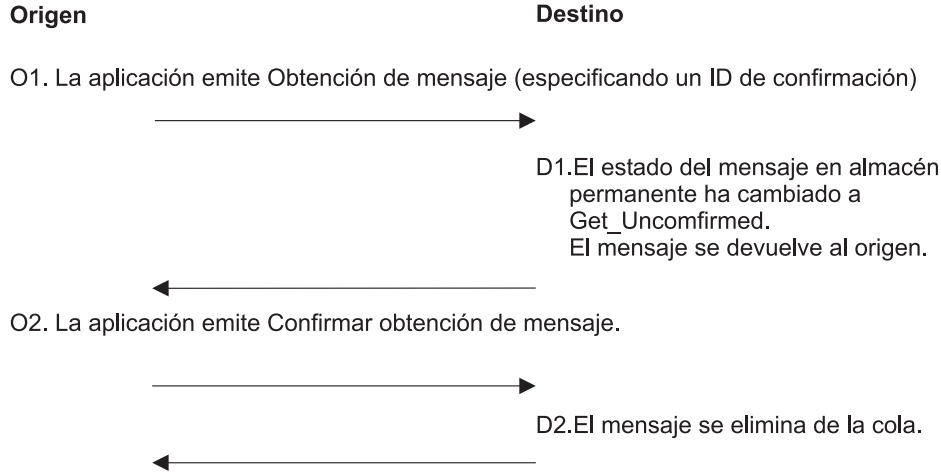

Figura 7. Obtención asegurada de mensajes síncronos

Ejemplo (Java) - get asegurado:

Este código de ejemplo se ha extraído del programa de ejemplo examples.application.example6.

```
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    */
  }
}
```

Ejemplo (C) - get asegurado:

Este código de ejemplo se ha extraído del programa de ejemplo examples.application.example6.

```
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);
    }
}
```

Mandato deshacer:

El valor que se ha pasado como parámetro ID de confirmación (`confirmId`) también tiene otro uso. El valor se utiliza para identificar el mensaje mientras está bloqueado y esperando la confirmación. Si se produce un error durante una operación `get`, el mensaje puede quedarse potencialmente bloqueado en la cola. Esto sucede si el mensaje está bloqueado en respuesta al mandato `get`, pero se produce un error antes de que la aplicación reciba el mensaje. Si la aplicación vuelve a emitir el mandato `get` como respuesta a la excepción, no podrá obtener el mismo mensaje porqué estará bloqueado y no será visible para las aplicaciones de MQe.

Sin embargo, la aplicación que ha emitido el mandato `get` puede restaurar los mensajes mediante el método `undo` . La aplicación debe suministrar el valor ID de confirmación (`confirmId`) proporcionado al mandato de mensaje `get`. El mandato `undo` restaura el estado que tenían los mensajes antes de emitir el mandato `get`.

El mandato `undo` también es válido para los mandatos mqeQueueManager_`putMessage` y mqeQueueManager_`browseMessagesAndLock`. Al igual que sucede con el mandato `get`, el mandato `undo` restaura el anterior estado de cualquier mensaje bloqueado por el mandato mqeQueueManager_`browseMessagesandLock`.

Si una aplicación emite un mandato `undo` después de un mandato mqeQueueManager_`putMessage` anómalo, se suprimen los mensajes bloqueados en la cola de destino a la espera de confirmación.

El mandato `undo` funciona en operaciones tanto de colas locales como de colas remotas.

Ejemplo de mandato Undo: 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    */
  }
}
```

Ejemplo de mandato Undo - 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);
}
```

Topologías de red y resolución de mensajes

Introducción a las rutas de mensajes y su uso con MQe.

Visión general

En este tema se explica de forma detallada el concepto de ruta de mensaje y cómo utilizarlo con MQe.

Varias funciones de MQe permiten que se altere de forma dinámica la ruta de los mensajes. No obstante, tiene que asegurarse de que no haya mensajes 'dudosos' a los que pudiera afectar el cambio. Si se transfiere un mensaje con un ID de confirmación que no sea cero y, a continuación, se cambia la topología

de red de MQe para alterar la ruta de la invocación posterior de confirmGetMessage, no se encontrará el mensaje sin confirmar. El protocolo de MQe considera un error de confirmación de transferencia como una indicación de que el mensaje de transferencia ya se ha confirmado y, por lo tanto, presupone que la ejecución ha sido satisfactoria. Esto puede dejar un mensaje sin confirmar en una cola, lo que representa que el mensaje se pierde y, por lo tanto, se deja de cumplir la promesa de entrega asegurada.

Puesto que MQe utiliza el mismo proceso de dos pasos para asegurar la entrega de mensajes enviados de forma asíncrona, independientemente de si se ha utilizado un ID de confirmación cero o no cero, la modificación de la topología de red puede dejar de cumplir la entrega asegurada de envío de mensajes asíncronos.

Notación

Los temas de *Topologías de red y resolución de mensajes* utilizan una notación coherente para ilustrar los recursos. Esto permite que se puedan mostrar con prioridad las áreas de interés específico, mientras que se pueden ocultar las partes menos relevantes de un sistema. Esto es más sencillo mostrarlo con un diagrama:

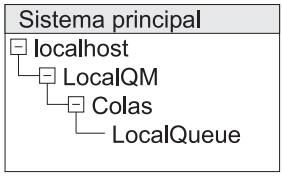

Figura 8. Sistema principal y sus recursos de MQe

En el diagrama siguiente se muestran los mismos recursos pero de forma 'dispersa':

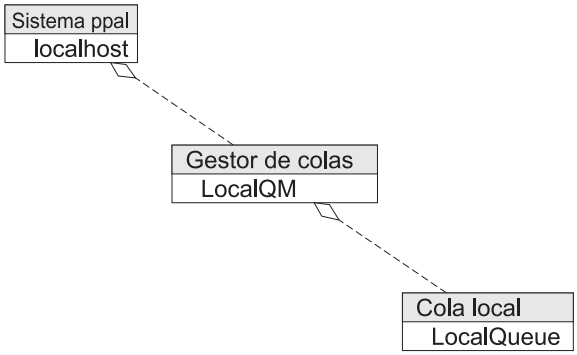

Figura 9. Sistema principal y sus recursos: forma 'dispersa'

La línea que aparece con un rombo muestra que el gestor de colas es un elemento subordinado del sistema principal. Con ello se mantiene la relación entre elemento superior y elemento subordinado del árbol, que se perdería si se separaran los elementos.

Introducción

La ruta que un mensaje toma a través de una red de MQe puede depender de muchos recursos (colas, definiciones de conexión, escuchas, etc). Estos recursos tienen que configurarse correctamente, a menudo, en pares cuyos valores tienen que ser complementarios. Si no se configuran los recursos correctos o se establecen algunos de sus valores de forma incorrecta, puede que los mensajes no se entreguen. Puesto que la tarea de configurar una red que direccione correctamente mensajes puede parecer compleja inicialmente, en este tema se describe la teoría subyacente a la resolución de mensajes.

Una fuente común de confusión con MQe es la diferenciación entre una cola local que reside en una máquina remota (o gestor de colas) y una definición local de esa cola en la máquina remota. Estas dos entidades se denominan 'colas remotas'. Para aclarar esto, el término 'referencia de cola remota' se utiliza para describir una definición local de una cola que reside en otra máquina (remota) (o gestor de colas).

Resolución de cola local

La transferencia de mensajes locales es fundamental para MQe. Los mensajes, si son útiles, deben acabar siempre en una cola local. La resolución de rutas de mensaje es el mecanismo por el que un mensaje se desplaza a través de una red de MQe a su destino último.

En el diagrama siguiente se muestra una transferencia de mensaje local.

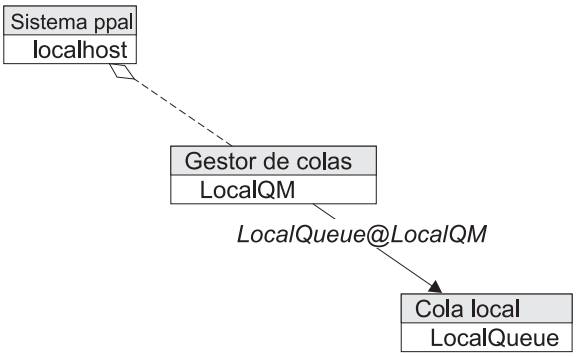

Figura 10. Transferencia de mensaje local

Se muestra la ruta de un mensaje transferido a (QueueManager)LocalQM y destinado a (Queue)LocalQueue@LocalQM. Esto es claramente una transferencia a una cola remota, ya que el 'nombre del gestor de colas' de la cola es el mismo que el nombre del gestor de colas al que se transfiere el mensaje.

La ruta del mensaje se muestra con una flecha etiquetada con el nombre de la ruta del mensaje. La flecha indica la dirección en la que el mensaje fluye. El texto de la etiqueta indica el nombre de destino utilizado actualmente (puede cambiar durante la resolución de mensajes). El gestor de colas local (LocalQM) busca una cola para aceptar un mensaje para LocalQueue@LocalQM. El proceso de determinación de la cola en la que se debe colocar un mensaje se denomina resolución de colas. El gestor de colas local (LocalQM) busca una coincidencia exacta del destino, la cola local. A continuación, transfiere el mensaje a la cola local. El mensaje residirá, a continuación, en la cola local hasta que se recupere a través de la llamada a la API getMessage().

Alias de cola local

Las colas locales pueden tener alias. Si añadimos un alias de cola a la cola local, le proporcionamos otro nombre por el que se la reconocerá. Por lo tanto, se puede asignar el alias 'LocalQueueAlias' a la cola local LocalQueue@LocalQM, como se muestra en el diagrama siguiente:

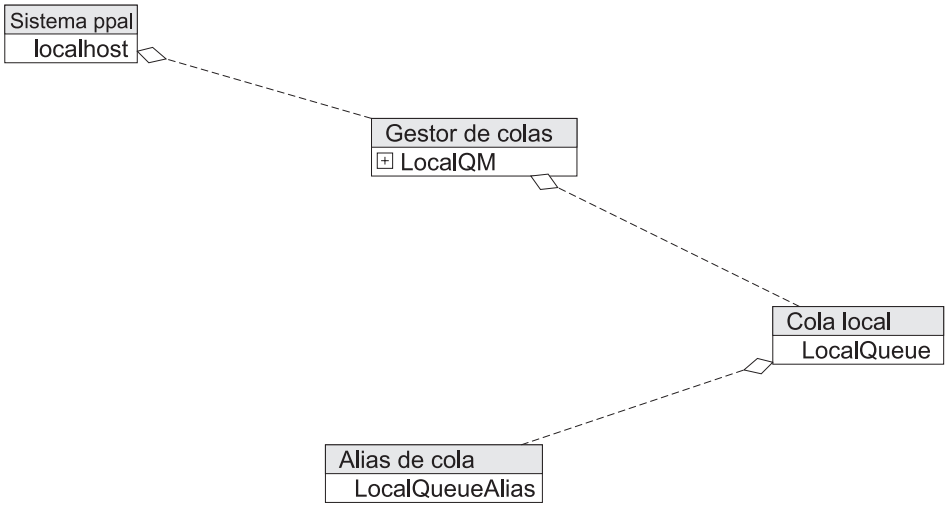

Figura 11. LocalQueue@LocalQM con un alias 'QueueAlias'.

El gestor de colas redirigiría a LocalQueue@LocalQM los mensajes dirigidos a LocalQueueAlias@LocalQM. Podríamos considerar esto como que el mensaje se coloca en el alias coincidente, casi como si el alias estuviera en una cola y, a continuación, el alias moviera el mensaje al destino correcto, como se muestra en el diagrama siguiente:

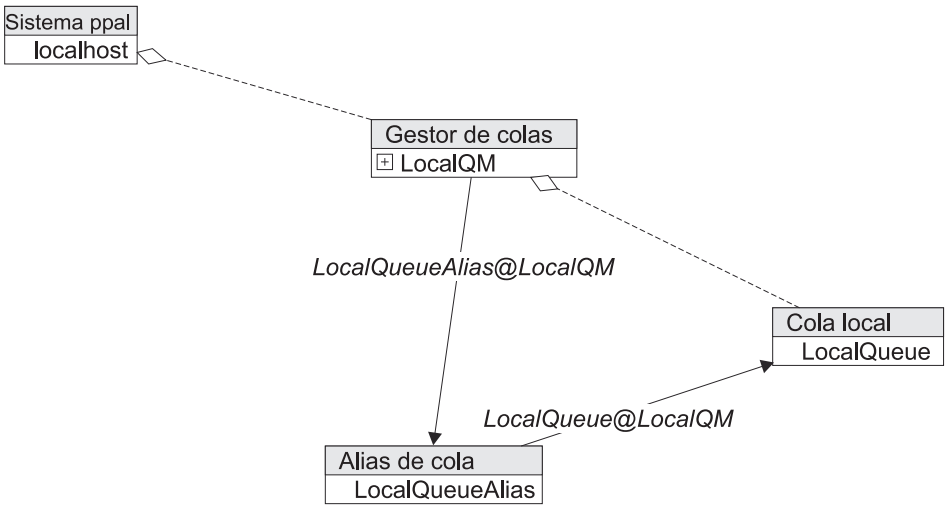

Figura 12. Mensaje que se coloca en un alias coincidente

El redireccionamiento del mensaje por parte del alias va acompañado por un cambio en el 'nombre de la cola de destino' de LocalQueueAlias@LocalQM a LocalQueue@LocalQM. Se pierde totalmente el hecho de que el mensaje se hubiera transferido originalmente al alias. Esto se puede ver en la etiqueta de la ruta del mensaje del alias a la cola. En este caso concreto, el cambio de 'nombre de transferencia' es de poca o ninguna importancia, pero sí que es importante en resoluciones de mensajes más complejas.

La resolución del alias de cola se realiza justo antes de que el mensaje se direccione a la cola. La resolución se retrasa tanto como es posible y, en algunas ocasiones, se denomina 'resolución tardía'.

Alias de gestor de colas

Los alias de colas permiten hacer referencia a colas mediante más de un nombre. Los alias de gestores de colas permiten hacer referencia a gestores de colas mediante más de un nombre. Podemos definir un alias

de gestor de colas 'AliasQM' que haga referencia al gestor de colas local, como se muestra en el diagrama siguiente:

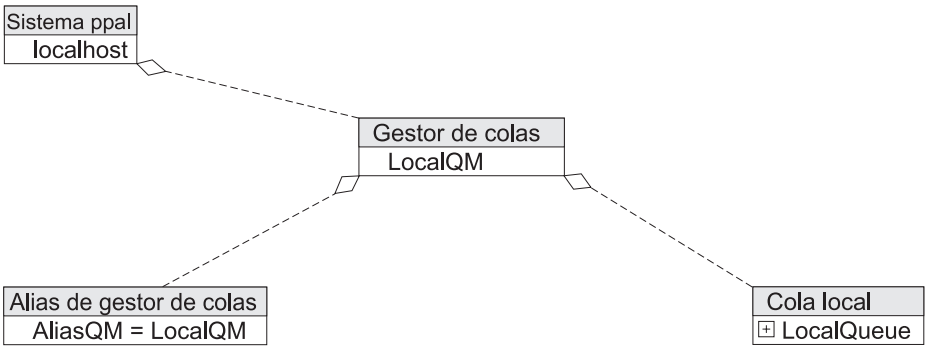

Figura 13. Definición de un alias de gestor de colas

Los mensajes dirigidos a 'AliasQM' se direccionan a 'LocalQM', como se muestra en el diagrama siguiente:

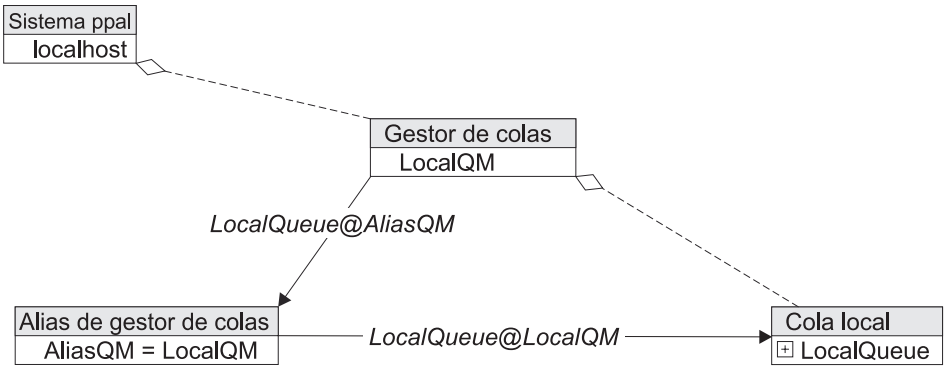

Figura 14. Direccionamiento de mensajes a un alias de gestor de colas

El redireccionamiento del mensaje por parte del alias va acompañado por un cambio en el 'nombre de la cola de destino' de LocalQueue@AliasQM a LocalQueue@LocalQM. Se pierde totalmente el hecho de que el mensaje se hubiera transferido originalmente al alias. Esto se puede ver en la etiqueta de la ruta del mensaje del alias a la cola. Los alias de gestores de colas se resuelven al principio de la resolución de mensajes. Los alias de gestores de colas son muy efectivos como parte de topologías complejas.

Para finalizar el ejemplo, podemos resolver tanto el alias de gestor de colas como el alias de cola, como se muestra en el diagrama siguiente:

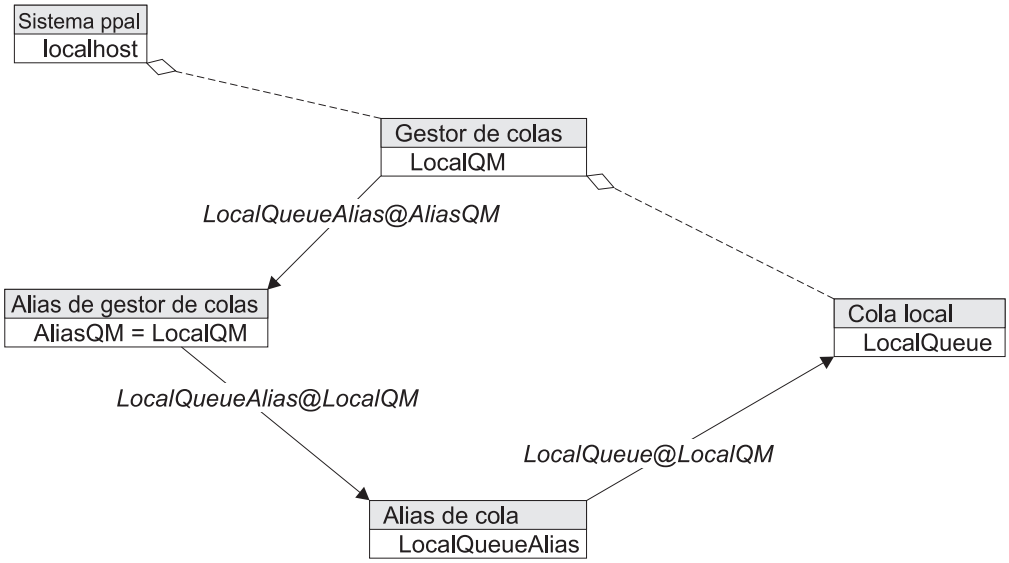

Figura 15. Resolución del alias de gestor de colas y el alias de cola

Aquí hemos transferido un mensaje a LocalQueueAlias@AliasQM y se ha resuelto, en primer lugar, a través del alias de gestor de colas y, a continuación, a través del alias de cola.

La resolución de los alias de gestores de colas tiene lugar en el momento en el que la petición llega a un gestor de colas. El efecto es sustituir la serie de caracteres con alias por la serie de caracteres de asignación de alias. Por tanto, en el primer ejemplo anterior, en el momento en que la invocación de putMessage(″AliasQM″,....) cruza la API, se convierte en una invocación de putMessage(″LocalQM″,....) . Esta resolución también tiene lugar cuando se transfiere un mensaje a un gestor de colas remoto. En un gestor de colas remoto, se utilizan los alias de colas de ese gestor de colas y no los del gestor de colas de origen.

Un alias puede apuntar a otro alias. No obstante, las definiciones circulares tienen resultados imprevisibles. También se puede crear un alias del nombre del gestor de colas local. Esto permite que un gestor de colas funcione como si fuera otro gestor de colas. Esto significa que podemos eliminar un gestor de colas completamente de la red y, creando los alias de gestor de colas adecuados en otras ubicaciones, podemos asignar su carga de trabajo a otro gestor de colas. Esta función es útil cuando se modifican las topologías de MQe, porque los servidores, bajo el control de los administradores del sistema, se pueden mover, eliminar o renombrar sin romper la conectividad de los clientes, que puede que no sean tan accesibles.

Resolución de cola remota

La resolución de colas remotas implica definiciones de conexión y resolución de red. Exige una configuración en la que hay dos gestores de colas: uno es el gestor de colas local que se utiliza para transferir el mensaje y el otro es el gestor de colas al que desea que vaya el mensaje. El gestor de colas remoto debe disponer de un escucha y el gestor de colas local debe disponer de una definición de conexión que describa al escucha, como se muestra en el diagrama siguiente:

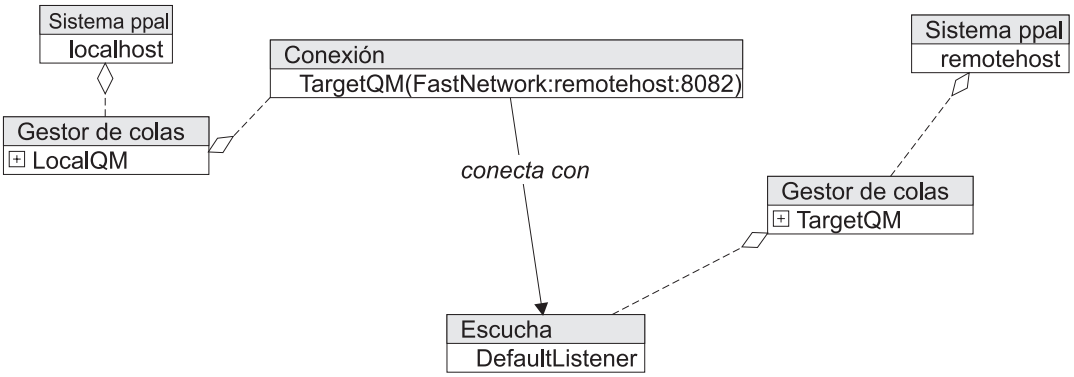

Figura 16. Gestores de colas local y remoto con un par de definición y escucha

El par de definición de conexión/escucha permite que MQe establezca las comunicaciones de red necesarias para que el mensaje fluya. La definición de conexión contiene información acerca de la comunicación con un solo gestor de colas. La definición de conexión recibe un nombre para el gestor de colas para el que define una ruta. Por lo tanto, en este ejemplo, la definición de conexión se denomina TargetQM y contiene la información necesaria para establecer la conexión con (QueueManager)TargetQM. Esta información incluye la dirección de la máquina en la que reside el gestor de colas (el sistema principal remoto en este ejemplo), el puerto en el que el gestor de colas está a la escucha (8081 en este ejemplo) y el protocolo que se debe utilizar al conversar con el gestor de colas (FastNetwork en este ejemplo).

Necesita una referencia de cola remota en LocalQM que represente la cola de destino TargetQueue que reside en TargetQM. Existen, pues, dos entidades denominadas TargetQueue@TargetQM. Una es la cola 'real', es decir, una cola local, y la otra es una referencia a la cola real, una referencia de cola remota, como se muestra en el diagrama siguiente:

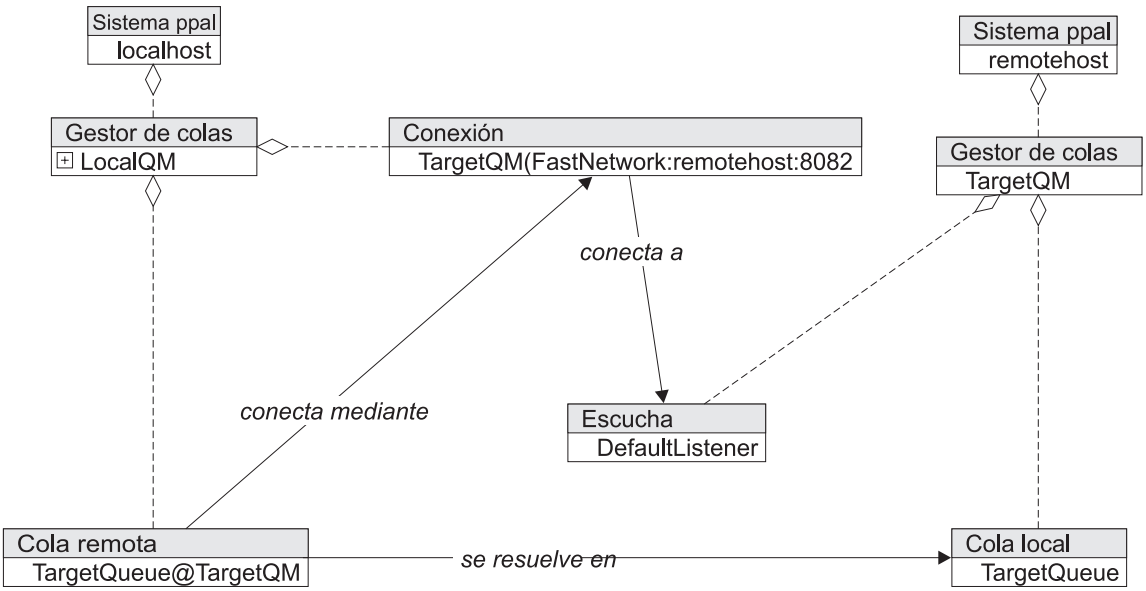

Figura 17. Referencia de cola remota

La resolución de mensajes para una transferencia en LocalQM a TargetQueue@TargetQM funciona como se muestra en el diagrama siguiente:

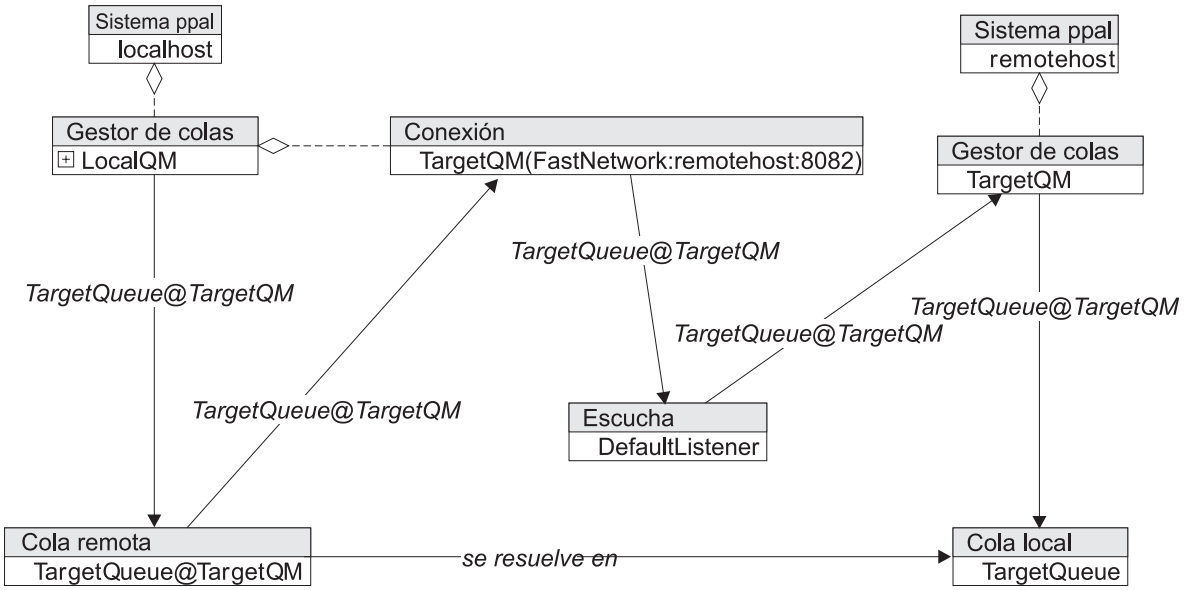

Figura 18. Resolución de mensajes para una transferencia

La ruta del mensaje es la siguiente:

- El mensaje se transfiere en LocalQM con destino a TargetQueue@TargetQM.
- LocalQM realiza una resolución de colas y busca la referencia de cola remota como coincidencia exacta. LocalQM coloca el mensaje en la referencia de cola remota.
- La referencia de cola remota realiza, a continuación, la resolución de conexiones. Busca una conexión que le permita pasar el mensaje al gestor de colas propietario de la cola final. La referencia de cola remota busca el TargetQM invocado por la definición de conexión y le pasa el mensaje.
- La definición de conexión ahora traslada el mensaje al escucha socio, que transfiere el mensaje al gestor de colas remoto.
- El gestor de colas remoto realiza la resolución de colas como si el mensaje se hubiera transferido localmente, busca TargetQueue@TargetQM y transfiere el mensaje en él.

Aunque la definición de conexión y el escucha son vitales para la resolución de mensajes, no afectan al direccionamiento en este ejemplo. Esto se muestra en el diagrama siguiente:

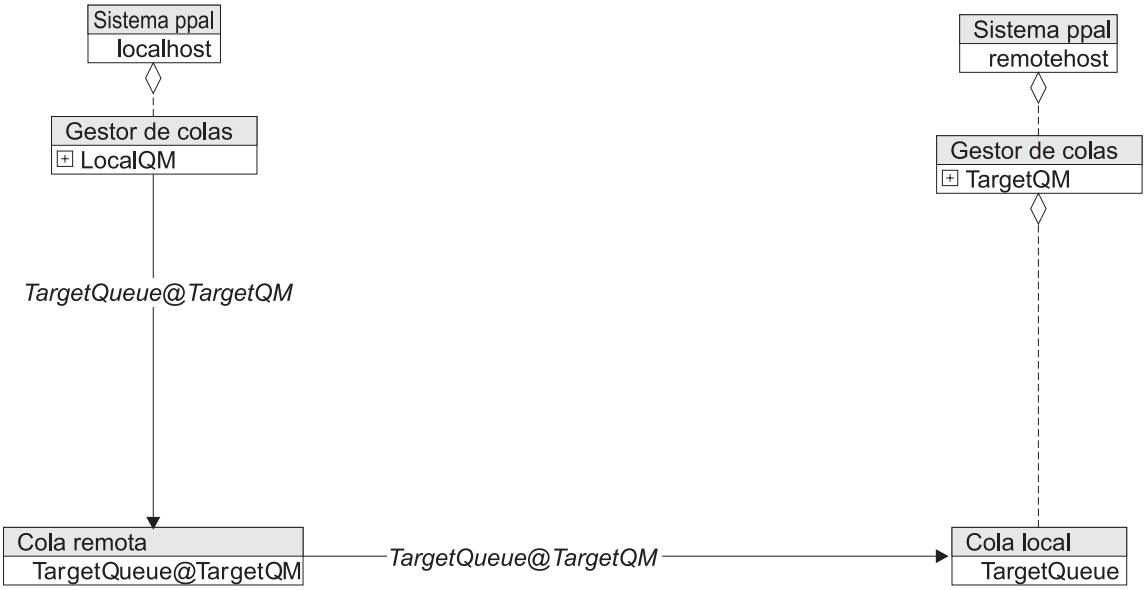

Figura 19. Resolución de mensajes para una transferencia

En mensajes posteriores, las definiciones de conexión juegan un papel más importante y se muestran de forma explícita. Por ahora, suponga que está presente el enlace lógico formado por el escucha y que no se muestra en los diagramas. A menudo es mucho más práctico utilizar una visión simplificada de la ruta de los mensajes. Puede hacerlo si considera como entidad única compuesta a los cuatro elementos que participan en esta resolución de mensajes. Esa entidad es una ruta de mensajes, como se muestra en el diagrama siguiente:

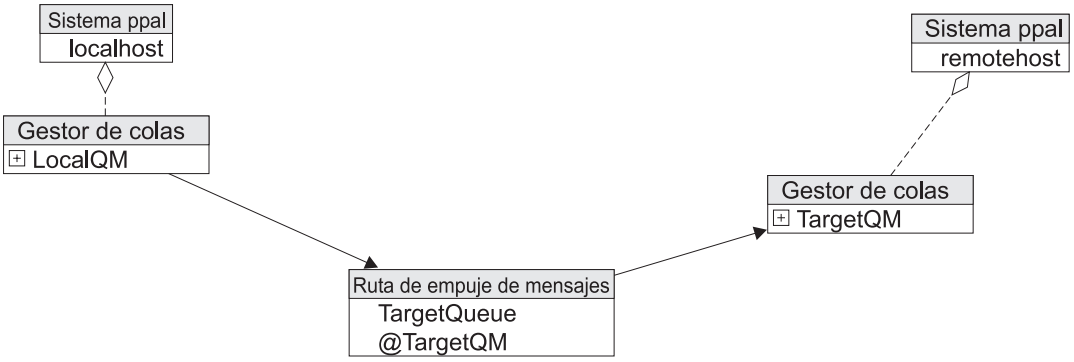

Figura 20. Entidad de ruta de mensajes

Aquí puede ver la ruta de mensajes que indica que todos los mensajes transferidos a LocalQM y dirigidos a TargetQueue@TargetQM se trasladarán directamente al destino. Una ruta de mensajes sólo es válida si todos los componentes necesarios (definición de conexión, escucha, definición de cola remota y cola de destino) están presentes y están bien configurados.

La ruta de mensajes se define como una ruta de empuje de mensajes porque LocalQM empuja los mensajes de la cola de origen a la cola de destino.

Aliasen colas remotas

Puede utilizar alias en la cola remota ya que el último paso es simplemente la resolución de colas realizada en TargetQM. El alias de cola de la cola de destino es considerado por el sistema local como una cola. La definición de cola remota del sistema local recibe un nombre, pues, para el alias de cola, en

lugar de para la cola remota. El diagrama siguiente aclara lo anterior (tenga en cuenta que la definición de conexión y el escucha están ocultos):

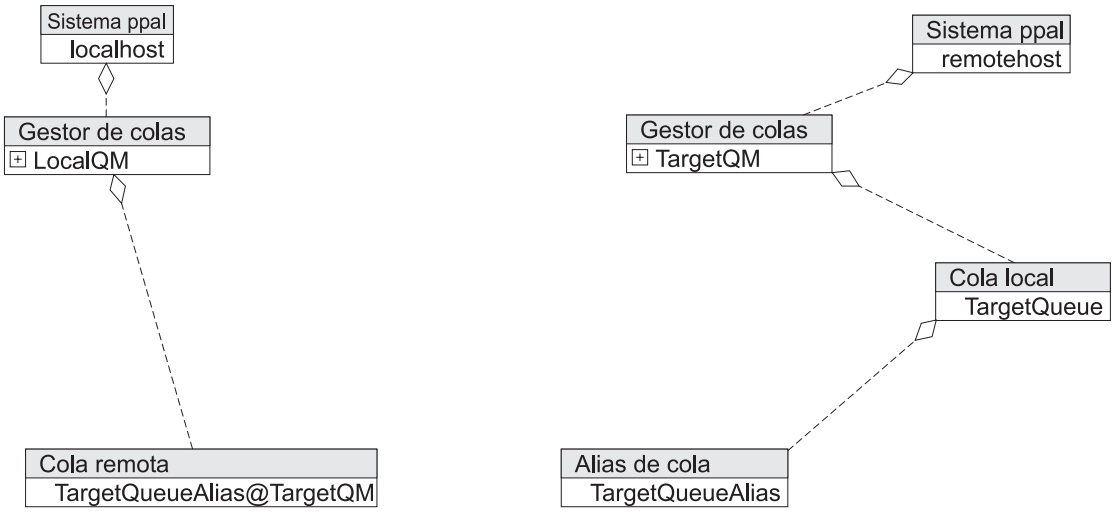

Figura 21. Utilización de alias en la cola remota

Aquí se define una referencia de cola remota que, de hecho, hace referencia a un alias para la cola de TargetQM. Cuando se realiza una transferencia en LocalQM dirigida a QueueAlias@TargetQM, la resolución funciona como se muestra en el diagrama siguiente:

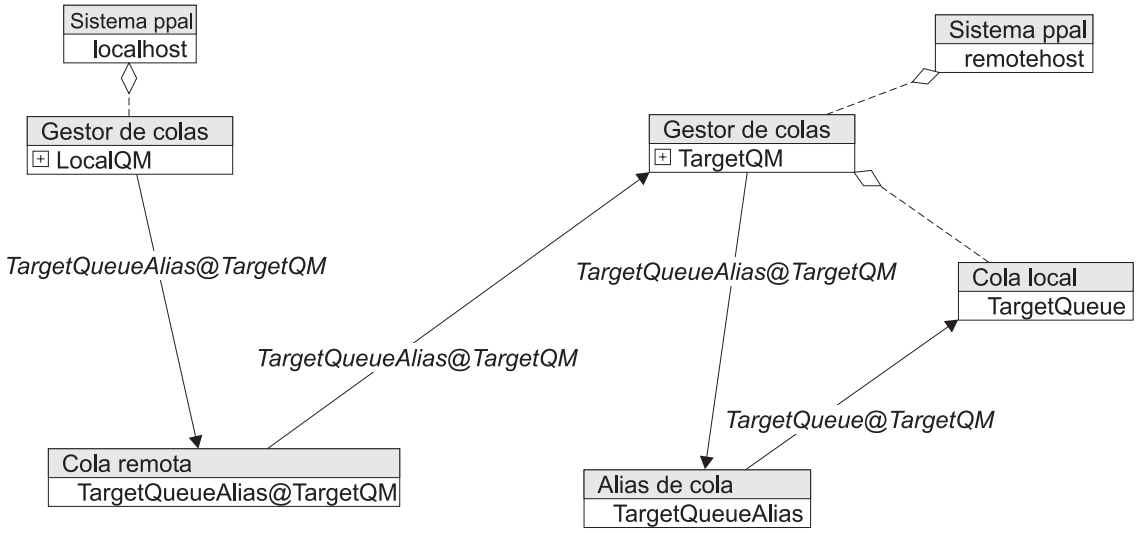

Figura 22. Resolución de mensajes para una transferencia a una cola remota, con un alias de cola definido en TargetQM

- La resolución de colas en LocalQM busca la referencia de cola remota. El hecho de que sea una referencia a un alias de cola no tiene ninguna importancia para la resolución de colas.
- La resolución de conexiones funciona exactamente como se describe más arriba.
- La resolución de colas en TargetQM ahora funciona exactamente como la resolución de colas locales de un alias de cola que se ha descrito anteriormente.

Tenga en cuenta que el nombre de destino del mensaje continúa siendo QueueAlias@TargetQM hasta la resolución de colas en TargetQM. La definición de cola remota cumple los requisitos de otra ruta de mensajes, como se muestra en el diagrama siguiente:

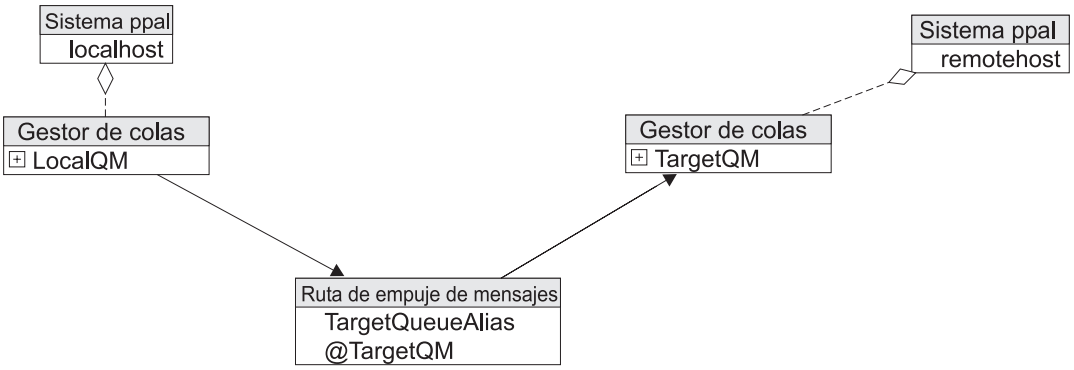

Figura 23. Entidad de ruta de mensajes transferidos a TargetQueueAlias en TargetQM

Rutas paralelas

Los alias permiten la creación de rutas paralelas entre un origen y un destino. Esto puede resultar útil en ocasiones cuando se desean enviar mensajes, si es posible, de forma síncrona o de forma asíncrona, si el extremo remoto no está conectado en ese momento. Puede hacerlo con la configuración que se muestra en el diagrama siguiente:

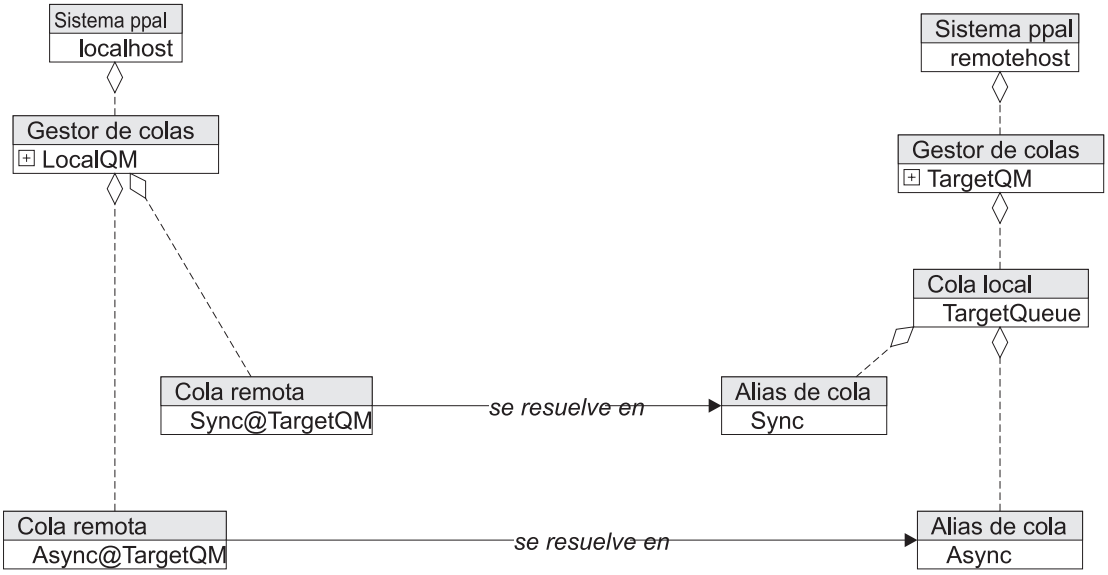

Figura 24. Creación de rutas paralelas entre un origen y un destino

Aquí se han definido dos alias en la cola de destino. Un alias se utilizará para direccionar el tráfico síncrono a la cola de destino y el otro se utilizará para direccionar el tráfico asíncrono.

En LocalQM se han creado dos definiciones de colas remotas y cada una de ellas apunta a un alias. Puede crear un Async@TargetQM invocado por la definición de cola remota asíncrona y un Sync@TargetQM invocado por la definición de cola remota síncrona. Al elegir el nombre de la cola a la que se transfiere (Sync@TargetQM o Async@TargetQM) podrá elegir la ruta que el mensaje seguirá, aunque el destino sea el mismo. En primer lugar, la resolución de la ruta síncrona transfiriendo un mensaje a Sync@TargetQM, como se muestra en el diagrama siguiente:

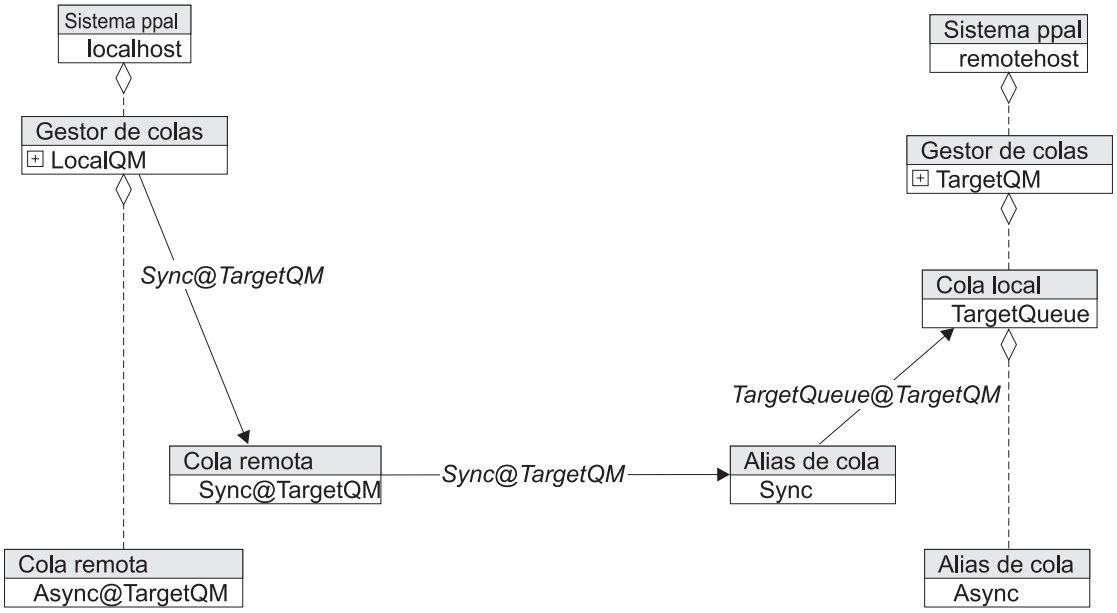

Figura 25. Resolución de la ruta síncrona

Y, en segundo lugar, la resolución asíncrona mediante AsyncAlias@TargetQM, como se muestra en el diagrama siguiente:

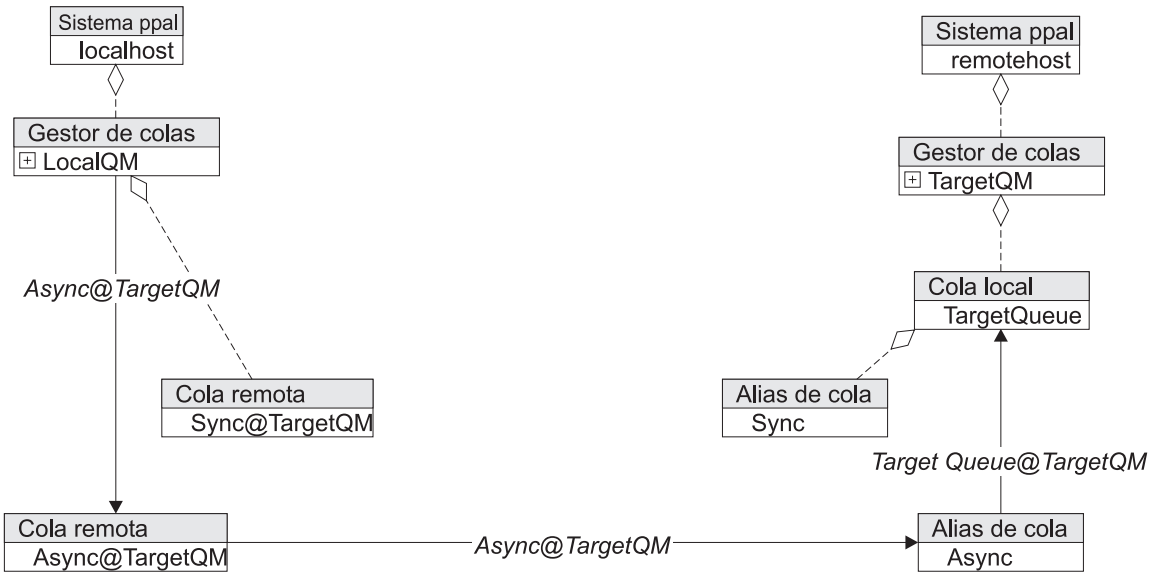

Figura 26. Resolución de la ruta asíncrona

Puede considerar esto como un par de rutas de empuje de mensajes, como se muestra en el diagrama siguiente:

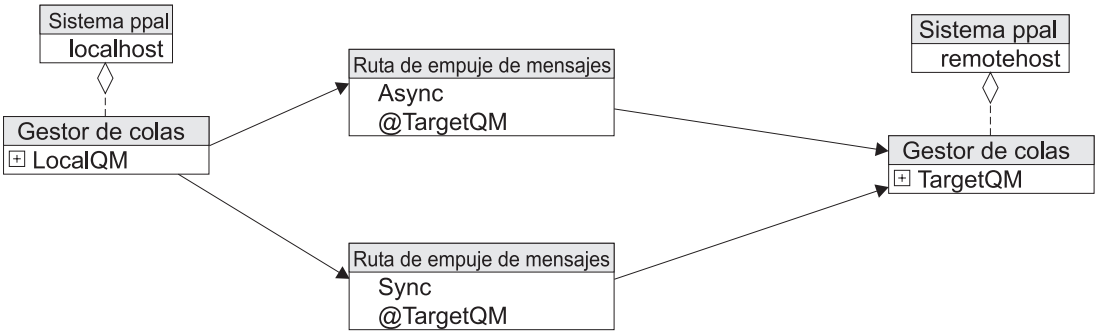

Figura 27. Par de rutas de empuje de mensajes

Encadenamiento de referencias de colas remotas

Las referencias de colas remotas se pueden encadenar juntas para formar una ruta larga. Esto exige el uso de "Conexiones de tipo vía" en la página 79.

Empuje de colas para almacenar y reenviar

MQe dispone de un tipo de colas que acepta mensajes por gestor de colas en lugar de hacerlo por cola. Estas colas se denominan colas para almacenar y reenviar (store and forward, S&F). Las colas S&F mantienen una lista de nombres de gestores de colas que se denominan entradas de gestor de cola (Queue Manager Entries, QME). La cola S&F aceptará los mensajes para cualquier gestor de colas representado por una QME. Esta aceptación es independiente del nombre de la cola de destino y, por lo tanto, permite que una cola (la cola S&F) direccione todos los mensajes para un gestor de colas determinado o para varios gestores de colas.

Las colas S&F pueden funcionar en dos modalidades: modalidad de empuje y modalidad de extracción. En la modalidad de empuje, los mensajes se trasladan al siguiente gestor de colas como con las referencias de colas remotas. En la modalidad de extracción, los mensajes se eliminan de la cola S&F gracias a una cola de servidor local. En este apartado sólo se describe el empuje de mensajes; la extracción de mensajes con un servidor local se describe en otro apartado. Un sistema de colas S&F de empuje puede tener este aspecto:

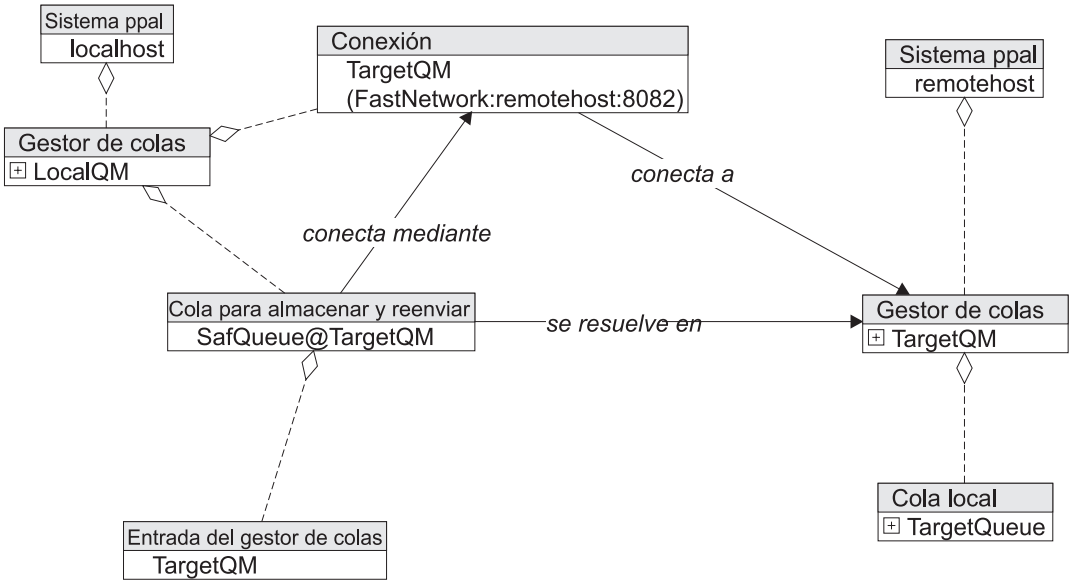

Figura 28. Sistema de colas S&F de empuje

Una SafQueue invocada por una cola S&F tiene una entrada de gestor de colas (QME) para TargetQM. Esto le permite aceptar mensajes para cualquier cola en TargetQM. Al igual que con las colas remotas típicas, una cola para almacenar y reenviar exige un par de definición de conexión/escucha configurado para empujar los mensajes. A diferencia de una definición de cola remota, una cola para almacenar y reenviar empuja los mensajes a un gestor de colas, en lugar de a una cola. El mensaje llega al gestor de colas, donde se lleva a cabo la resolución de colas. Cuando un mensaje se transfiere a un LocalQM dirigido a TargetQ@TargetQM, la resolución es la siguiente:

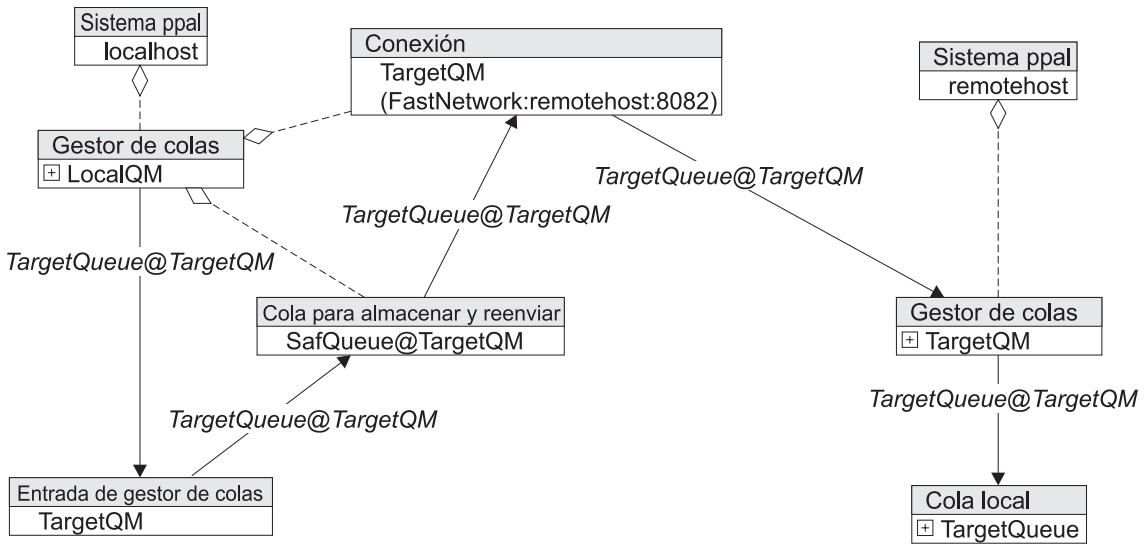

Figura 29. Direccionamiento de un mensaje transferido a LocalQM y dirigido a TargetQ@TargetQM

- LocalQM realiza una resolución de colas que busca el TargetQM de entrada de gestor de colas en SafQueue. LocalQM transfiere el mensaje a la QME.
- La transferencia de un mensaje a la QME es equivalente a transferir el mensaje en la cola S&F propietaria de la QME.
- La cola S&F realiza la resolución de conexiones y busca la definición de conexión y, por lo tanto, la utiliza para empujar mensajes a RemoteQM.

- El gestor de colas realiza, a continuación, la resolución de colas y coloca el mensaje en la cola de destino.

La cola para almacenar y reenviar forma parte de una ruta de varios mensajes. Esta entidad abstracta representa el potencial de los mensajes dirigidos a cualquier cola en TargetQM y, por ello, se denomina *@TargetQM, como se muestra en el diagrama siguiente:

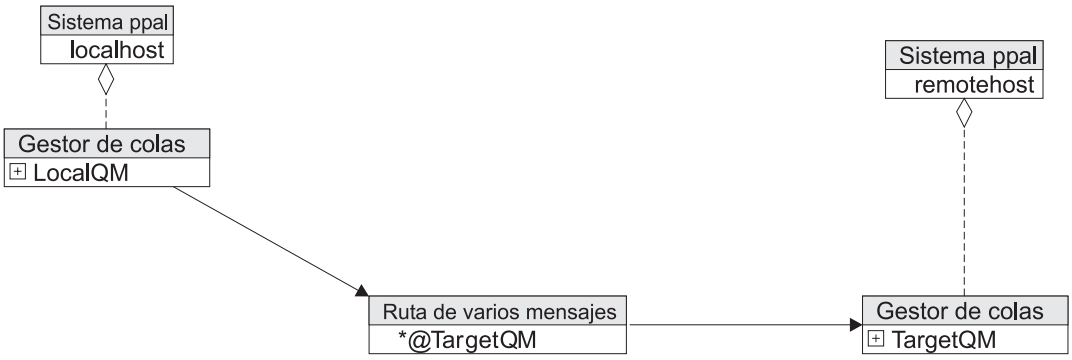

Figura 30. Ruta de varios mensajes

Si no hay ninguna cola a la que transferir el mensaje, éste no se entrega. Esto impide que se empujen más mensajes de la cola para almacenar y reenviar a ese gestor de colas.

REferencias de colas S&F y cola remota

Puesto que las colas para almacenar y reenviar (S&F) pueden aceptar mensajes para cualquier cola de un gestor de colas determinado, puede parecer que entran en conflicto con una referencia de cola remota. En tal caso, la referencia de cola remota tiene prioridad, porque es más específica. Por ello, si se añade una referencia de cola remota a la resolución de colas S&F, la resolución de rutas de mensajes cambia de inmediato y la cola S&F pasa a ser irrelevante, como se muestra en el diagrama siguiente:

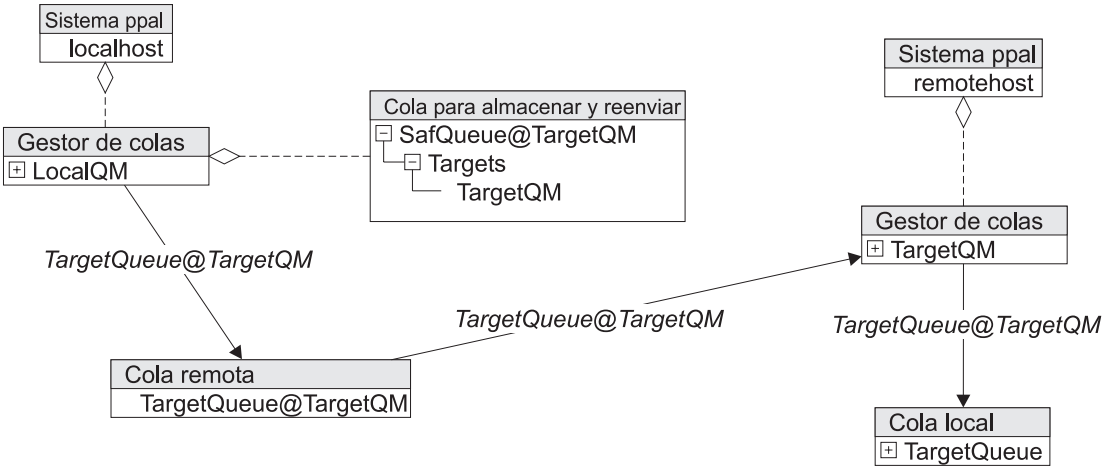

Figura 31. Prioridad de las rutas con definición de colas remotas sobre las rutas de colas para almacenar y reenviar

La resolución de colas busca la mejor coincidencia (más exacta) para la dirección del mensaje.

Por lo tanto, un mensaje transferido a QueueAlias@TargetQM se desplaza a través de la cola S&F (transmisión asíncrona), pero una transferencia a TargetQueue@TargetQM se desplaza de forma síncrona a través de la referencia de cola remota.

Encadenamiento de colas S&F

Las colas para almacenar y reenviar de empuje se pueden encadenar conjuntamente en una ruta más compleja, como se muestra en el diagrama siguiente:

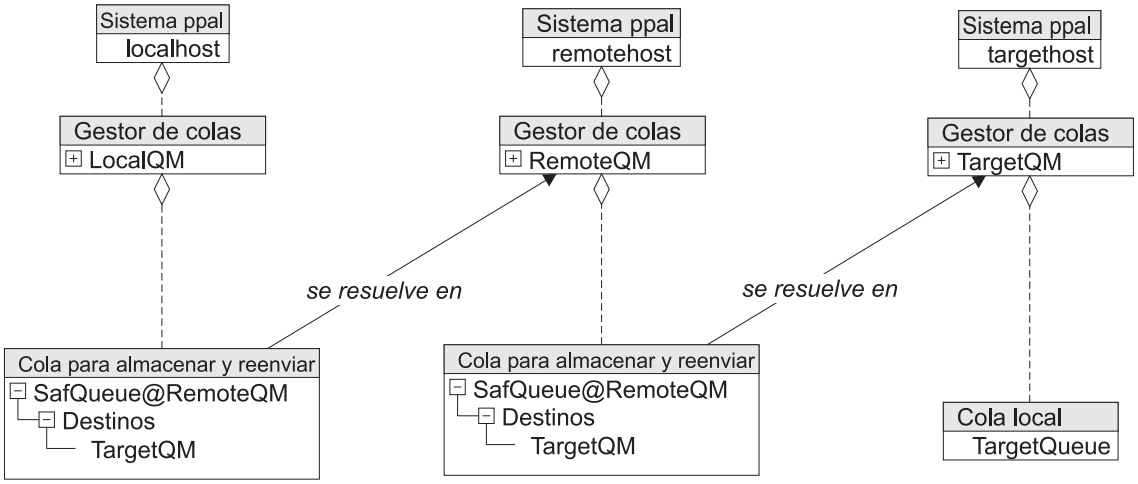

Figura 32. Colas S&F de empuje encadenadas conjuntamente

La cola para almacenar y reenviar en LocalQM (SaFQueue@RemoteQM) tiene una entrada de gestor de colas para TargetQM, pero en realidad empuja a RemoteQM. LocalQM exige una definición de conexión con RemoteQM, pero no con TargetQM. A continuación, se puede transportar un mensaje a través de la cola S&F intermedia, como se muestra en el diagrama siguiente:

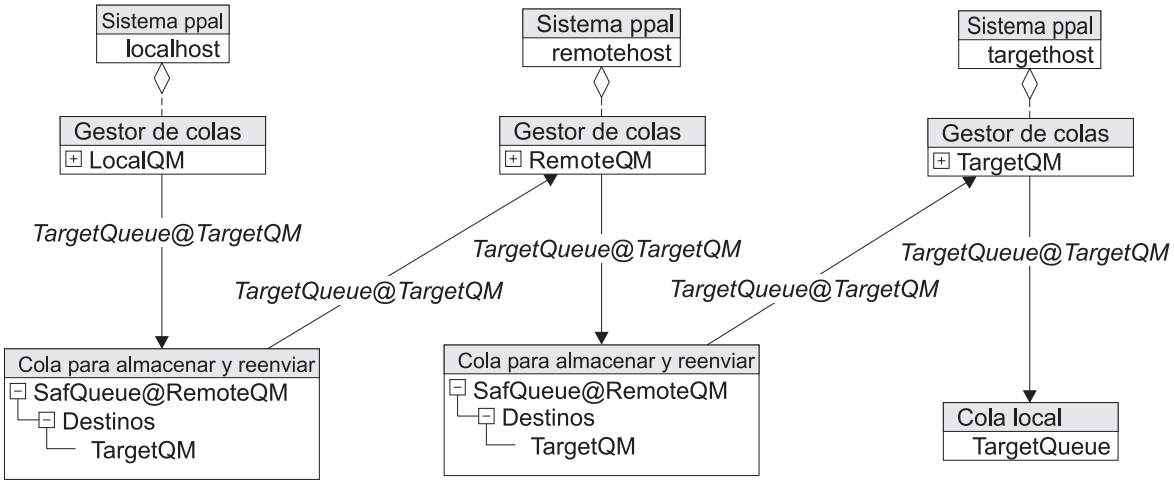

Figura 33. Transporte de mensajes a través de una cola S&F intermedia

Esto funciona porque la combinación de resolución de colas y resolución de conexiones en LocalQM hace que el mensaje se transfiera a la cola S&F de RemoteQM, que, a continuación, se puede mover a su destino. La cadena de colas para almacenar y reenviar puede ser larga de forma arbitraria y cada gestor de colas de la cadena sólo tiene que saber cuál es el siguiente gestor de colas de la cadena. Las rutas de mensajes expresan esto de forma muy sucinta, como se muestra en el diagrama siguiente:

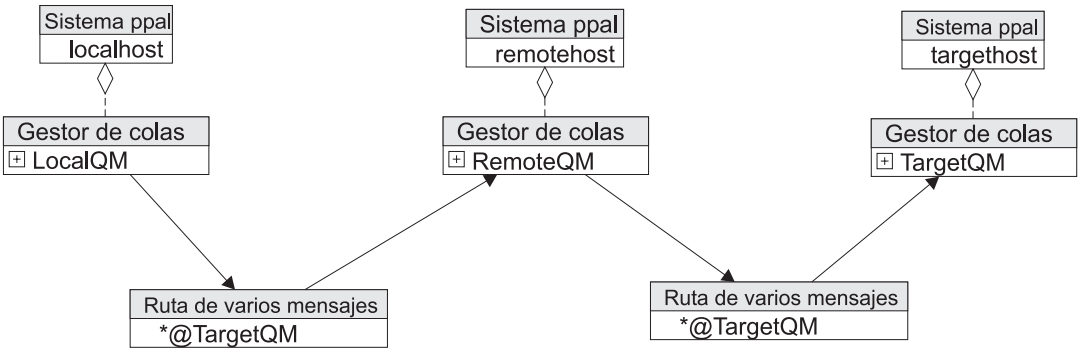

Figura 34. Cadena de colas para almacenar y reenviar

Colas del servidor local

Las colas de servidor local extraen mensajes de las colas para almacenar y reenviar. La cola S&F puede ser una cola S&F de 'empuje', es decir, puede tener una definición de conexión válida. Las colas de servidor local sólo extraen mensajes en un sólo 'salto', es decir, de un gestor de colas remoto con el que está conectado directamente, y sólo extraen mensajes cuyo destino es el gestor de colas local: el gestor de colas en el que reside la cola del servidor local. A continuación se muestra una configuración típica de servidor local:

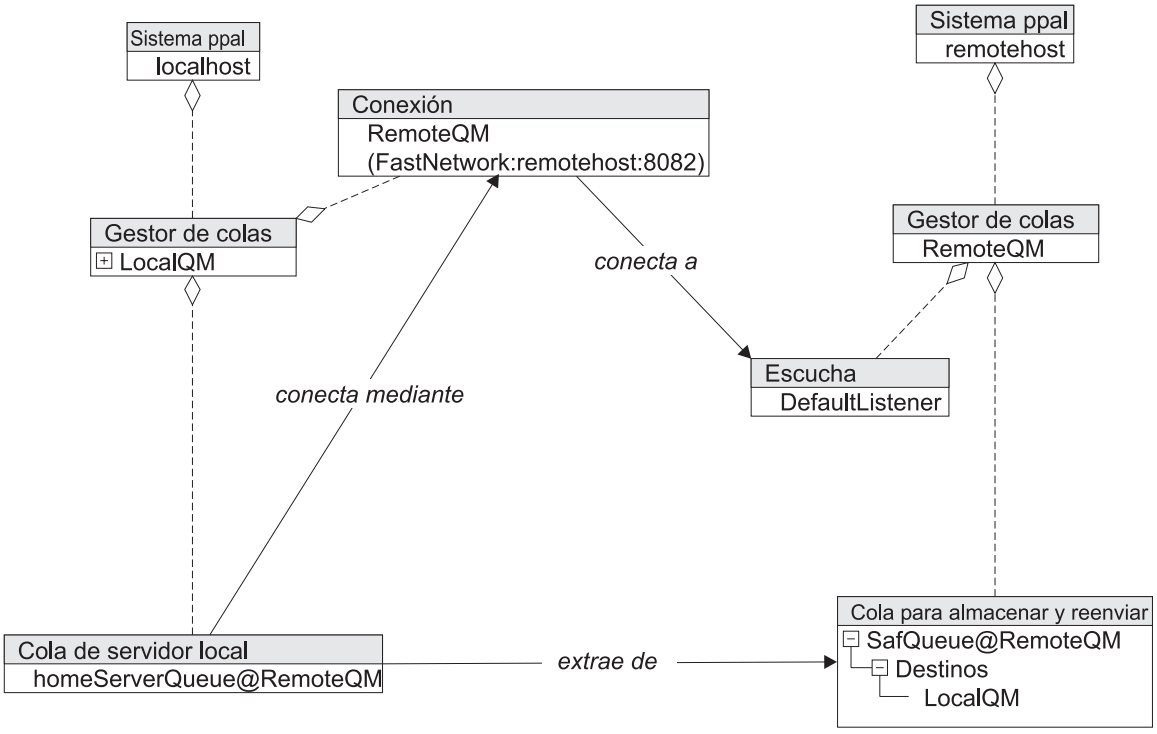

Figura 35. Configuración de cola de servidor local

En el diagrama se muestra una configuración de HomeServerQueue sencilla. En esta configuración, el gestor de colas de servidor no tienen ninguna definición de conexión con el cliente; en cambio, tiene una cola para almacenar, es decir, una cola para almacenar y reenviar sin gestor de colas de destino, que recopila todos los mensajes vinculados para el cliente. Esta recopilación de mensajes abarca todos los destinos de cola del cliente.

El cliente extrae los mensajes de la cola para almacenar mediante una cola de servidor local que apunta a la cola para almacenar del cliente. La cola de servidor local nunca almacena mensajes: los recopila en la cola para almacenar y los entrega a sus destinos en el cliente. El cliente realiza la petición de conexión al servidor mediante su definición de conexión.

La cola del servidor local 'homeServerQueue@RemoteQM' intenta extraer mensajes del gestor de colas 'RemoteQM'. Exige una definición de conexión para poder hacerlo. La cola de servidor local sólo puede extraer mensajes si una cola para almacenar y reenviar está almacenando mensajes para LocalQM.

Los mensajes que se extraen de RemoteQM se 'empujan' a las colas locales de LocalQM. Esto se muestra en el diagrama siguiente, en el que una cola de servidor local de LocalQM está extrayendo mensajes (para LocalQM) de RemoteQM. En este caso, se muestra cómo se está extrayendo un mensaje para TargetQueue@LocalQM y se ha ocultado la resolución en el gestor de colas para que resulte más claro. En realidad, la cola del servidor local presenta cada mensaje extraído al gestor de colas local para su resolución, como se muestra en el diagrama siguiente:

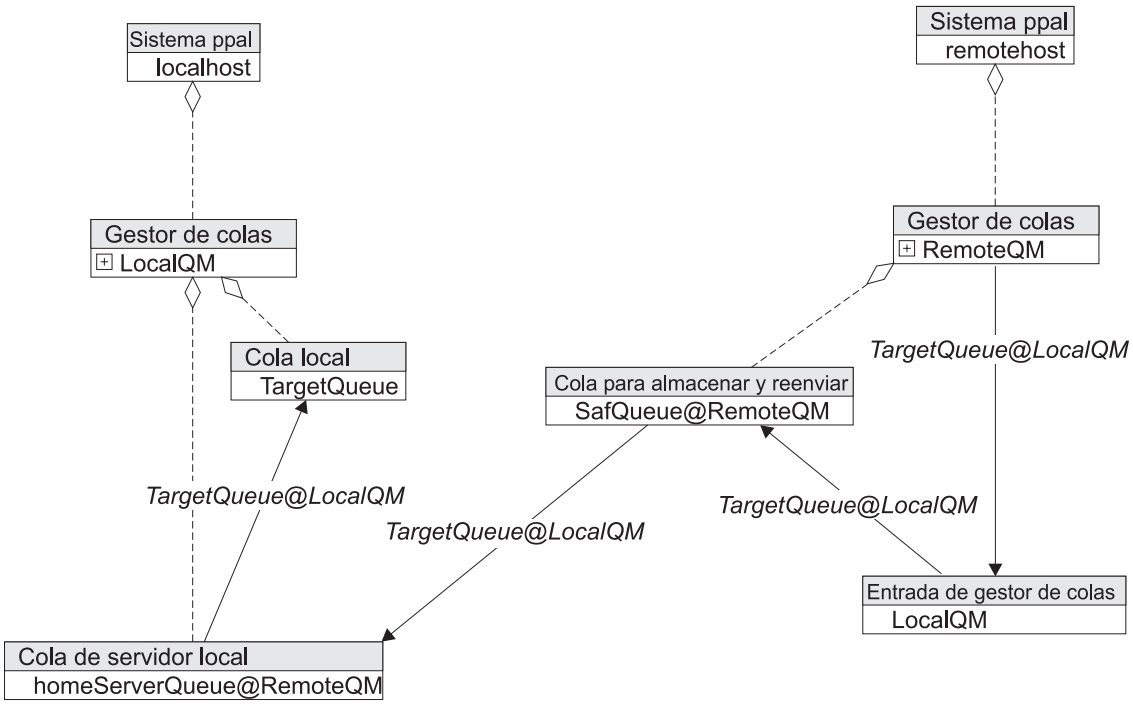

Figura 36. Cola de servidor local que extrae mensajes

La ruta de extracción de mensajes se puede ver en un nivel más abstracto, como se muestra en el diagrama siguiente:

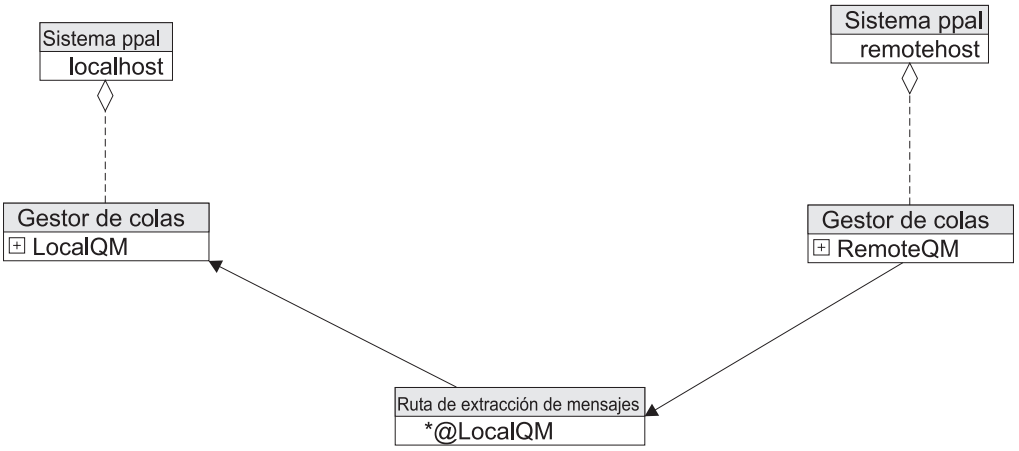

Figura 37. Ruta de extracción de mensajes

¿Por qué son útiles los mensajes extraídos y dónde se pueden utilizar? La función más importante de una ruta de mensaje extraído es que el flujo de mensajes está bajo el control del gestor de colas local. Esto resulta muy útil para un cliente que esté casi todo el tiempo desconectado. Si tuviera que confiar en el servidor que empuja el mensaje, el servidor tendría que sondear continuamente al cliente para comprobar si está disponible. Esto no sería una buena solución en el caso de que hubiera muchos clientes, ya que la mayor parte del tiempo los servidores se dedicarían al sondeo de clientes desconectados.

En cambio, con una cola de servidor local, cada cliente extrae los mensajes cuando está conectado y el servidor sólo tiene que encargarse de las peticiones reales de los clientes conectados. Un ejemplo concreto de esto es la administración de gestores de colas que no tienen funciones de escucha. Los mensajes de administración del cliente se colocan en una cola para almacenar y reenviar. El cliente podrá utilizar, pues, la cola del servidor local para extraer estos mensajes cuando esté conectado. Los mensajes de respuesta de administración se podrán empujar, pues, con la cola remota de empuje normal, como se muestra en el diagrama siguiente:

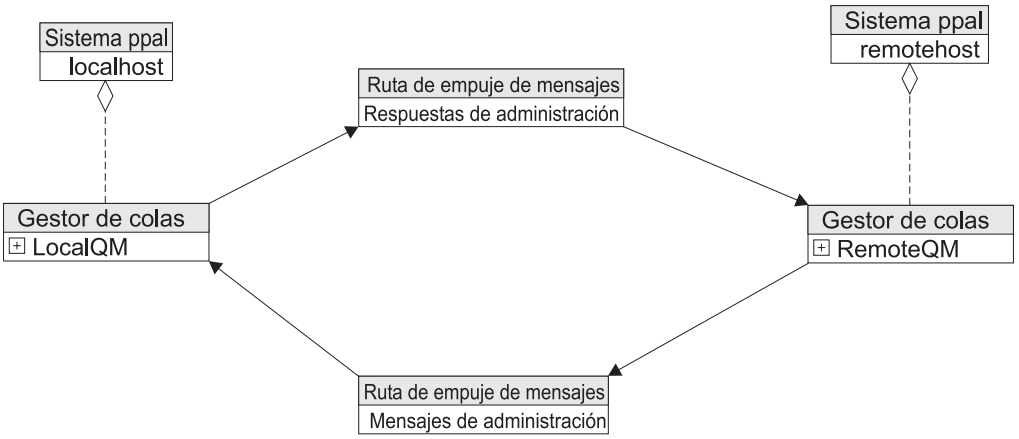

Figura 38. Administración de gestores de colas que no tienen funciones de escucha

Conexiones de tipo vía

Las conexiones de tipo vía permiten direccionar mensajes a través de un gestor de colas intermedio. Por ejemplo, puede que desee que los mensajes de LocalQM vayan a TargetQM a través de RemoteQM. También puede hacerlo con colas para almacenar y reenviar de 'empuje', pero las conexiones de tipo vía proporcionan otro mecanismo, como se muestra en el diagrama siguiente:

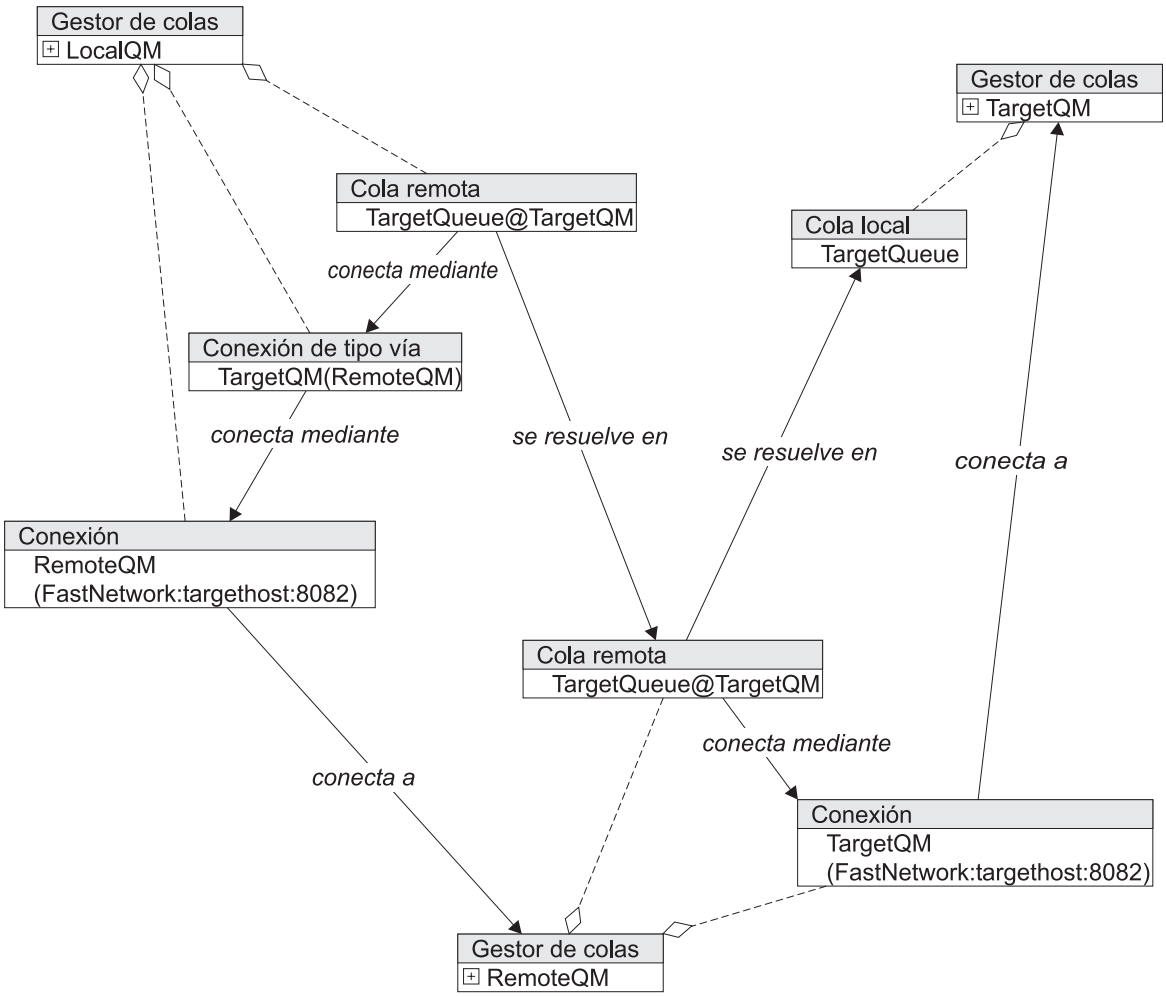

Figura 39. Conexiones de tipo vía

El diagrama anterior muestra los componentes que se utilizan. La definición de conexión denominada 'TargetQM' de LocalQM no contiene la dirección de TargetQM, sino que simplemente hace referencia a la definición de conexión denominada 'RemoteQM'. Esto significa que los mensajes destinados a TargetQM se enviarán a RemoteQM y RemoteQM podrá mover a partir de ahí los mensajes. En el diagrama anterior, RemoteQM tiene la conexión necesaria para mover el mensaje a TargetQM.

El mensaje fluye como se esperaba, como se muestra en el diagrama siguiente:

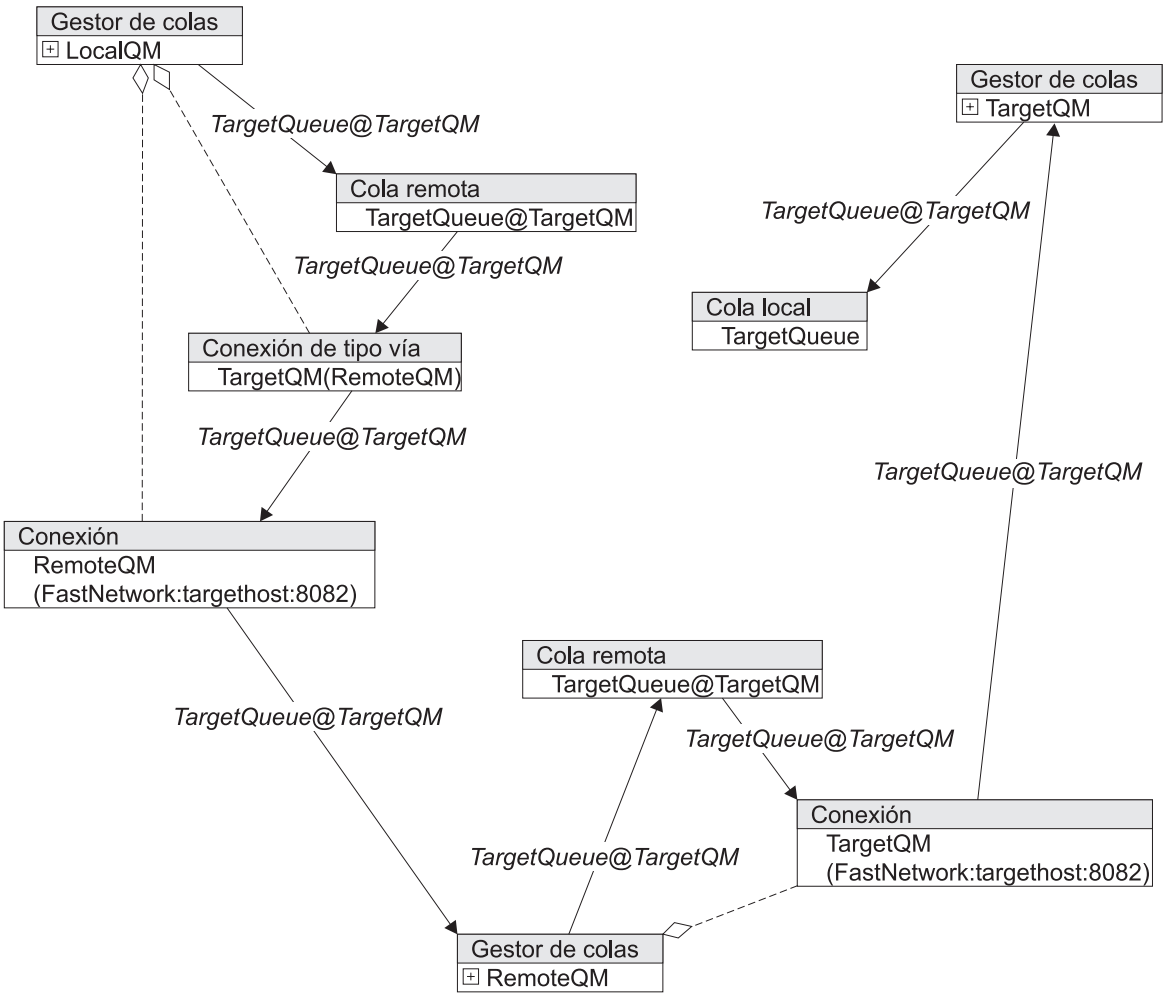

Figura 40. Flujo de mensajes con una conexión de tipo vía

La cola remota en LocalQM utiliza la resolución de conexiones para buscar la conexión de tipo vía. Con ello se pasa el mensaje, a continuación, a la conexión real que mueve el mensaje a RemoteQM. En RemoteQM, la resolución de colas continúa como en el caso sencillo.

Puede ver la topología con más claridad utilizando las rutas de mensajes, como se muestra en el diagrama siguiente:

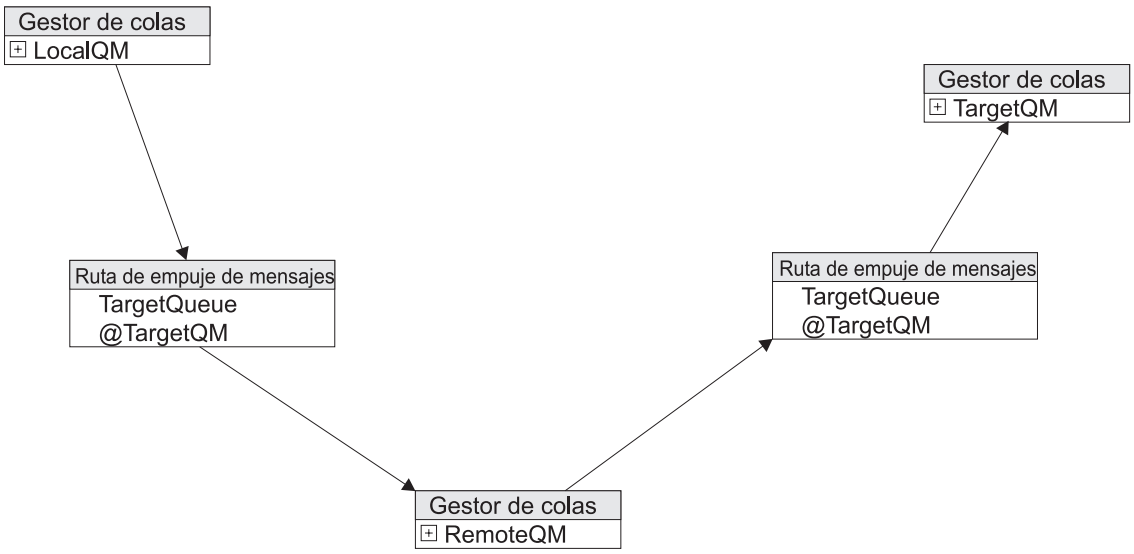

Figura 41. Conexiones de tipo vía expresadas con esquema de rutas de mensajes

Esto se denomina 'encadenamiento de colas remotas'. La cola remota central puede ser síncrona, asíncrona o incluso una cola para almacenar y reenviar.

Redireccionamiento con alias de gestor de colas

La sustitución por anomalía es una situación común que ilustra el importante papel que juegan los alias de gestor de colas en el direccionamiento.

En los ejemplos siguientes, puede ver cómo un cliente se comunica con un servidor y dispone de un servidor de reserva que se puede utilizar si el servidor principal falla o se apaga para realizar tareas de mantenimiento:

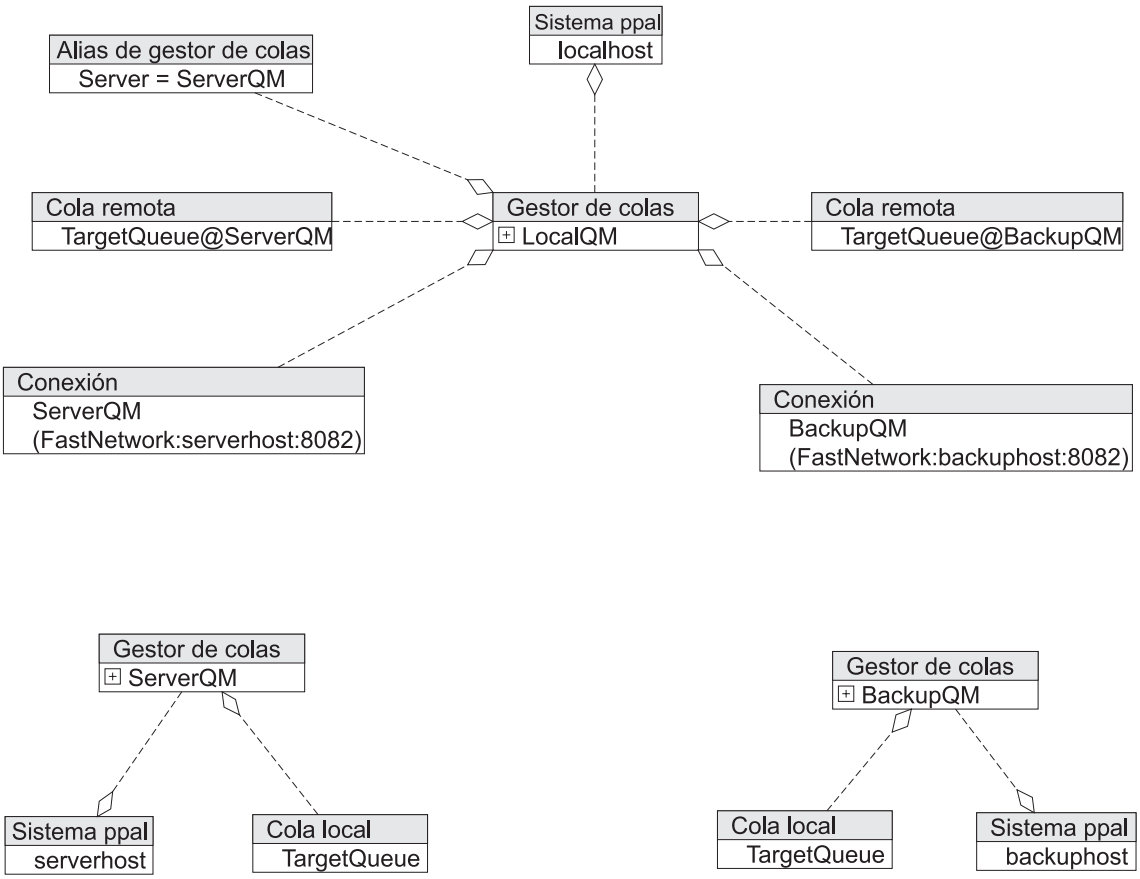

Figura 42. Alias de gestor de colas y sustitución por anomalía

En el diagrama anterior se muestra el gestor de colas de cliente local, con una conexión con ServerQM y una definición de cola remota para TargetQueue@ServerQM. El servidor (en la parte inferior izquierda) tiene una cola local como destino para el mensaje de ejemplo y el servidor de reserva mimetiza esto (parte inferior derecha). Además, en el gestor de colas del cliente, existe un alias de gestor de colas que correlaciona el servidor de nombres con ServerQM. Esta correlación se utiliza para los mensajes transferidos al servidor. A continuación se muestra la resolución de mensajes para la configuración de funcionamiento normal donde un mensaje transferido a TargetQueue@Server se dirige a TargetQueue@ServerQM:

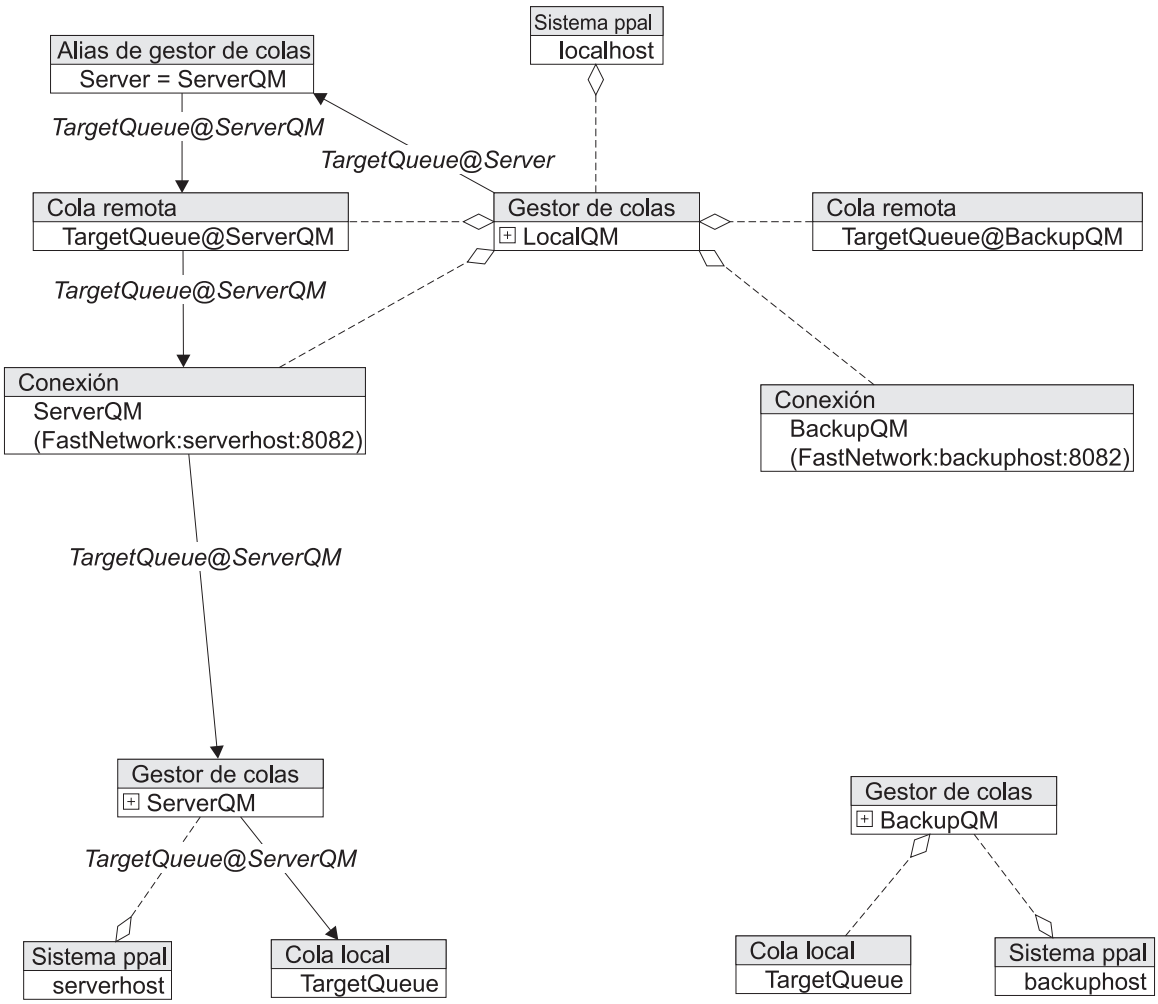

Figura 43. Direccionamiento de tráfico con un alias de "servidor"

El alias correlaciona los mensajes para el servidor con ServerQM y este selecciona la definición de cola remota TargetQueue@ServerQM. Si el administrador de red tiene que direccionar tráfico al servidor de copia de seguridad, sólo se tiene que cambiar el alias de gestor de colas (de hecho se suprime y se vuelve a crear con otro nombre de destino, en este caso BackupQM):

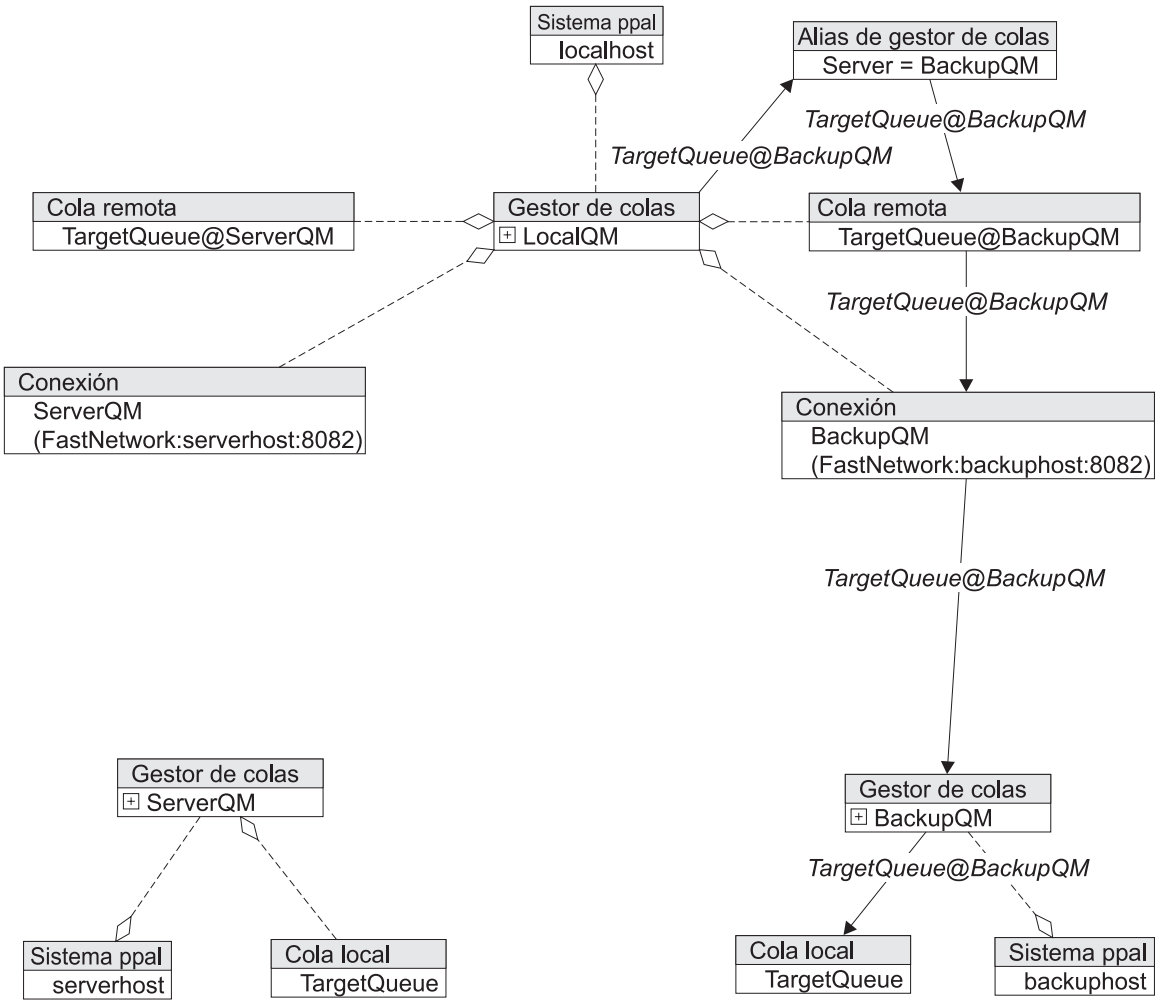

Figura 44. Direccionamiento de tráfico al servidor de reserva mediante un alias de "servidor"

El cambio de alias redirecciona el mensaje a otra cola remota y, a partir de ahí, al gestor de colas de reserva y a TargetQueue@BackupQM. Hay un par de rutas de mensajes, una a cada servidor, y un alias de gestor de colas para elegir entre las rutas de mensajes, como se muestra en el diagrama siguiente:

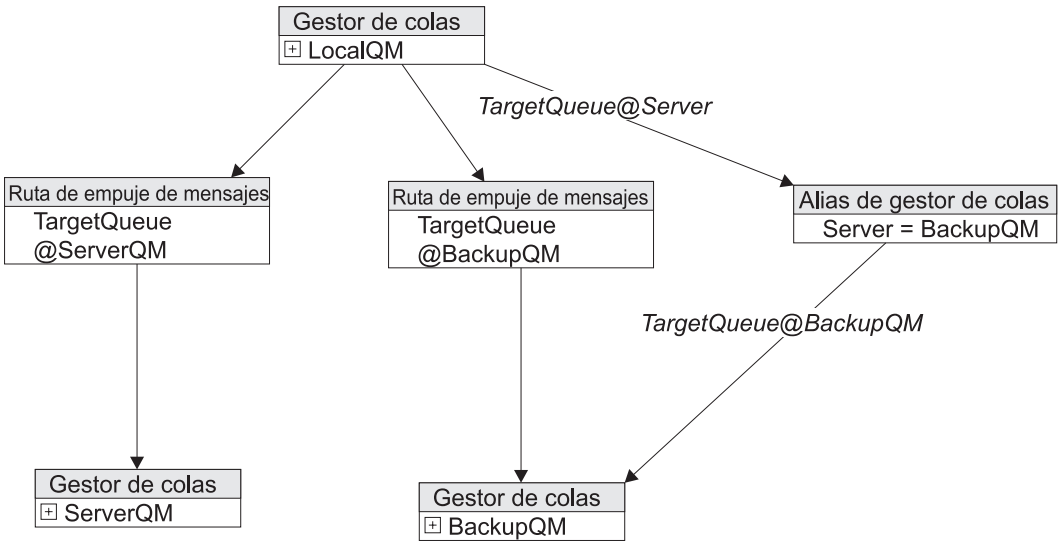

Figura 45. Elección entre rutas de mensajes

En el ejemplo anterior se exigía un cambio en cada cliente de un sistema que necesite un redireccionamiento a un servidor de reserva. Si existe un gran número de clientes, esto puede que no resulte práctico. Además, cada cliente exige dos definiciones de ruta de mensajes completas (una cola remota y una definición de conexión para cada uno). Puede evitar que sea necesario cambiar el cliente preparando un segundo servidor para que esté a la escucha en la misma dirección y puerto que el primero. Cuando el administrador desee cambiar, el primero se puede desactivar y se puede cambiar al segundo. En esta situación, puede resultar útil conservar los nombres de los distintos servidores. Se puede asignar al servidor de reserva un alias de gestor de colas que correlacione BackupQM con ServerQM. Esto permite que BackupQM se haga con el puesto de ServerQM.

Resolución de mensajes de puente MQe-MQ

Una conexión entre gestores de colas de MQe y MQ implica una recopilación de objetos. En el diagrama siguiente sólo se muestran las entidades que forman el enlace de comunicaciones entre los dos gestores de colas:

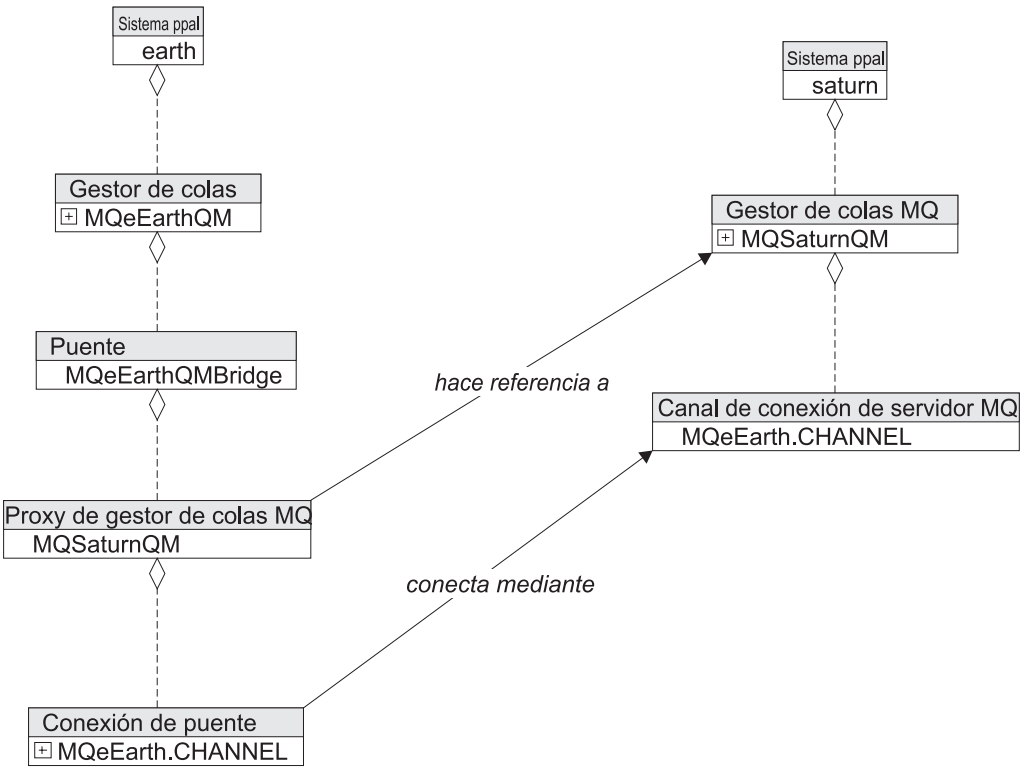

Figura 46. Conexión de gestores de colas de MQe y MQ

Las entidades importantes son:

- (Puente)MQeEarthQMBridge: recurso de puente propiedad y bajo el control del gestor de colas MQeEarthQM.
- (Proxy de gestor de colas de MQ)MQSaturnQM: describe MQSaturnQM y cómo conectarse a él.
- (Conexión de puente)MQeEarth.CHANNEL: vía de acceso de comunicaciones entre MQeEarthQM y MQSaturnQM.
- (Canal de conexión de servidor de MQ)MQeEarth.CHANNEL: canal de servidor de MQ estándar que proporciona un punto de entrada a MQSaturnQM para MQeEarthQM.

Estas entidades se describen más detalladamente en otras partes de esta documentación. Estas entidades se utilizan en los ejemplos siguientes de conectividad de puente, pero no se muestran en los diagramas.

Extracción de mensajes de MQ

Al configurar una cola de transmisión en MQ y un escucha puente en un gestor de colas de MQe, puede habilitar el gestor de colas para extraer mensajes de la cola de transmisión. Aunque teóricamente con esto basta para extraer mensajes de la cola de transmisión, no se pueden colocar mensajes en la cola de transmisión sin crear colas adicionales en un gestor de colas de MQ.

Ruta de extracción única:

Para permitir que los mensajes se direccionen correctamente, puede crear colas adicionales en un gestor de colas de MQ. La manera más sencilla es crear una cola remota en MQ para permitir que el gestor de colas de MQ acepte los mensajes dirigidos a TargetQueue@MQeEarthQM, como se muestra en el diagrama siguiente:

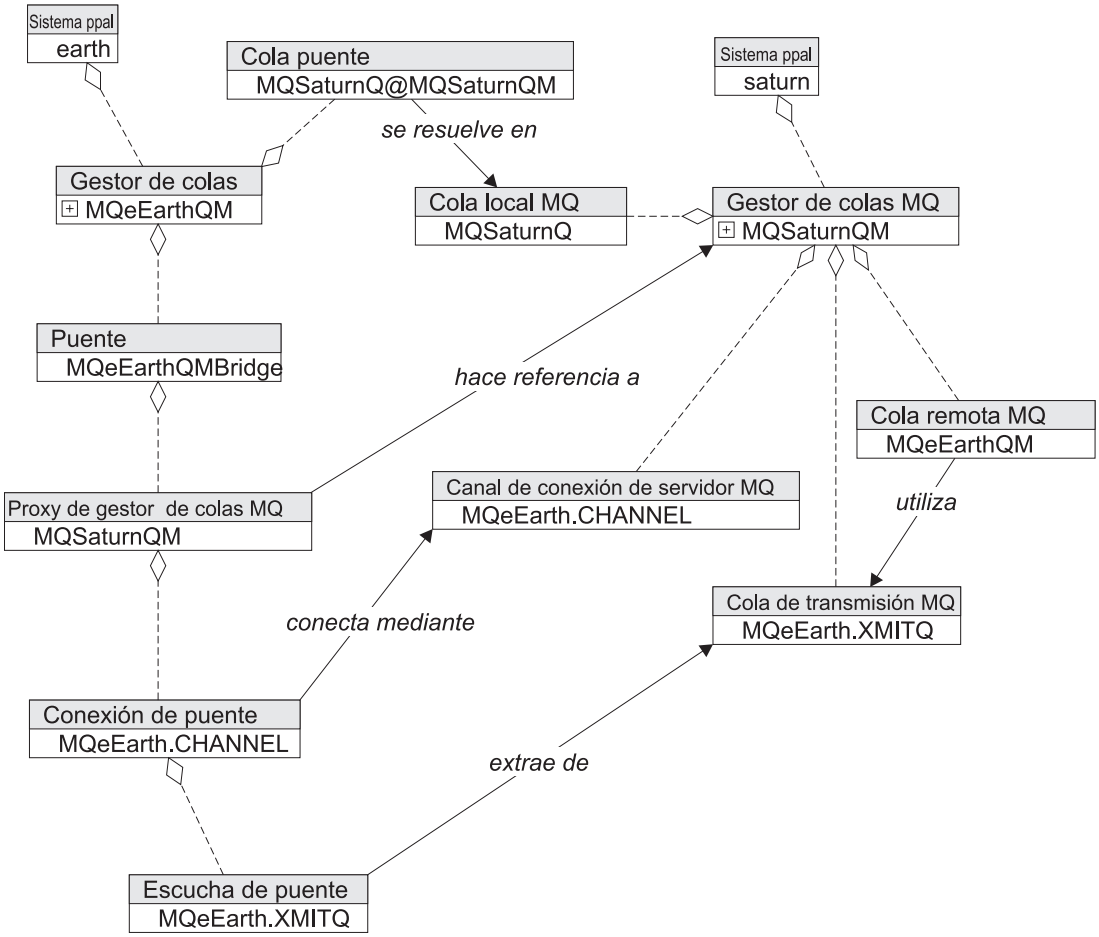

Figura 47. Creación de una cola remota en MQ

Los mensajes dirigidos a TargetQueue@MQeEarthQM se colocan en la cola de transmisión de MQ. El escucha puente los extrae, a continuación, de la cola de transmisión y los presenta al gestor de colas de MQe. Finalmente, tiene lugar la resolución de mensajes, como se muestra en el diagrama siguiente:

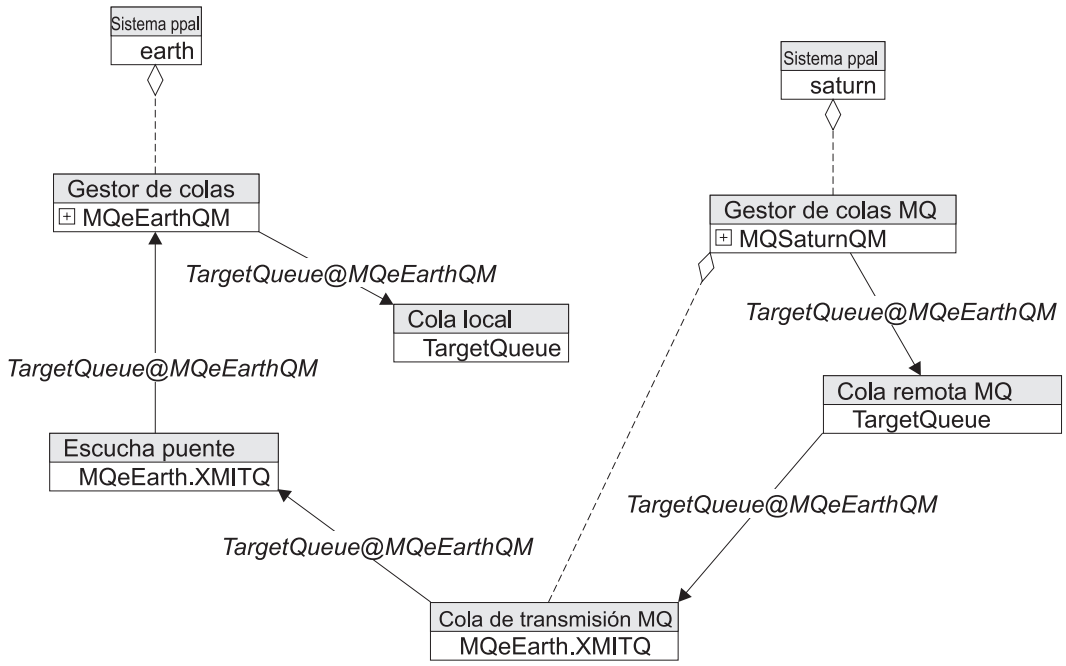

Figura 48. Escucha puente que extrae mensajes de una cola de transmisión de MQe

Se trata efectivamente de una ruta de extracción de mensajes única.

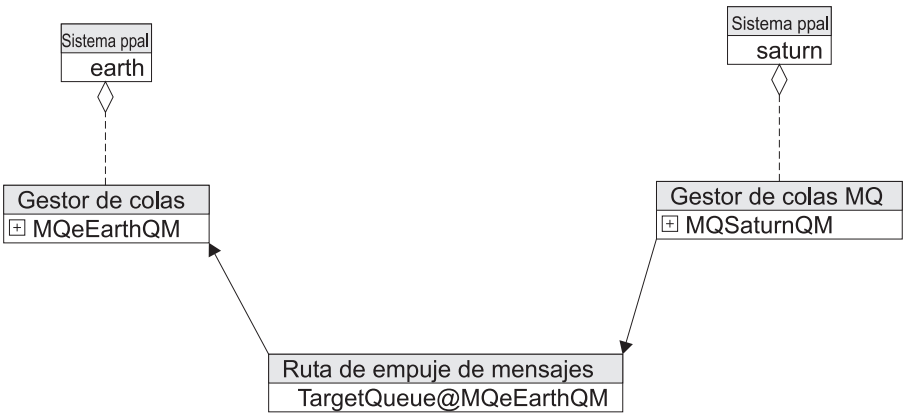

Figura 49. Ruta de extracción de mensajes única

Ruta de extracción múltiple:

Por lo general, es más útil usar una ruta de extracción de mensajes múltiple, ya que con ello se exige el mismo número de definiciones de recursos, pero se manejará todo el tráfico para el gestor de colas de MQe. Para ello se utiliza un alias de gestor de colas remoto en MQ (se trata efectivamente de una cola remota en la que el nombre de la cola de destino es el mismo que el nombre del gestor de colas de destino), como se muestra en el diagrama siguiente:

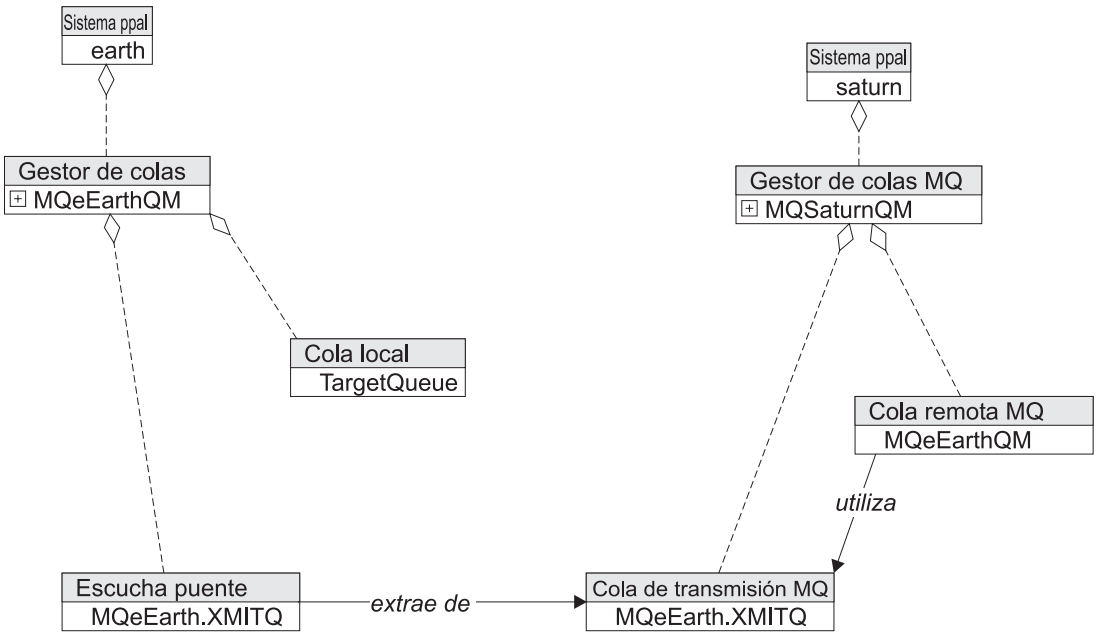

Figura 50. Ruta de extracción de mensajes múltiple

La resolución de mensajes funciona como antes, pero ahora los mensajes para la cola en MQeEarthQM se moverán, lo que convertirá este proceso en una ruta de extracción de mensajes múltiple, como se muestra en el diagrama siguiente:

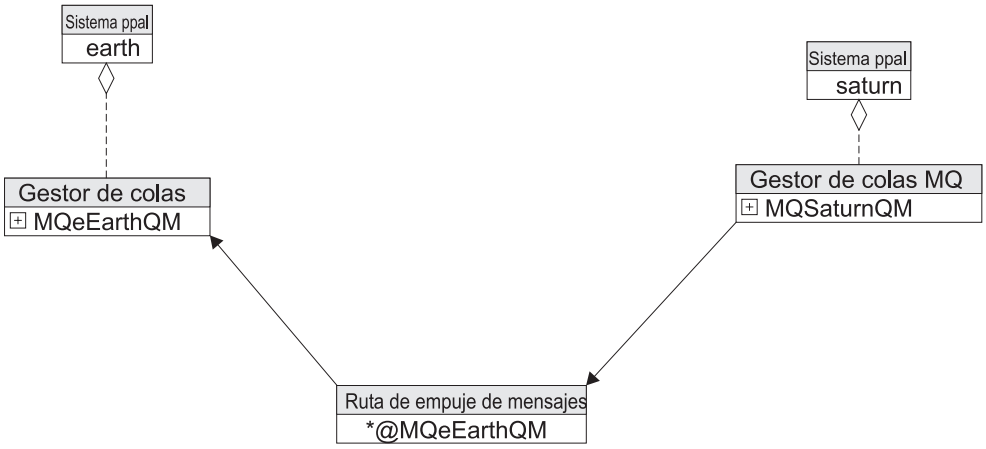

Figura 51. Ruta de extracción múltiple, expresada con un esquema de ruta de mensajes

Empuje de mensajes a MQ

El empuje de mensajes a MQ es bastante directo. Tiene que presuponer de nuevo que están presentes los componentes que se describen en “Resolución de mensajes de puente MQe-MQ” en la página 86, pero ahora tiene que crear una cola puente que es una cola remota de MQe que hace referencia a una cola de un gestor de colas de MQ, como se muestra en el diagrama siguiente:

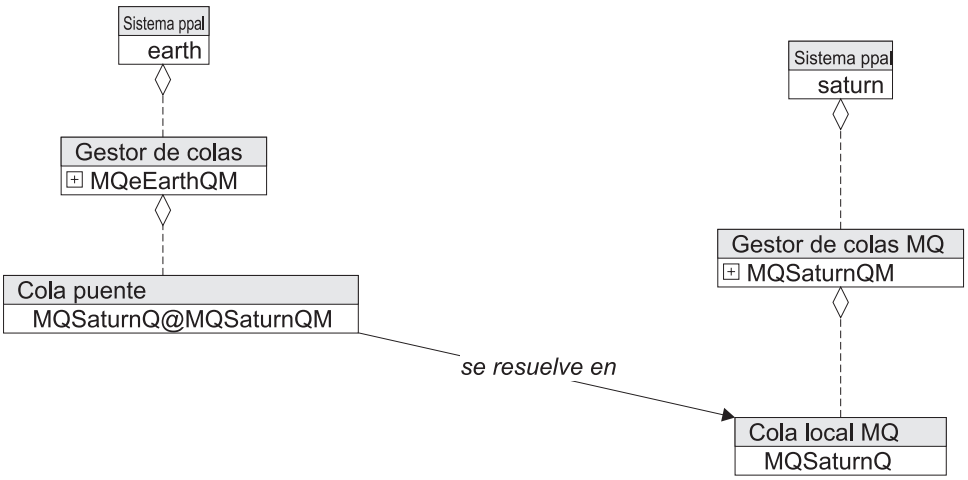

Figura 52. Empuje de mensajes a MQ

Los mensajes se desplazan como se esperaba por esa definición de cola remota, como se muestra a continuación:

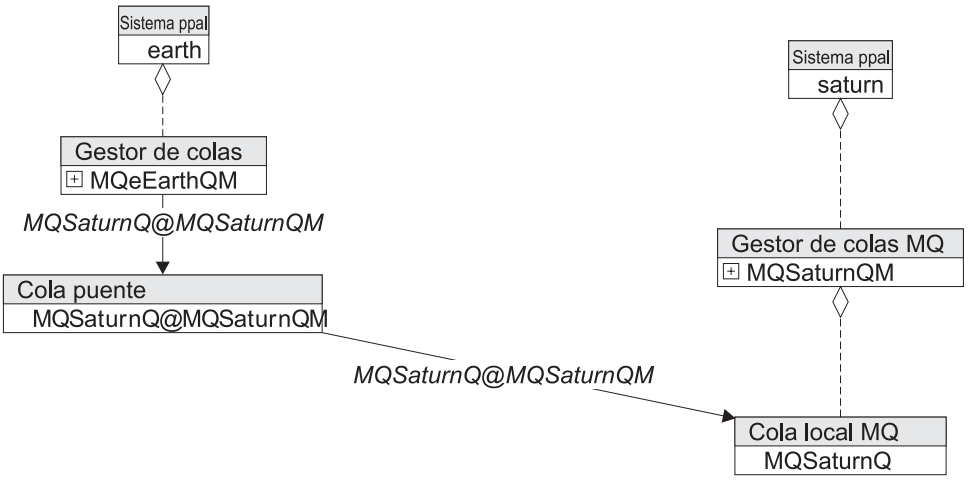

Figura 53. Mensajes que se desplazan por una definición de cola remota

Es exactamente lo mismo que una ruta de empuje de mensajes simple entre dos gestores de colas, como se muestra a continuación:

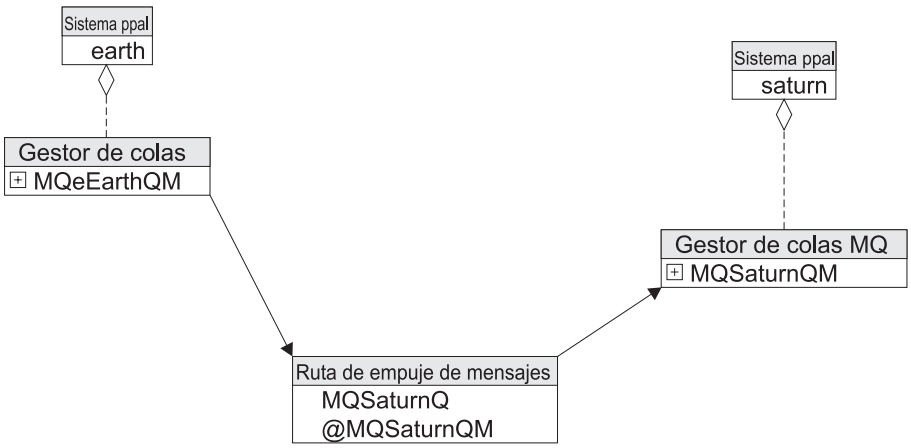

Figura 54. Visión simplificada de una ruta de empuje de mensajes a MQ

Conexión de un cliente a MQ a través de un puente

Una topología común es permitir que los mensajes fluyan entre MQ y un gestor de colas de MQe de cliente. Esto no puede tener lugar de forma directa, sino que exige un gestor de colas de MQe intermedio habilitado para puentes. El cliente puede ser un dispositivo de pequeño tamaño que no reconozca a MQ. Es necesario realizar adiciones para que un cliente (MQeMoonQM, en un dispositivo denominado moon) pueda comunicarse con MQ, como se muestra en el diagrama siguiente:

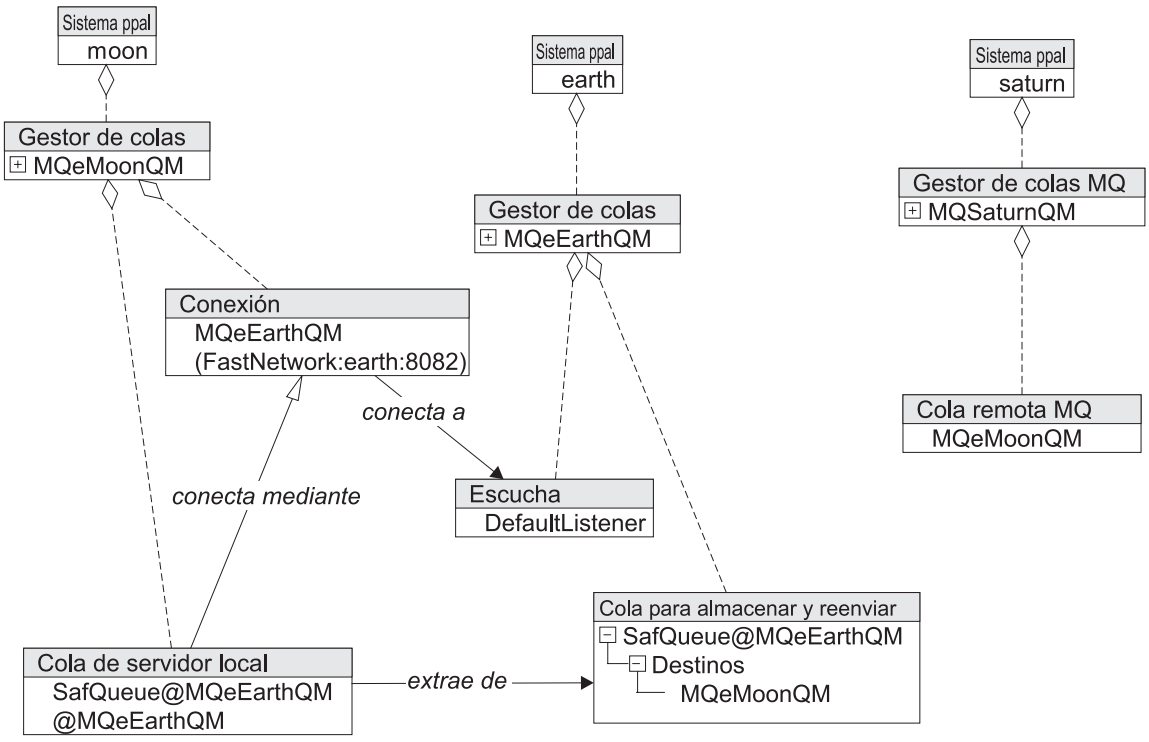

Figura 55. Cliente que se comunica con MQ

Esto añade lo siguiente:

- (Sistema principal)moon
- (Gestor de colas) MQeMoonQM en (Sistema principal)moon

- Definición de conexión de MQeMoonQM con un escucha coincidente en MQeEarthQM para proporcionar la conectividad entre los dos gestores de colas de MQe.
- Una cola para almacenar y reenviar en MQeEarthQM que acepta y mantiene los mensajes para MQeMoonQM y un servidor local en MQeMoonQM que extrae los mensajes de la cola para almacenar y reenviar.
- Una definición de cola remota en el gestor de colas de MQ que direcciona los mensajes para MQeMoonQM a la cola de transmisión MQeEarth.XMITQ. Esto permite que los mensajes para MqeMoonQM se coloquen en la cola de transmisión, desde donde se extraen a MQeEarthQM.

Esta topología se aprecia mejor como rutas de mensajes, como se muestra en el diagrama siguiente:

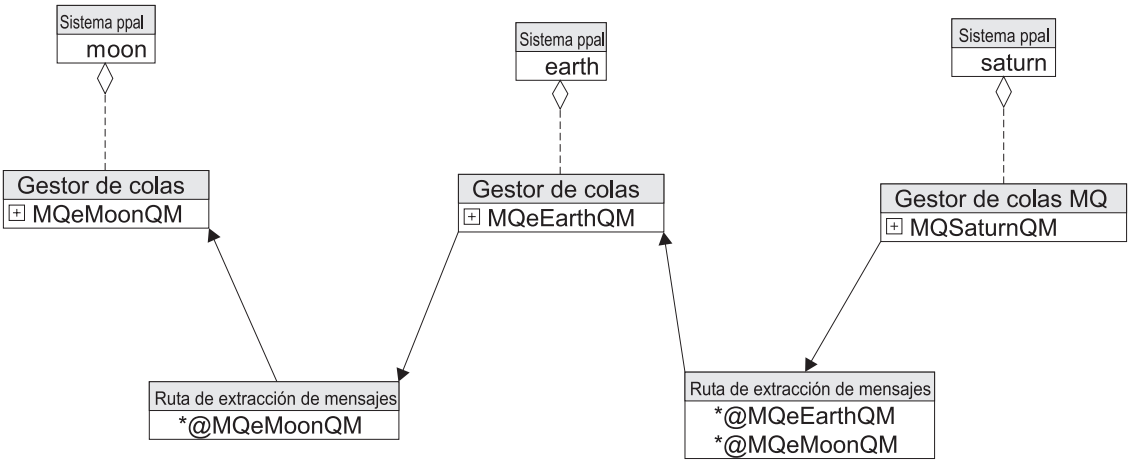

Figura 56. Rutas de extracción simplificadas de MQ a través de una pasarela de MQe a un gestor de colas de estilo de dispositivo de MQe

Los mensajes se pueden empujar a MQ utilizando una conexión de tipo vía para encadenar las colas remotas, como se muestra a continuación:

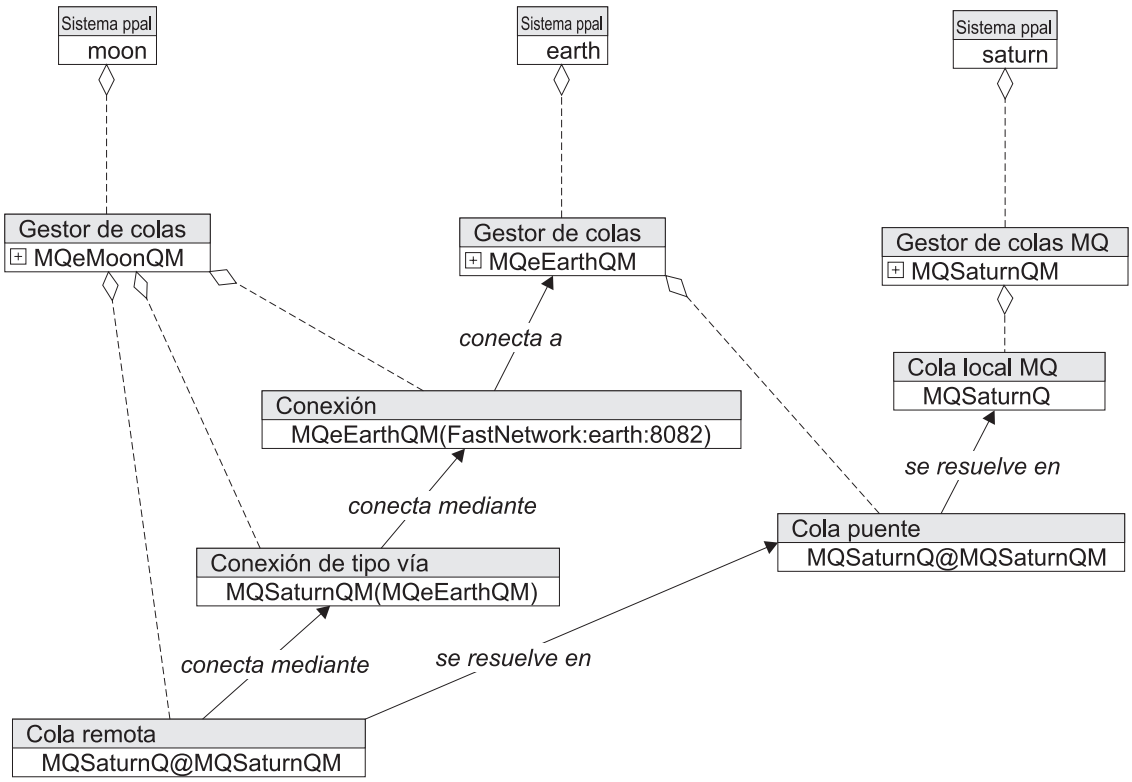

Figura 57. Empuje de mensajes mediante una conexión de tipo vía

Aquí se ha añadido una conexión de tipo vía para direccionar mensajes destinados a MQSaturnQM a través de MQeEarthQM y se ha añadido una definición de cola remota para MQSaturnQ@MQSaturnQM. Los mensajes ahora pueden fluir del cliente a MQ, como se muestra en el diagrama siguiente:

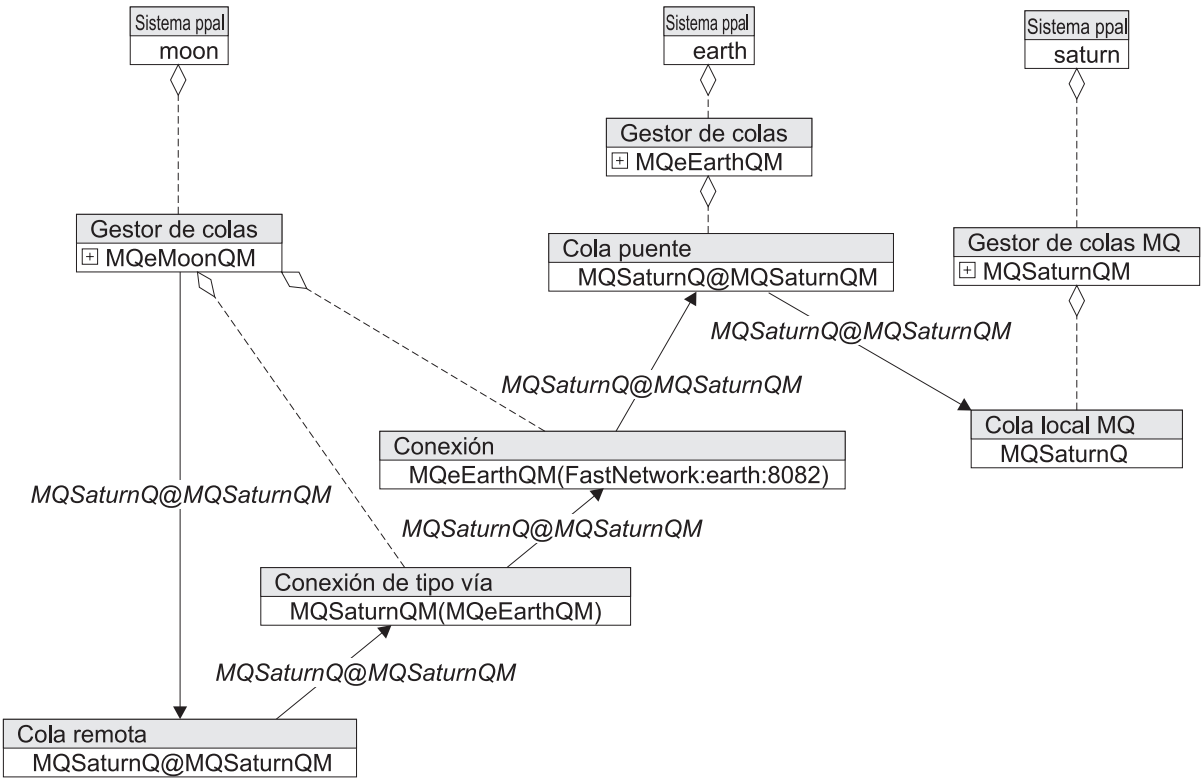

Figura 58. Empuje de mensajes a MQ

Esta topología se entiende mejor como recopilación de rutas de mensajes:

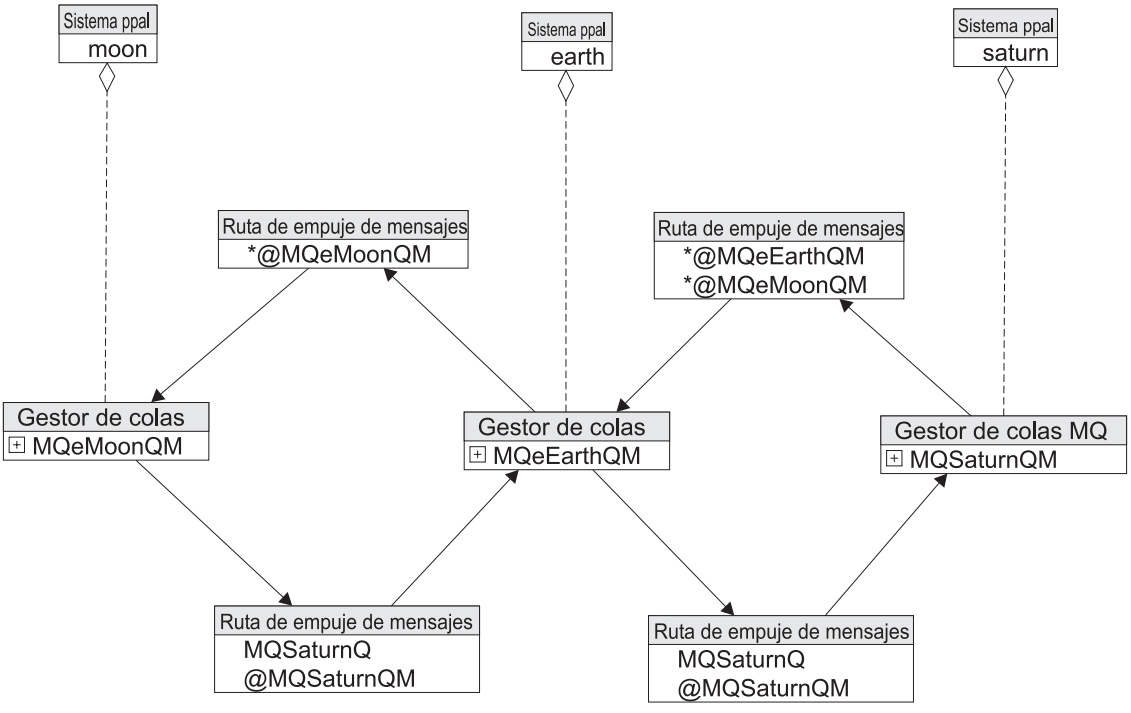

Figura 59. Visión simplificada que muestra las rutas que empujan mensajes de un gestor de colas de estilo de dispositivo de MQe a un gestor de colas de MQ

Empuje de mensajes a MQ con una conexión de tipo vía

Una topología común permite que los mensajes fluyan entre MQ y un gestor de colas de MQe de cliente. Esto no puede tener lugar de forma directa, sino que exige un gestor de colas de MQe intermedio habilitado para puentes. El cliente puede ser un dispositivo de pequeño tamaño que no reconozca a MQ. Si empieza con la configuración anterior, se necesitan las siguientes adiciones para que un cliente (MQeMoonQM, en un dispositivo denominado moon) pueda comunicarse con MQ, como se muestra en el diagrama siguiente:

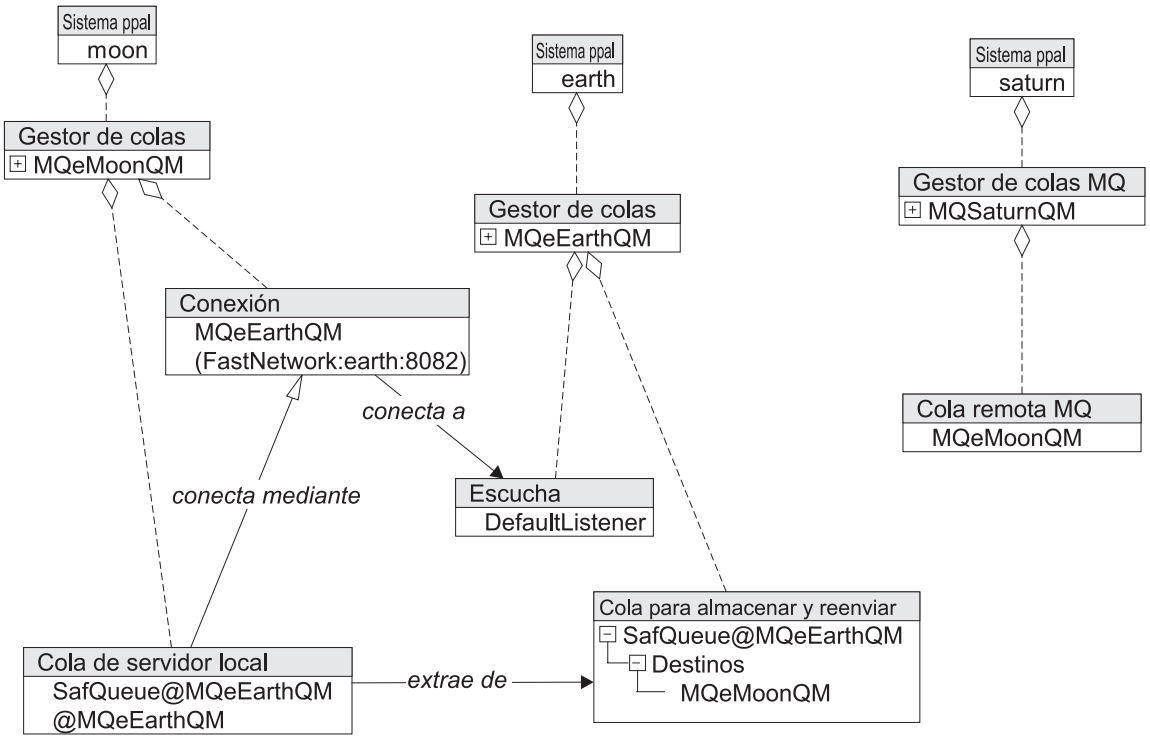

Figura 60. Cliente que se comunica con MQ

Se ha añadido lo siguiente:

- (Sistema principal)moon
- (Gestor de colas) MQeMoonQM en (Sistema principal)moon
- Definición de conexión de MQeMoonQM con un escucha coincidente en MQeEarthQM para proporcionar la conectividad entre los dos gestores de colas de MQe.
- Una cola para almacenar y reenviar en MQeEarthQM que acepta y mantiene los mensajes para MQeMoonQM y un servidor local en MQeMoonQM que extrae los mensajes de la cola para almacenar y reenviar.
- Una definición de cola remota en el gestor de colas de MQ que direcciona los mensajes para MQeMoonQM a la cola de transmisión MQeEarth.XMITQ. Esto permite que los mensajes para MqeMoonQM se coloquen en la cola de transmisión, desde donde se extraen a MQeEarthQM.

Esta topología se aprecia mejor como rutas de mensajes, como se muestra en el diagrama siguiente:

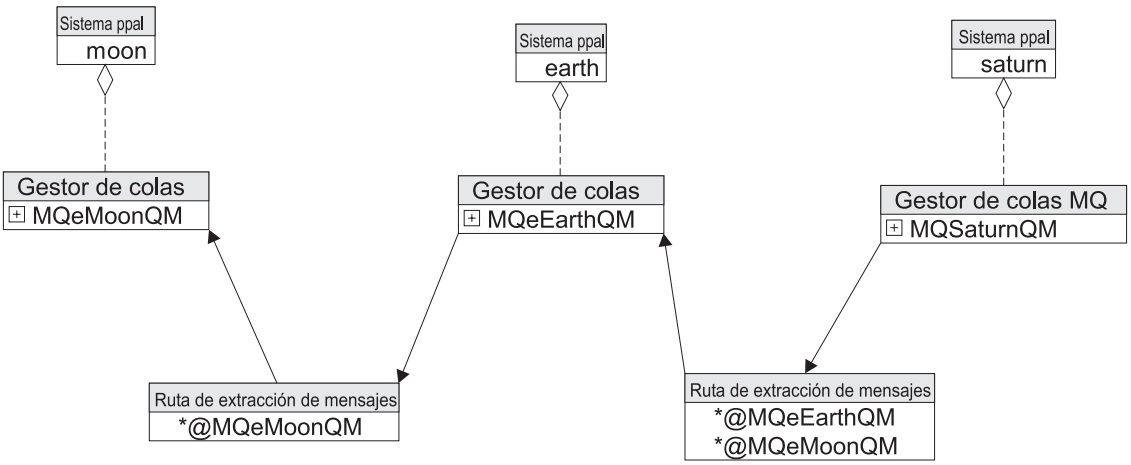

Figura 61. Rutas de extracción simplificadas de MQ a través de una pasarela de MQe a un gestor de colas de estilo de dispositivo de MQe

Los mensajes se pueden empujar a MQ utilizando una conexión de tipo vía para encadenar las colas remotas, como se muestra en el diagrama siguiente:

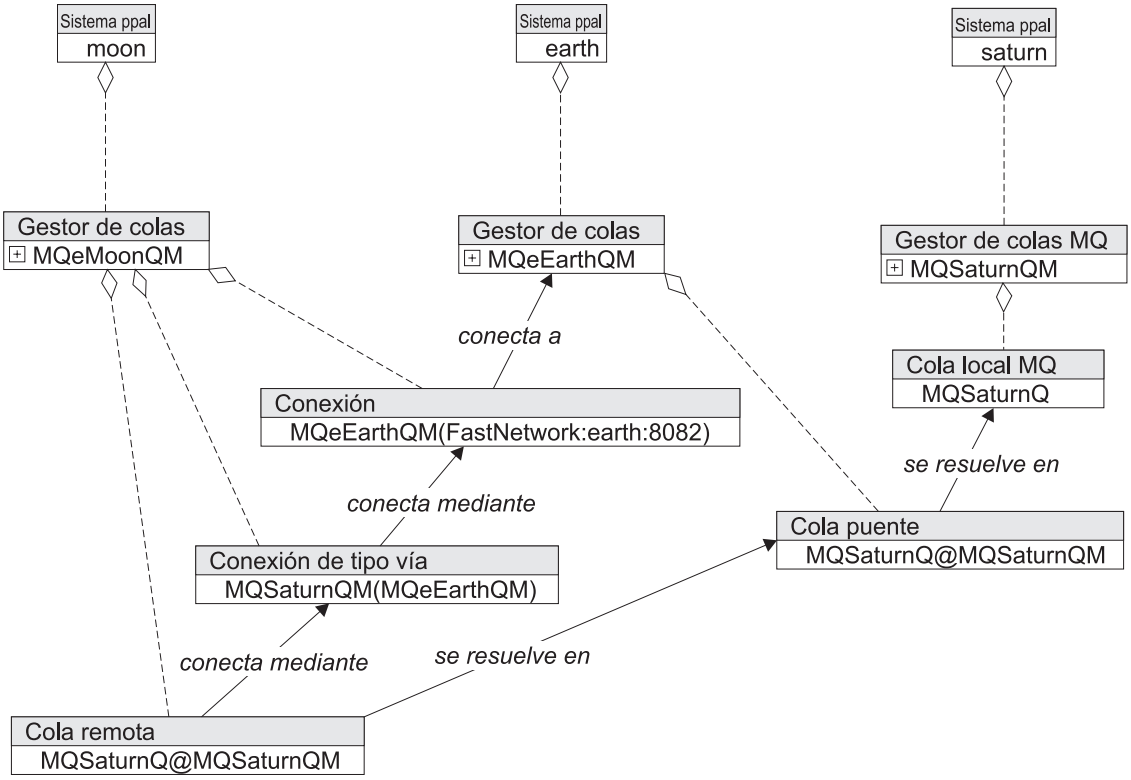

Figura 62. Empuje de mensajes mediante una conexión de tipo vía

Aquí hemos añadido una conexión de tipo vía para direccionar mensajes destinados a MQSaturnQM a través de MQeEarthQM y hemos añadido una definición de cola remota para MQSaturnQ@MQSaturnQM. Los mensajes ahora pueden fluir del cliente a MQ, como se muestra en el diagrama siguiente:

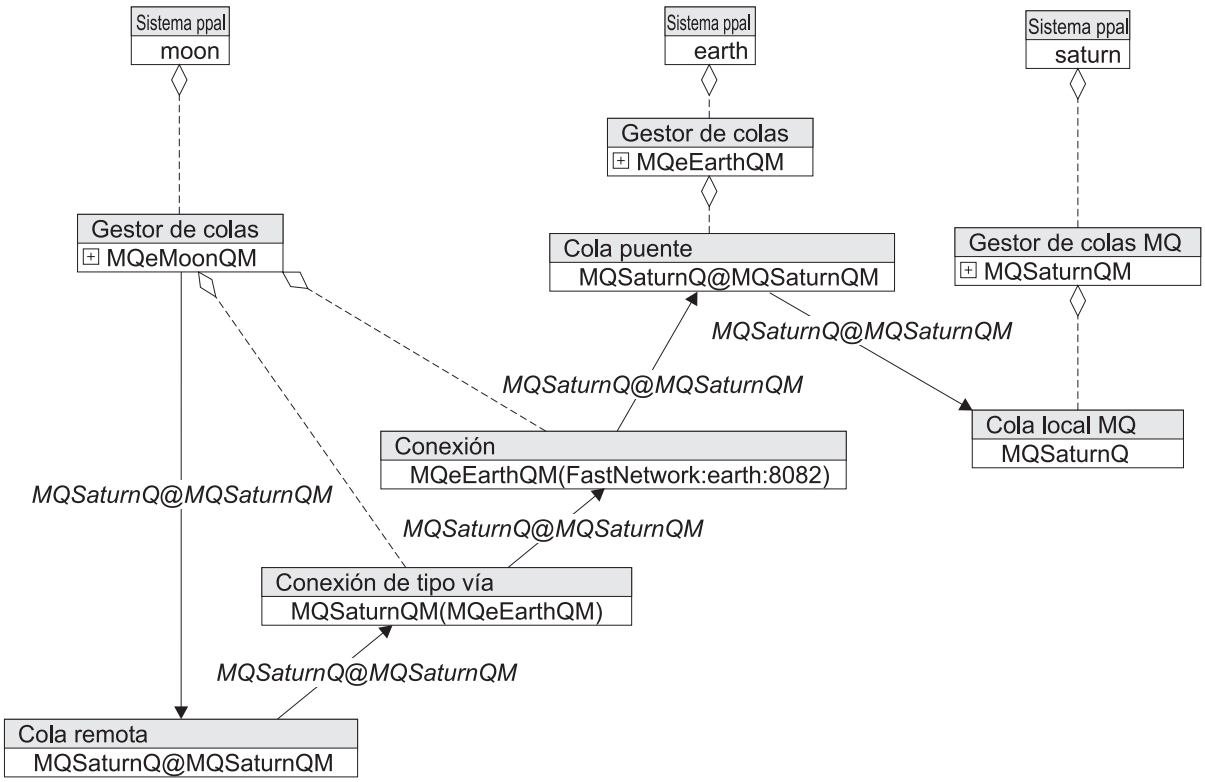

Figura 63. Empuje de mensajes a MQ

Esta topología se entiende mejor como recopilación de rutas de mensajes, como se muestra en el diagrama siguiente:

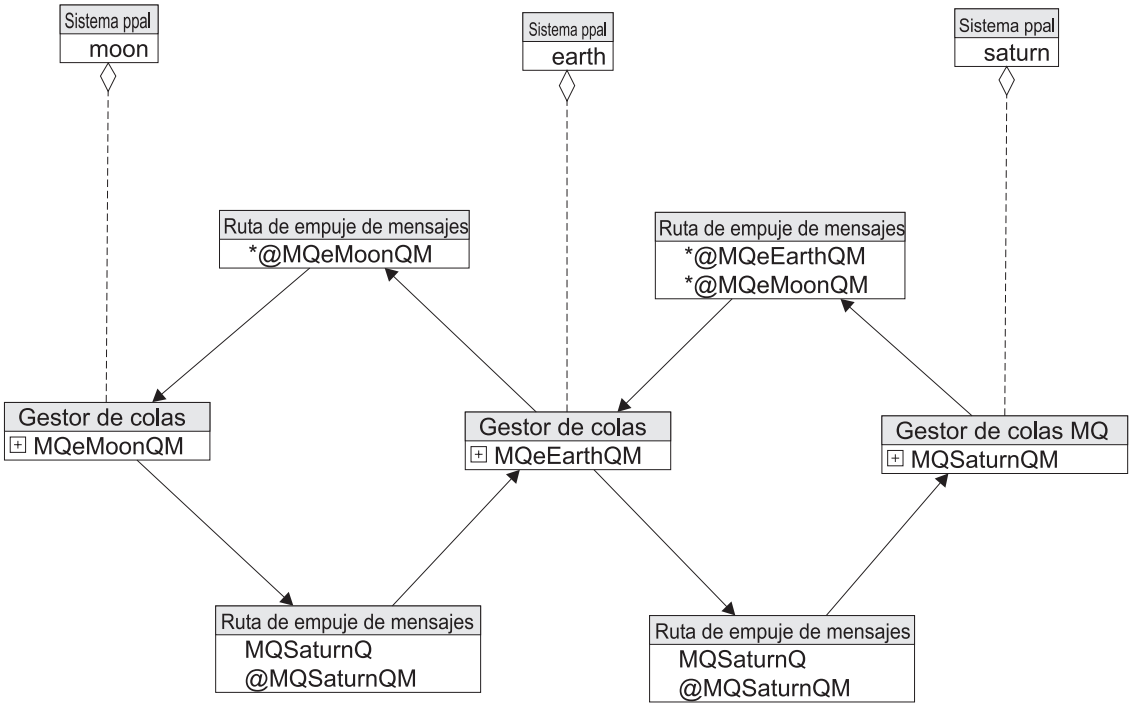

Figura 64. Visión simplificada que muestra las rutas que empujan mensajes de un gestor de colas de estilo de dispositivo de MQe a un gestor de colas de MQ

Consideraciones de seguridad

Las definiciones de colas remotas especifican los requisitos de seguridad que deben cumplir los canales que desplazan mensajes a colas de destino. La norma de atributos de gestor de colas define las normas para actualizar los canales; por lo tanto, con una norma suficientemente flexible, un solo canal puede cumplir varios requisitos de seguridad.

Cuando un mensaje se almacena en una cola, en ruta o en el destino, la norma de atributo de cola determina si la seguridad del canal cumple los requisitos de la cola. Tenga en cuenta, no obstante, que hay transferencias de mensajes que no implican un canal, por ejemplo cuando un servidor local coloca un mensaje que ha recibido de una cola para almacenar en su cola de destino. En estos casos, no se tiene que cumplir ningún requisito de seguridad en la transferencia, pero el mensaje se almacenará en su cola de destino bajo el control de las características de seguridad de esa cola. Cuando el servidor local obtiene el mensaje de la cola para almacenar, un canal se ve implicado (con las características determinadas por la cola del servidor local y que deben ser aceptables para la cola de almacenamiento). No obstante, cuando la cola del servidor de local pasa el mensaje a la cola de destino, no hay ninguna característica de canal que comparar con las características de seguridad de la cola de destino.

En un solo salto (transferencia de mensajes), la comprobación de seguridad está entre los gestores de colas de origen y de destino. En varios saltos (transferencias de mensajes asíncronas), la comprobación de seguridad tiene lugar paso a paso en cada salto.

Normas de resolución

Las normas de resolución siempre empiezan con un mensaje que se presenta a un gestor de colas, con un nombre de gestor de colas de destino especificado y un nombre de cola de destino especificado. Esto es equivalente a la invocación de API putMessage(queueManagerName, queueName, msg,....). destinationQueueManagerName y destinationQueueName deben identificar una cola local en la que se deba colocar finalmente el mensaje.

Norma 1: resolver alias de gestor de colas

Si el gestor de colas tiene una correlación de alias de destinationQueueManagerName con otro nombre, como realQueueManagerName, esta sustitución se realiza en primer lugar y la invocación de:

```
putMessage(destinationQueueManagerName, destinationQueueName
```

se transforma efectivamente en

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

A partir de ese momento destinationQueueManagerName queda totalmente olvidado y se utiliza realQueueManagerName.

Resolución de colas

El gestor de colas ahora busca una cola en la que colocar el mensaje y, para ello, selecciona la cola con la mejor coincidencia según las reglas que se describen en *Coincidencia exacta*, *Coincidencia de alias de cola*, *Cola S&F*, *Descubrimiento de cola* y *Anomalía* a continuación:

Coincidencia 'exacta'

Definición de cola local o de cola remota en la que el nombre de la cola coincide con el destinationQueueName y el nombre del gestor de colas de la cola coincide con el destinationQueueManagerName.

El término 'nombre del gestor de colas de la cola' tiene que explicarse con más detalle. En el caso de una cola local, es el mismo nombre del gestor de colas donde reside la cola. En el caso de una cola local localQ@localQM, localQM es el nombre del gestor de colas de la cola.

En el caso de una definición de colas remota remoteQ@remoteQM que resida en localQM, el nombre del gestor de colas de la cola es remoteQM.

Coincidencia de alias de cola

Si una cola (definición remota o local) tiene un nombre de gestor de colas y un alias coincidentes y ese alias coincide con destinationQueueName, esta cola se considerará una coincidencia. En efecto, la invocación de la transferencia de mensaje:

```
putMessage(destinationQueueManagerName, queueAliasName
```

se transforma en:

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

en este punto. El nombre original de la cola que se utiliza en la invocación de transferencia queda totalmente olvidado a partir de aquí en la resolución.

Cola S&F

Si no existe ninguna coincidencia exacta, el gestor de colas busca una coincidencia inexacta. Una coincidencia inexacta es una cola para almacenar y reenviar que acepta los menajes para el nombre de gestor de colas concreto. En la búsqueda de una cola para almacenar y reenviar se ignora el destinationQueueName. Si se encuentra una cola para almacenar y reenviar adecuada, se transfiere a ella el mensaje, mediante destinationQueueManagerName y destinationQueueName y la cola para almacenar y reenviar almacena el destino con el mensaje.

Descubrimiento de cola

Si no se ha encontrado ninguna cola que acepte el mensaje y el mensaje no es para una cola local, el gestor de colas intenta buscar la cola de destino remota y crear una definición de cola remota para ella de forma automática. Esto se denomina descubrimiento de cola. El gestor de colas sólo puede realizar el descubrimiento si:

- Existe una definición de conexión con el gestor de colas de destino.
- Existe una vía de acceso de comunicaciones activa al gestor de colas de destino.
- La cola de destino existe.
- Existe una conexión de tipo vía con un gestor de colas en la que hay una definición de conexión remota.

Si el descubrimiento es satisfactorio, se utiliza la definición de cola remota que se acaba de crear. Esto funciona como si se hubiera encontrado una coincidencia exacta en una definición de cola remota en primer lugar.

La definición de cola remota creada por el descubrimiento siempre es síncrona, aunque la cola en la que se resuelva sea asíncrona o una cola para almacenar y reenviar.

Anomalía

Si no se ha encontrado ninguna cola en los pasos anteriores, la transferencia de mensaje se considera fallida.

Empujar por la red

Un mensaje que se coloca en una cola remota se empuja por la red. La cola localiza en primer lugar una definición de conexión con el nombre correcto y, a continuación, transfiere el mensaje al gestor de colas remoto utilizando la definición de conexión como entrada al enlace de comunicaciones.

La cola busca una definición de conexión cuyo nombre sea el mismo que el nombre del gestor de colas de la cola. La conexión puede ser una conexión normal o una conexión de tipo vía.

Normal

Una conexión normal apunta a un escucha en el gestor de colas de destino. El mandato de transferencia de mensaje se direcciona directamente al gestor de colas de destino. La invocación de putMessage se resuelve como si se hubiera colocado en el gestor de colas a través de la API.

Vía

Una conexión de tipo vía apunta a otra conexión denominada conexión 'real'. Todos los mandatos ejecutados en la conexión de tipo vía se delegan a la conexión real. Las conexiones de tipo vía se pueden encadenar y, por lo tanto, el mandato puede desplazarse por varias vías indirectas antes de llegar a una conexión real. Los nombres del destino de la transferencia de mensaje no cambian por utilizar una conexión de tipo vía.

Finalmente el mandato se direcciona a una definición de conexión 'normal' y, a continuación, por la red a un gestor de colas, donde se resuelve la transferencia del mensaje.

Empuje de servidor local

Las colas de servidor local extraen mensajes de las colas para almacenar y reenviar. La ruta del empuje sólo abarca un salto de red. Sólo se extraen los mensajes para el gestor de colas que albergue la cola de servidor local. Los mensajes extraídos de la cola para almacenar y reenviar se presentan al gestor de colas mediante una invocación de método de transferencia normal y, a continuación, se resuelven como los normales. Los mensajes extraídos de este modo deben ir destinados a colas locales.

Utilización de alias

Introducción al uso de alias con colas de MQe y gestores de colas

Pueden asignarse alias para las colas de MQe a fin de proporcionar un nivel de direccionamiento indirecto entre la aplicación y las colas reales. Por ejemplo, se puede dar a una cola un número de alias y los mensajes enviados a cualquiera de estos nombres serán aceptados por la cola.

Utilización de alias de colas

Consulte "Utilización de alias de colas" en la página 14 para obtener información acerca de cómo se puede utilizar la asignación de alias con las colas de MQe.

Utilización de los alias de gestor de colas

En este tema se describe cómo se puede utilizar la asignación de alias con los gestores de colas de MQe.

Direccionamiento de un gestor de colas con varios nombres distintos

Imagine que tiene un gestor de colas SERVER23QM en el servidor SAMPLEHOST, que escucha en el puerto 8082. Si tiene una aplicación SERVICEX que accede a este gestor de colas y desea referirse al gestor de colas como SERVICEXQM. puede hacerlo utilizando un alias para el gestor de colas como se indica a continuación:

- **Configure una conexión en el SERVER23QM:**

  Nombre de la conexión/Gestor de colas de destino:
  SERVICEXQM

  Descripción:
  Definición de alias que permite que SERVER23QM reciba los mensajes que se envían a SERVICEXQM

  Canal: "null"

  Adaptador de red:
  "null"

  Opciones del adaptador de red:
  "null"

- **Cree una cola local en el gestor de colas SERVER23QM:**

  Nombre de cola:
  SERVICEXQ

  Gestor de colas:
  SERVER23QM

  La aplicación del área de servidor toma los mensajes de esta cola y los procesa, volviendo a enviar los mensajes al cliente.

una aplicación de MQe puede colocar mensajes en SERVICEXQ de los gestores de colas SERVER23QM o SERVICEXQM. En cualquier caso, el mensaje llegará a la SERVICEXQ.

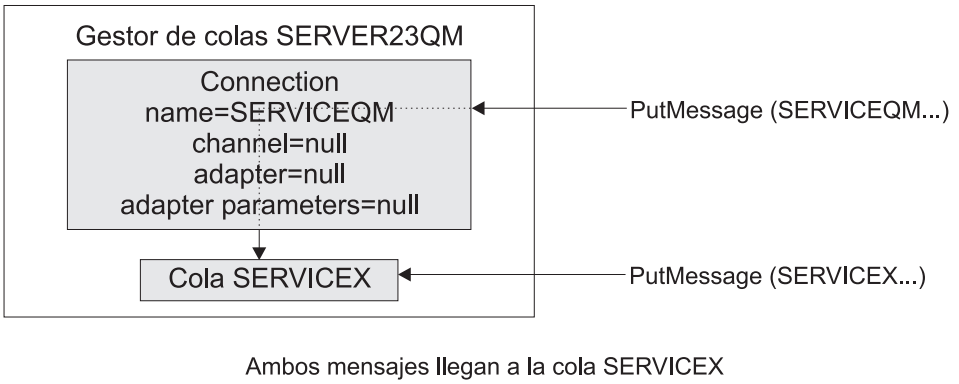

Figura 65. Referencia a un gestor de colas con dos nombres diferentes

Si la cola SERVICEXQ se mueve a otro gestor de colas, el alias de conexión se puede configurar en el nuevo gestor de colas y no es necesario cambiar las aplicaciones.

Direccionamientos diferentes de un gestor de colas a otro

Utilizando el caso que se acaba de describir en "Direccionamiento de un gestor de colas con varios nombres distintos", un gestor de colas de MQe de un dispositivo móvil (MOBILE0058QM) puede acceder a la cola SERVICEXQ de diferentes maneras.

Asignación de alias en el área de envío:

Si utiliza este método de direccionamiento, el gestor de colas receptor no sabe que el gestor de colas emisor le ha dado un nombre de alias. Este alias sólo se refiere al gestor de colas emisor.

En el dispositivo móvil:

- Cree una conexión desde `MOBILE0058QM` al gestor de colas `SERVER23QM`:

  Nombre de la conexión
  `SERVER23QM`

  Parámetro del adaptador de red
  `Red:SAMPLEHOST:8082`

- Cree un alias denominado `SERVICEXQM` para el gestor de colas `SERVER23QM`

Cuando se envía un mensaje desde la aplicación de dispositivo móvil al gestor de colas `SERVICEXQM`, MQe correlaciona el nombre de `SERVICEXQM` con `SERVER23QM` en la conexión y envía el mensaje al gestor de colas `SERVER23QM`.

Si, a continuación, `Mobile58QM` deseara enviar sus mensajes a otro gestor de colas del servidor, `Server24QM`, eliminaría el alias `SERVICEXQM` de la conexión `Server23QM` y lo añadiría a una conexión `Server24QM`. Esta acción no repercute en los gestores de colas receptores ni en las aplicaciones de envío.

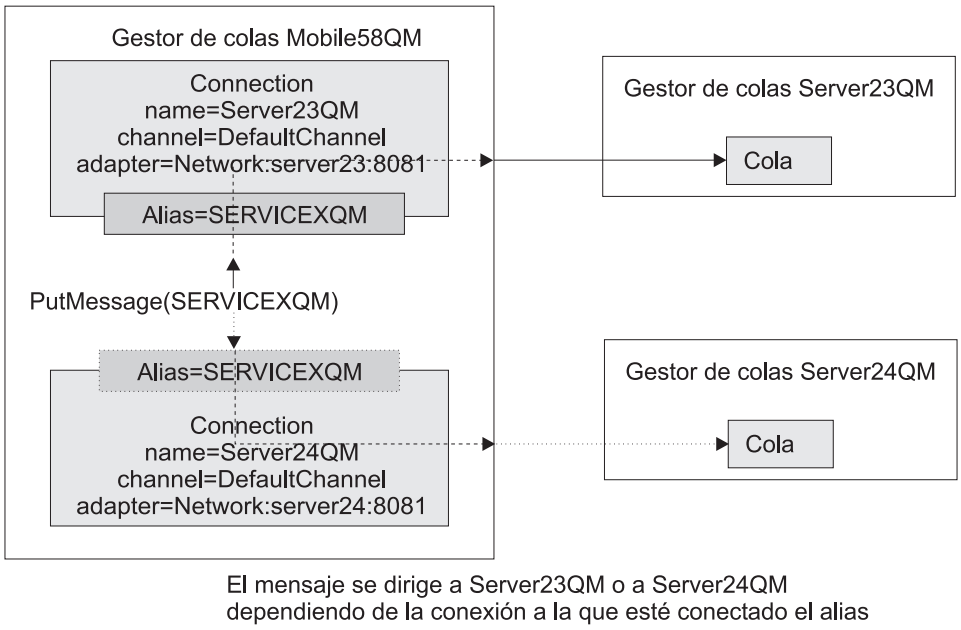

Figura 66. Referencia a un gestor de colas con dos nombres diferentes

Gestor de colas virtual en el área de recepción:

Utilizando este método, los gestores de colas emisores creen que sus mensajes se direccionan a través de un gestor de colas intermedio antes de llegar al gestor de colas de destino. En realidad el gestor de colas de destino no existe. El gestor de colas 'intermedio' calcula todo el tráfico de mensajes de este gestor de colas de destino virtual.

En el dispositivo móvil:

- Cree una conexión desde `MOBILE0058QM` al gestor de colas `SERVER23QM`:

Nombre de la conexión
SERVER23QM

Parámetro del adaptador de red
Red:SAMPLEHOST:8082

- Cree una segunda conexión con SERVICEXQM que direccione los mensajes a través de la primera conexión:

  Nombre de la conexión
  SERVICEXQM

  Parámetro del adaptador de red
  SERVER23QM

  Nota: Esto no es un alias. Es un *direccionamiento de ruta*, que indica que los mensajes que se dirigen a SERVICEXQM se tienen que direccionar a través del gestor de colas SERVER23QM de la parte receptora.

El direccionamiento de ruta del dispositivo móvil hace que los mensajes que se colocan en SERVICEXQM se dirijan a Server23QM. Server23QM obtiene los mensajes y advierte que su destino es el gestor de colas SERVICEXQM. Resuelve el nombre SERVICEXQM y encuentra que es un alias que representa el gestor de colas Server23QM (él mismo). A continuación, el gestor de colas Server23QM acepta los mensajes y los coloca en la cola.

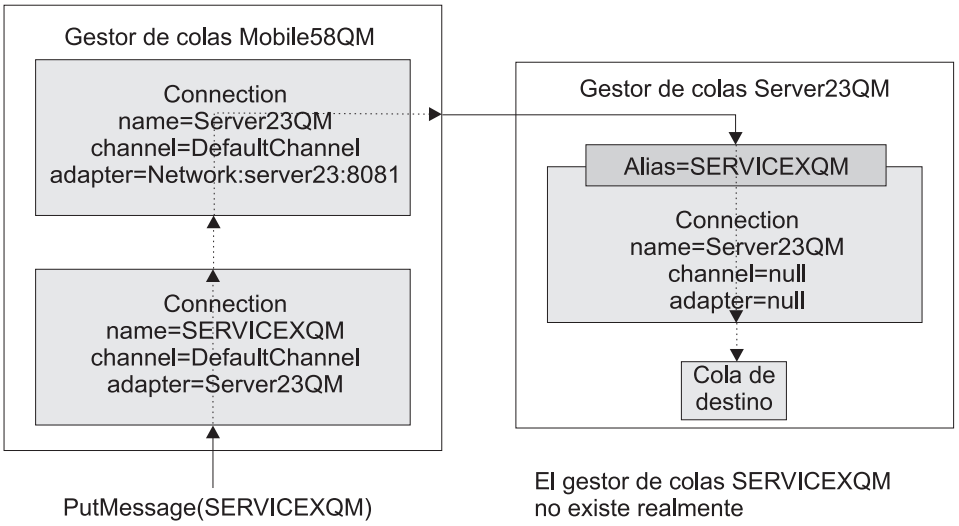

Figura 67. Referencia a un gestor de colas con dos nombres diferentes

Como alternativa a lo mencionado anteriormente, puede mantener SERVICEXQM, pero moverlo desde su máquina original a la misma máquina (pero una JVM diferente) que el gestor de colas Server23QM. SERVICEXQM necesita escuchar en un puerto diferente, así que la conexión desde Server23QM a SERVICEXQM también se tiene que cambiar.

Utilización de adaptadores

Describe el uso de adaptadores de almacenamiento y adaptadores de comunicaciones en aplicaciones de MQe y explica cómo escribir adaptadores propios.

En este capítulo se describe cómo implementar los adaptadores en una aplicación de MQe. Puede utilizar los adaptadores de MQe para correlacionar MQe con las interfaces del dispositivo de comunicaciones o almacenamiento. También puede escribir sus propios adaptadores.

Este capítulo contiene los siguientes apartados:

- Adaptadores de almacenamiento
- Adaptadores de comunicaciones
- Cómo escribir adaptadores

Adaptadores de almacenamiento

MQe proporciona los siguientes adaptadores de almacenamiento:

Adaptadores de almacenamiento

MQeCaseInsensitiveDiskAdapter
Proporciona soporte para la identificación no sensible a las mayúsculas y minúsculas cuando se localiza un determinado archivo en el almacenamiento permanente.

MQeDiskFieldsAdapter
Proporciona soporte para leer y grabar en el almacenamiento permanente.

MQeMappingAdapter
Proporciona soporte para correlacionar los nombres de archivo largos con los nombres de archivo cortos.

MQeMemoryFieldsAdapter
Proporciona soporte para leer y grabar en el almacenamiento no permanente.

MQeMidpFieldsAdapter
Proporciona soporte para leer y grabar en el almacenamiento permanente en un entorno MIDP.

MQeReducedDiskFieldsAdapter
Proporciona soporte para grabar a alta velocidad en el almacenamiento permanente.

Tenga en cuenta que el comportamiento de estos adaptadores no se puede alterar. Para obtener más información sobre el comportamiento específico de cada adaptador de almacenamiento, consulte la consulta de programación en Java de MQe y la consulta de programación en C de MQe.

Adaptadores de comunicaciones

MQe proporciona los siguientes adaptadores de comunicaciones.

Adaptadores de comunicaciones

MQeMidpHttpAdapter
Proporciona soporte para leer y grabar en la red utilizando el protocolo HTTP 1.0 en un entorno MIDP.

MQeTcpipHistoryAdapter
Proporciona soporte para leer y grabar en la red utilizando el protocolo TCP. Este adaptador ofrece el mejor rendimiento de TCP al colocar en antememoria los datos utilizados recientemente. Por lo tanto, recomendamos que utilice este adaptador.

MQeTcpipLengthAdapter
Proporciona soporte para leer y grabar en la red utilizando el protocolo TCP.

MQeTcpipHttpAdapter
Proporciona soporte para leer y grabar en la red utilizando el protocolo HTTP 1.0. También ofrece soporte para pasar las peticiones HTTP a través de los servidores proxy.

Nota: Si utiliza la JVM de Microsoft, ésta establece automáticamente las propiedades http:proxyHost y http:proxyPort mediante la configuración de Internet Explorer. Si no es necesario utilizar servidores proxy para MQe, establezca la propiedad de Java http.proxySet en false.

MQeUdpipBasicAdapter
Proporciona soporte para leer y grabar en la red utilizando el protocolo UDP. Este adaptador sólo utiliza un puerto en el servidor. El comportamiento de este adaptador es especialmente sensible a los diferentes valores de propiedades de Java, como se detalla en la consulta de programa en Java de MQe.

MQeWESAuthenticationAdapter
Proporciona soporte para pasar las peticiones HTTP a través de los servidores proxy de autentificación de MQe y los servidores proxy transparentes.

Puede modificar el comportamiento de estos adaptadores utilizando las propiedades de Java. Para obtener más información acerca de cómo utilizar estas propiedades y de su repercusión en cada adaptador de comunicaciones, consulte la consulta de programación en Java de MQe.

También puede escribir sus propios adaptadores para adaptar MQe a su entorno concreto. En la sección siguiente se describen algunos ejemplos de adaptador que se suministran para ayudarle a realizar esta tarea.

Cómo escribir adaptadores

También puede escribir sus propios adaptadores para adaptar MQe a su entorno concreto. En este tema se describen algunos ejemplos de adaptador que se suministran para ayudarle a realizar esta tarea.

Este ejemplo no tiene el propósito de sustituir a los adaptadores suministrados con MQe, sino que es una sencilla introducción sobre cómo crear un adaptador de comunicaciones.

Para utilizar su adaptador de comunicaciones, tiene que especificar el nombre de clase correcto cuando cree el escucha en el gestor de colas de servidor y especificar la definición de conexión en el gestor de colas de cliente.

Todos los adaptadores de comunicaciones deben heredar de MQeCommunicationsAdapter e implementar los métodos necesarios. Para mostrar cómo se hace, utilizaremos el adaptador de ejemplo `examples.adapters.MQeTcpipLengthGUIAdapter`. Se trata de un sencillo ejemplo que acepta que se graben datos. También coloca la longitud y la cantidad de datos que se van a grabar en la salida estándar, delante de los datos. Cuando el adaptador lee los datos, la longitud de datos se graba en la salida estándar. No se lleva a cabo una adecuada recuperación y comprobación de errores. El usuario debe añadir esto a los adaptadores.

Los adaptadores de MQe utilizan el constructor predeterminado. Por este motivo, se utiliza un método `activate()` para configurar el adaptador con un método `open()` para preparar el adaptador para la comunicación.

El método `activate()` sólo se invoca una vez durante el ciclo de vida de un adaptador y, por lo tanto, se utiliza para configurar la información de MQePropertyProvider. MQePropertyProvider busca internamente para verificar que la propiedad especificada esté disponible. Si no está disponible, comprueba las propiedades de Java. De esta manera, es posible que un usuario especifique una propiedad que puede establecer la aplicación o la línea de mandatos de JVM. MQeCommunicationsAdapter proporciona dos variables que permiten que el adaptador identifique su función en la conversación de comunicaciones:

- Si MQeListener utiliza el adaptador, la variable `listeningAdapter` se establece en true.
- Si el adaptador de escucha ha creado el adaptador en respuesta a una petición de entrada, la variable `responderAdapter` se establece en true.

El siguiente código, que se ha tomado del método `activate()`, muestra cómo obtener la información de MQePropertyProvider.

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

El método `open()` se invoca antes de cada conversación y, por lo tanto, se debe utilizar para establecer la información que se tiene que restablecer de cada petición o respuesta. Por ejemplo, un adaptador que no sea permanente debe crear un socket cada vez que se abre. El siguiente código muestra la utilización de las variables que identifican la función del adaptador en la conversación:

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

Después de invocar los métodos `activate()` y `open()`, se invoca el método `waitForContact` del adaptador de escucha. Este método debe esperar en una ubicación con nombre. En una red IP, será un puerto con nombre. Cuando se recibe una petición, se crea un nuevo adaptador.

Nota: Este método debe establecer listeningAdapter en false y responderAdapter en true.

Después de configurar correctamente el adaptador, hay que devolverlo al canal de llamada. El siguiente código muestra cómo hacerlo:

```
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;
```

El adaptador del iniciador y el adaptador de respuesta son responsables de la parte principal de la conversación. El iniciador inicia la conversación. El canal de respuesta lo crea el adaptador de escucha, lee la petición que se vuelve a pasar a MQe, que a continuación graba una respuesta. El adaptador determina cómo se lleva a cabo la lectura y la grabación. El ejemplo utiliza una BufferedInputStream y una BufferedOutputStream.

Nota: Utilice una modalidad de no bloqueo de lectura y grabación. Esto permite que el adaptador responda a las peticiones de conclusión.

El siguiente código, que se ha tomado del método `waitForContact()`, muestra cómo se puede grabar la lectura de no bloqueo. Puesto que MQe da soporte a todos los entornos del tiempo de ejecución de Java, no podemos utilizar las clases específicas de Java versión 1.4 para nuestros ejemplos, aunque esta versión contiene clases que no son de bloqueo nuevas.

```
do {
      try {
         clientSocket = serverSocket.accept();
```

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

Ejemplo de adaptador de comunicaciones

En este ejemplo se utilizan las clases Java estándar para manipular TCPIP y se añade un protocolo propio en la parte superior. Este protocolo contiene una cabecera cuyo paquete de datos contiene datos de una longitud de cuatro bytes y va seguido por los datos reales. Esto es así para que el extremo receptor sepa la cantidad de datos que ha de esperar.

Este ejemplo no está diseñado para sustituir a los adaptadores suministrados con MQe, sino que es una sencilla introducción sobre cómo crear adaptadores de comunicaciones. En realidad, se debe prestar mucha atención en el manejo de errores, las tareas de recuperación y la comprobación de parámetros. Dependiendo de la configuración de MQe que se utilice, los adaptadores suministrados pueden ser suficientes.

Se crea un nuevo archivo de clase, heredado de MQeAdapter. Para contener la información de esta instancia del adaptador se definen algunas variables como el nombre del sistema principal, el número de puerto y los objetos de la corriente de salida.

Nota: Con las comunicaciones, asegúrese de que la información de conexión sea la correcta. Por ejemplo, la conexión http de J2ME no tiene implementación del tiempo de espera. En J2SE, el cliente excede el tiempo de espera con una excepción de E/S. En Midp el servidor excede el tiempo de espera. Si se ha incrementado el tiempo de espera de lectura predeterminada del cliente J2SE, se lanza la misma excepción, que es `com.ibm.mqe.MQeException: Data: (code=7)`. Esto es así porque el servidor vuelve a grabar la excepción en el cliente y el cliente no puede restaurar estos datos.

Para el objeto se utiliza el constructor MQeAdapter, de modo que no es necesario añadirle al constructor ningún código adicional.

```
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( );
```

A continuación, se codifica el método `activate`. Este es el método que extrae del descriptor de archivo el nombre de la dirección de red de destino si se trata de un conector o el puerto de escucha si se trata de un escucha. El parámetro fileDesc contiene el nombre de clase del adaptador, o el nombre de alias, y cualquier dato de dirección de red del adaptador, como `MyTcpipAdapter:127.0.0.1:80`. El parámetro thisParam contiene cualquier dato de parámetro que se haya establecido cuando el administrador ha definido la conexión, normalmente el valor será "?Channel". El parámetro thisOpt contiene las opciones de configuración del adaptador que ha establecido el administrador, por ejemplo, `MQe_Adapter_LISTEN` si este adaptador ha de escuchar las conexiones de entrada.

```
  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 );
    }
  }
```

Se debe definir el método `close` de modo que cierre las corrientes de salida y deseche los datos restantes de los almacenamientos intermedios de las corrientes. Durante una sesión entre un cliente y un servidor se invoca muchas veces el método `close`, sin embargo, cuando el canal ha finalizado por completo con el adaptador, invoca a MQe con la opción `MQe_Adapter_FINAL`. Si el adaptador ha de tener una conexión de socket para la duración del canal, la llamada que tiene establecido `MQe_Adapter_FINAL` es la que en realidad cierra el socket, las demás llamadas simplemente desechan los almacenamientos intermedios. Sin embargo, si se ha de utilizar un nuevo socket en cada petición, cada invocación a MQe cerrará el socket y las invocaciones de `open` posteriores deberán asignar un nuevo 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;
  }
```

Para aceptar una petición de conexión de entrada, se debe codificar el método control para que pueda manejar una petición MQe_Adapter_ACCEPT. Esto sólo está permitido si el socket es un escucha (un socket servidor). Todas las opciones especificadas para el socket de escucha (excluida MQe_Adapter_LISTEN) se copian en el socket que se ha creado como resultado de la aceptación. Esto se consigue utilizando otra opción de control, MQe_Adapter_SETSOCKET, que permite que se pase un objeto de socket al adaptador cuya instancia se acaba de crear.

```
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 ) );
  }
```

El método open deberá comprobar si existe un socket de escucha o un socket de conector y crear el objeto de socket adecuado. Utilizando el método control y pasándole un nuevo objeto de socket se consigue la reinicialización de las corrientes de entrada y salida. El parámetro opt se puede establecer en MQe_Adapter_RESET, lo que significa que las operaciones anteriores se han completado y cualquier lectura o grabación nuevas constituyen una nueva petición.

```
  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 ) );
    }
```

El método read puede tomar un parámetro en el que se especifique el tamaño máximo de registro que se debe leer.

En este ejemplo se invocan las rutinas internas para que lean los bytes de datos y efectúen la recuperación de errores (si corresponde) y, a continuación, devuelvan la longitud de matriz de bytes correcta para el número de bytes leídos. Asegúrese de que en este socket sólo se lleve a cabo una lectura cada vez. El parámetro opt se puede establecer en:

MQe_Adapter_CONTENT
lee cualquier contenido de mensaje

MQe_Adapter_HEADER
lee cualquier información de cabecera

```
{ 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     */
    }
```

El método readByte es una rutina interna diseñada para leer un solo byte de datos del socket y reintentar cualquier error un número específico de veces o generar una excepción de fin de archivo si ya no hay más datos que leer.

```
  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     */
   }
```

El método readBytes es una rutina interna diseñada para leer un número de bytes de datos del socket y reintentar cualquier error un número específico de veces o generar una excepción de fin de archivo si ya no hay más datos que leer.

```
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    */
    }
```

El método `readln` lee una serie de bytes que finalizan con un carácter 0x0A y pasa por alto los caracteres 0x0D.

```
    {
    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 ?       */
        }
      }
    }
```

El método `status` devuelve información de estado acerca del adaptador. En este ejemplo, para la opción `MQe_Adapter_NETWORK` devuelve el tipo de red (TCPIP), para la opción `MQe_Adapter_LOCALHOST` devuelve la dirección del sistema principal local tcpip.

```
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 ) );
    }
```

El método `write` graba un bloque de datos en el socket. Debe asegurarse de que sólo se emita en el socket una grabación cada vez. En este ejemplo, invoca una rutina interna `writeBytes` para que grabe los datos reales y efectúe cualquier recuperación de errores que corresponda.

El parámetro opt se puede establecer en:

MQe_Adapter_FLUSH
desecha los datos de los almacenamientos intermedios

MQe_Adapter_HEADER
graba cualquier registro de cabeceras

MQe_Adapter_HEADERRSP
graba cualquier registro de respuestas de cabecera

```
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 */
    }

  }
```

`writeBytes` es un método interno que graba en un socket una matriz (o una matriz parcial) de bytes e intenta una sencilla recuperación de errores si se producen errores.

```
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  */
    }
   }
```

El método `writeLn` graba en el socket una serie de caracteres, finalizando con los caracteres 0x0A y 0x0D.

El parámetro opt se puede establecer en:

MQe_Adapter_FLUSH
desecha los datos de los almacenamientos intermedios

MQe_Adapter_HEADER
graba cualquier registro de cabeceras

MQe_Adapter_HEADERRSP
graba cualquier registro de respuestas de cabecera

```
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  */
  }
```

Ya ha completado un adaptador TCPIP (aunque muy sencillo) que se comunicará con otra copia de sí mismo, una de las cuales se habrá iniciado como un escucha y la otra como un conector.

Ejemplo de adaptador de almacenamiento de mensajes

Este ejemplo crea un adaptador que se utilizará como una interfaz con un almacenamiento de mensajes. Para manipular los archivos del almacenamiento utiliza las clases de E/S Java estándar.

Este ejemplo no está diseñado para sustituir a los adaptadores suministrados con MQe, sino que es una sencilla introducción sobre cómo crear un adaptador de almacenamiento de mensajes.

Se crea un nuevo archivo de clase, heredado de MQeAdapter. Para contener la información de esta instancia del adaptador se definen algunas variables como el nombre del archivo/mensaje y la ubicación del almacenamiento de mensajes.

Para el objeto se utiliza el constructor MQeAdapter, de modo que no es necesario añadirle al constructor ningún código adicional.

```
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;
```

Debido a que este adaptador implementa FilenameFilter, se debe codificar el siguiente método. Este es el mecanismo de filtrado que se utiliza para seleccionar archivos de un tipo determinado en el almacenamiento de mensajes.

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

A continuación, se codifica el método `activate`. Este es el método que extrae del descriptor de archivo el nombre del directorio que se debe utilizar para contener todos los mensajes.

El parámetro Object de la invocación al método puede ser un objeto de atributo. Si lo es, este es el atributo que se utiliza para codificar y/o descodificar el mensaje en el almacenamiento de mensajes.

Las opciones de Object para este adaptador son:

- MQe_Adapter_READ
- MQe_Adapter_WRITE
- MQe_Adapter_UPDATE

Deberá ignorarse cualquier otra opción.

```
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 );
    }
```

El método `close` no permite la lectura o grabación.

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

El método control se debe codificar para manejar MQe_Adapter_LIST, esto es, una petición para listar todos los archivos del directorio que coincidan con el filtro, y también para que pueda manejar una petición MQe_Adapter_FILTER, esto es, una petición para establecer un filtro que controle cómo se listan los archivos.

```
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        */
    }
```

El método erase se utiliza para suprimir un mensaje del almacenamiento de mensajes.

```
  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" );
    }
```

El método open establece los valores booleanos que permiten la lectura o grabación de mensajes.

```
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 );
    }
```

El método readObject lee un mensaje del almacenamiento de mensajes y vuelve a crear un objeto del tipo correcto. También descifra y descomprime los datos si se suministra un atributo en la llamada activate. Esta es una función especial ya que para una petición de lectura de un archivo que coincida con el criterio especificado en el parámetro de lectura, se devuelve el primer mensaje coincidente encontrado.

```
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" );
    }
```

El método status devuelve información de estado acerca del adaptador. En este ejemplo puede devolver el tipo de filtro o el nombre de archivo.

```
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 ) );
    }
```

El método `writeObject` graba un mensaje en el almacenamiento de mensajes. Comprime y cifra el objeto de mensaje si se suministra un atributo en la invocación al método `activate`.

```
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" );
    }
```

Ya ha completado un adaptador de almacenamiento de mensajes (aunque muy sencillo) que lee y graba los objetos de mensajes en un almacenamiento de mensajes.

Se pueden codificar variaciones de este adaptador para, por ejemplo, almacenar los mensajes en una base de datos en la memoria no volátil.

Adaptador de comunicaciones de WebSphere Everyplace Suite (WES)

MQe proporciona una sofisticada seguridad que permite que las aplicaciones se ejecuten mediante HTTP, a través de la protección que ofrece un cortafuegos de Internet. La finalidad del adaptador de comunicaciones de WebSphere Everyplace es permitir que las aplicaciones de MQe se autentifiquen con el proxy de autentificación de WebSphere Everyplace y, por lo tanto, los mensajes puedan fluir por él. En el diagrama siguiente se muestra un caso práctico básico con dos aplicaciones que se comunican por Internet a través del proxy de autentificación de WebSphere Everyplace.

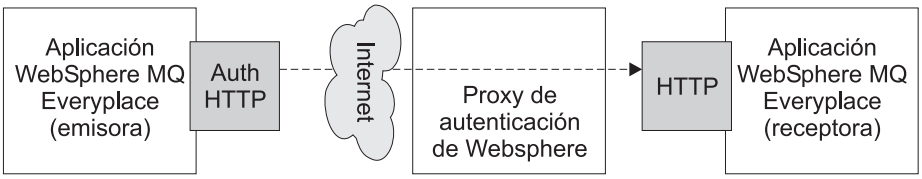

Figura 68. Aplicaciones que se comunican a través del proxy de autentificación de WebSphere

El adaptador de MQe actúa como el adaptador HTTP de autentificación en la aplicación que envía los datos. La aplicación que los recibe puede utilizar el mismo adaptador o bien el adaptador HTTP estándar proporcionado con MQe.

No obstante, el verdadero valor de MQe es que permite que se realice el envío asíncrono de mensajes en un entorno típicamente síncrono. Se pueden recopilar las peticiones colocadas en cola procedentes de la aplicación receptora y manejarlas independientemente de la hora. En el diagrama siguiente se muestra cómo hacer que las peticiones entrantes alcancen los servidores de MQ de forma asíncrona.

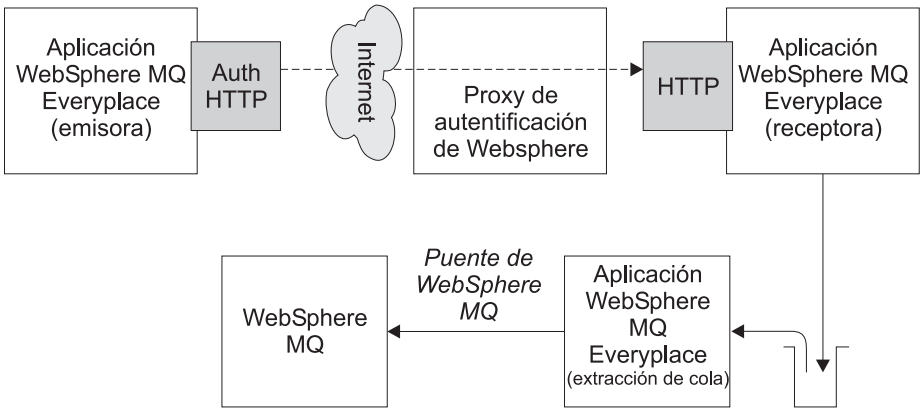

Figura 69. Aplicaciones que se comunican de forma asíncrona a través del proxy de autentificación de WebSphere

En cada uno de estos entornos, el proxy de autentificación de WebSphere añade la posibilidad de controlar el acceso a las aplicaciones receptoras. El código del adaptador da soporte a esta capacidad añadiendo información de contraseña e ID de usuario (que suministra la aplicación) a cada petición HTTP saliente. El proxy de autentificación de WebSphere acepta estas peticiones y verifica que las credenciales suministradas sean válidas para el entorno actual. Si las credenciales son válidas, el proxy reenvía la petición a la aplicación receptora.

Archivos de adaptador de WebSphere Everyplace Suite (WES)

En una instalación estándar de MQe, el adaptador de WebSphere Everyplace consta de los archivos siguientes de los que recibe soporte:

...\Java\com\ibm\mqe\adapters\MQeWESAuthenticationAdapter.class
- Clase de adaptador WebSphere Everyplace.

...\Java\examples\application\Example7.class
- Aplicación de ejemplo compilada que utiliza el adaptador.

...\Java\examples\application\Example7.java
- Código fuente de la aplicación de ejemplo.

...\Java\examples\adapters\WESAuthenticationGUIAdapter.class
- Adaptador de ejemplo compilado que añade una interfaz al adaptador de WebSphere Everyplace. Tal como ocurre con otras clases de ejemplo, esta clase no sustituye realmente a la clase del adaptador básico de WES, sino que es sólo una demostración de cómo puede ajustar el adaptador de WES a sus necesidades.

...\Java\examples\adapters\WESAuthenticationGUIAdapter.java
- Código fuente del adaptador de ejemplo.

Si la variable de entorno *CLASSPATH* se establece para que busque todas las clases dentro de la carpeta de Java de MQe, se podrá acceder a los archivos de clases de adaptador de WebSphere Everyplace desde el entorno Java. Si no puede acceder a los archivos, escriba un mandato parecido al siguiente:

```
set CLASSPATH=%CLASSPATH%;c:\mqe\java
```

Esto hace que las nuevas clases estén visibles para Java. (El formato exacto de este mandato puede variar de un sistema a otro.) Tras llevar a cabo esta acción, debe poder utilizar las clases de adaptador de WebSphere Everyplace de la misma manera que otras clases de MQe.

Utilización del adaptador de WebSphere Everyplace Suite (WES)

En este apartado se proporciona información sobre cómo utilizar el adaptador de WebSphere Everyplace. La información se divide en tres partes:

Funcionamiento general
Describe en detalle cómo utilizar el adaptador en sus aplicaciones.

Utilización del ejemplo de diálogo de autentificación
Describe cómo utilizar una clase de ejemplo, examples.adapters.WESAuthenticationGUIAdapter. Esta clase se deriva de la clase del adaptador básico de WES y proporciona una pequeña interfaz de usuario para recopilar el ID y la contraseña del usuario.

Utilización del ejemplo de aplicación
Describe cómo utilizar el archivo de ejemplo suministrado examples.application.Example7 que se ha configurado para usar el adaptador básico de WES.

En la información de este apartado se presupone que se han instalado y configurado correctamente el proxy de autentificación de WebSphere Everyplace y MQe. También se presupone que se han configurado un gestor de colas de servidor de MQe y un gestor de colas de cliente de MQe.

Funcionamiento general:

1. Configure el gestor de colas del cliente para que envíe mensajes mediante el nuevo adaptador modificando el archivo .ini de configuración del gestor de colas del cliente de forma que el alias *Network* haga referencia a com.ibm.mqe.adapters.MQeWESAuthenticationAdapter. Utilice el mandato siguiente:

   ```
   (ascii)Network=com.ibm.mqe.adapters.MQeWESAuthenticationAdapter
   ```

2. Configure el gestor de colas del servidor para que descodifique la corriente de datos que el adaptador del cliente proporciona, bien mediante el nuevo adaptador o bien mediante el adaptador HTTP estándar. Para ello, cambie la línea del archivo .ini de configuración del gestor de colas del servidor para que el alias de *Network* apunte a com.ibm.mqe.adapters.MQeWESAuthenticationAdapter o a com.ibm.mqe.adapters.MQeTcpipHttpAdapter. Utilice uno de los mandatos siguientes:

   ```
   (ascii)Network=com.ibm.mqe.adapters.MQeWESAuthenticationAdapter
   ```

   ```
   (ascii)Network=com.ibm.mqe.adapters.MQeTcpipHttpAdapter
   ```

3. Modifique el código del gestor de colas del cliente de forma que el ID de usuario y la contraseña requeridos se definan antes de que se inicie la primera operación de la red. Por ejemplo, inserte la línea siguiente cerca del principio del código:

   ```
   com.ibm.mqe.adapters.MQeWESAuthenticationAdapter.
   setBasicAuthorization("miIDUsuario@miDominio", "miContraseña");
   ```

   Sustituya los parámetros por un ID de usuario y una contraseña de servidor WES válidos.

   Es posible que también tenga que añadir código para recoger la nueva excepción `Except_Authenticate` de MQeException después de realizarse cada operación de red, en el caso de que las credenciales proporcionadas no sean válidas.

4. Compruebe que el gestor de colas del cliente todavía pueda enviar mensajes al gestor de colas del servidor sin pasar a través del proxy.

5. Configure la máquina cliente para que envíe peticiones HTTP a través del proxy. En función de cómo se haya configurado WES, el adaptador necesitará trabajar con un *proxy transparente* o bien con un *proxy de autentificación*.

   Como *proxy transparente*
   En este modo, el servidor WES actúa como un proxy HTTP sencillo. En este caso, tiene que establecer las siguientes propiedades del sistema de la aplicación en Java que se relacionan con la información del proxy:

   http.proxyHost
   Debe establecerse en el nombre del sistema principal del proxy de WES

http.proxyPort
Debe establecerse en el nombre del puerto en el que el proxy esté a la escucha

http.proxySet
Debe establecerse en `true`, que indica al adaptador que utilice el modo de proxy transparente

Los parámetros anteriores pueden establecerse añadiendo lo siguiente a la aplicación en Java:

```
System.getProperties( ).put( "http.proxySet",  "true" );
 System.getProperties( ).put( "http.proxyHost", "wes.hursley.ibm.com" );
 System.getProperties( ).put( "http.proxyPort", "8082" );
```

La conexión del gestor de colas del cliente con el servidor de MQe de destino es parecida a una conexión que no utiliza el proxy de WES.

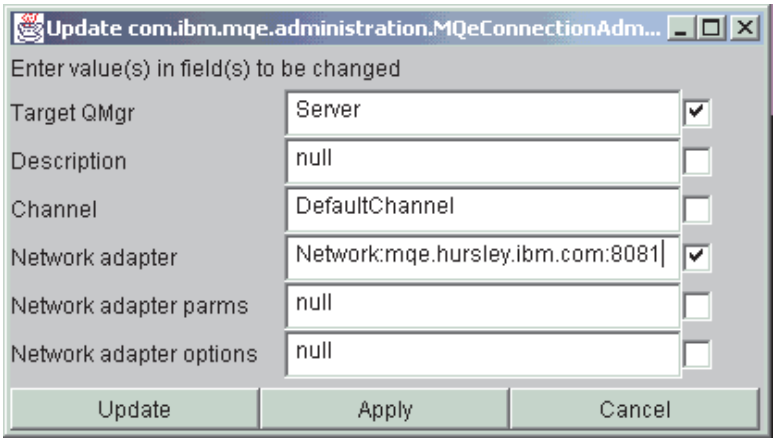

Figura 70. Panel de interfaz de administración

Debe reiniciar los gestores de colas del cliente y del servidor para que los nuevos valores entren en vigor. De este modo, el cliente debería poder enviar mensajes al servidor a través del proxy.

Como un *proxy de autentificación*
En este modo, el servidor WES reenvía las peticiones para servicios que se han recibido, en función del URL suministrado. Por ejemplo, puede que desee que las peticiones para `http://wes.hursley.ibm.com/mqe` se reenvíen a un gestor de colas de MQe que se ejecute en `mqe.hursley.ibm.com:8082`.

Para configurar esto desde MQe, debe actualizar la referencia de *`connection`* del cliente al servidor.

Adaptador de red de destino
Debe señalar a la máquina del proxy de autentificación y al puerto

Parámetros de adaptador de red
Debe contener el nombre de vía de acceso del servicio solicitado

Si utiliza la herramienta de administración de ejemplo de MQe, seleccione **Conexión** y, a continuación, **Actualizar** para configurar dichos valores.

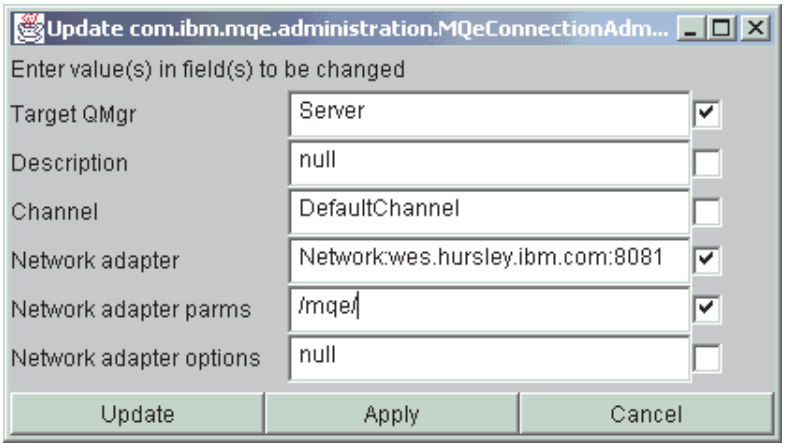

Figura 71. Panel de interfaz de administración

> **Nota:** La referencia al servidor WES se especifica en el campo **Adaptador de red** y el nombre de vía de acceso se especifica en el campo **Parámetros de adaptador de red**.
> Debe reiniciar los gestores de colas del cliente y del servidor para que los nuevos valores entren en vigor. De este modo, el cliente debería poder enviar mensajes al servidor a través del proxy.

Utilización del ejemplo de diálogo de autentificación:

En la información siguiente se describe la utilización del archivo de clases de ejemplo, examples.adapters.WESAuthenticationGUIAdapter. Esta clase añade una pequeña interfaz de usuario a la función del adaptador básico de WES.

1. Siga los pasos (1) y (2) de los procedimientos de "Funcionamiento general" en la página 121, pero sustituya 'WESAuthenticationGUIAdapter' por 'WESAuthenticationAdapter' en el paso (1).
2. Configure los valores TCP/IP del cliente tal como se indica en el paso (5) del apartado 'Funcionamiento general'.

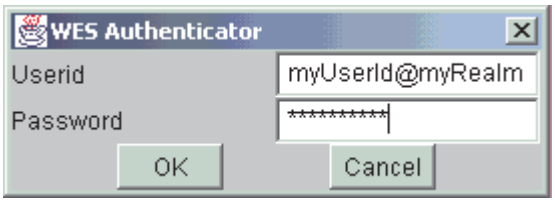

Figura 72. Diálogo del usuario del adaptador de WebSphere Everyplace Suite

Ahora el cliente debería poder enviar mensajes al servidor utilizando el adaptador WESAuthenticationGUIAdapter. Este adaptador intercepta las llamadas de grabación efectuadas en el adaptador de WES y al realizarse la primera petición hace emerger un recuadro de diálogo que solicita información de ID de usuario y de contraseña.

Cuando el usuario pulsa **Aceptar** o pulsa la tecla **Intro**, se invoca el método setBasicAuthorization() con los valores de los campos **ID de usuario** y **Contraseña**. A continuación se reenvía write() al adaptador de WES subyacente. El recuadro de diálogo también tiene un botón Cancelar que, al seleccionarse, cancela la operación de grabación actual anulando el reenvío de la petición al adaptador de WES. Esta acción hace que se lance `MQeException (Except_Stopped)`.

Si la autentificación no es satisfactoria, el recuadro de diálogo vuelve a aparecer en el siguiente método write() junto con toda la información proporcionada por el servidor. Para aprender de una anomalía de

autentificación, el adaptador de ejemplo intercepta las llamadas read() y recoge todas las excepciones MQeExceptions de `Except_Authenticate` procedentes del adaptador.

Nota: Generalmente, los navegadores de la web no envían información de autentificación en el primer flujo. Normalmente esto da como resultado una respuesta `401` ó `407` que contiene la información del reino. Sólo entonces el navegador envía la petición autentificada. Puede que los clientes del usuario deseen seguir este convenio.

Utilización del ejemplo de aplicación:

En la información siguiente se describe la utilización del archivo de aplicación de ejemplo, examples.application.Example7. Este ejemplo es parecido al ejemplo de programación de MQSeries Everyplace examples.application.Example1 y en él se utiliza el adaptador básico de WES para las comunicaciones.

1. Siga los pasos (1) y (2) de los procedimientos de "Funcionamiento general" en la página 121.
2. Configure los valores de TCP/IP del cliente como en el paso (5) de "Funcionamiento general" en la página 121.
3. Modifique el archivo de ejemplo ...\Java\examples\application\Example7.java insertando un ID de usuario y una contraseña válidos y, a continuación, vuelva a compilar la aplicación.
4. Reinicie el servidor.
5. Ejecute el programa Example7 utilizando el mandato siguiente:

   ```
   java examples.application.Example7 servidor client.ini
   ```

   donde

   Servidor
   es el nombre del gestor de colas remotas (el cliente ya sabe cómo alcanzar dicho gestor de colas)

   client.ini
   señala el archivo de configuración .ini del cliente.

   La aplicación inicia el gestor de colas del cliente con el proxy, coloca un mensaje en el servidor y, a continuación, obtiene un mensaje de éste.

Utilización de las normas

Introducción a la utilización de las normas de MQe

MQe utiliza normas (que son básicamente salidas de usuario) para permitir que las aplicaciones supervisen y modifiquen el comportamiento de algunos de sus principales componentes. Las normas tienen forma de métodos en las clases Java o funciones en los métodos C que se cargan al inicializar los componentes de MQe.

Las normas de un componente se invocan en determinados puntos de su ciclo de ejecución. Se espera que los métodos de las normas con signaturas particulares estén disponibles, por lo que cuando se proporcionen las implementaciones de las normas, asegúrese de utilizar las signaturas correctas.

Se proporcionan normas predeterminadas o de ejemplo de todos los componentes importantes de MQe. Puede personalizar estas normas para satisfacer los requisitos particulares del usuario. En la base de código en Java, la interfaz de `MQeQueueProxy` proporciona al usuario métodos de acceso a las colas, lo que permite que el usuario interactúe en ciertos métodos de normas.

Las normas se pueden agrupar en las siguientes categorías:

- Normas del gestor de colas.
- Normas de colas.

- Normas de atributos.
- Normas del puente.

Las normas también se pueden categorizar en dos grupos en función de si pueden afectar al comportamiento de la aplicación (normas de modificación) o si tienen finalidades de notificación (normas de notificación).

Normas del gestor de colas

Se invocan las normas del gestor de colas cuando:
- Se activa el gestor de colas
- Se cierra el gestor de colas
- Se añade una cola al gestor de colas (sólo base de código en Java)
- Se elimina una cola del gestor de colas (sólo base de código en Java)
- Se lleva a cabo una operación put de mensajes
- Se lleva a cabo una operación get de mensajes
- Se lleva a cabo una operación delete de mensajes
- Se lleva a cabo una operación undo de mensajes
- Se activa el gestor de colas para que transmita los mensajes pendientes, según se describe en las Normas de transmisión

Carga y activación de las normas del gestor de colas

En este tema se describe cómo cargar y activar las normas del gestor de colas en Java y en C.

Java:

Java:

Las normas del gestor de colas se cargan, o se cambian, cada vez que se recibe un mensaje de administración del gestor de colas que contiene una petición de actualización de la clase de normas del gestor de colas.

Si una norma del gestor de colas ya se ha aplicado al gestor de colas, se le pide a la norma existente si se puede sustituir por una norma diferente. Si la respuesta es afirmativa, la nueva norma se carga y se activa. No es necesario reiniciar el gestor de colas.

La herramienta de la línea de mandatos `QueueManagerUpdater` del paquete `examples.administration.commandline` muestra cómo se crea dicho mensaje de administración.

Norma de gestor de colas de ejemplo en C:

El módulo de normas del usuario se carga e inicializa cuando el gestor de colas se carga en la memoria. Esto se produce como consecuencia de las llamadas a `mqeAdministrator_QueueManager_create()` o bien a `mqeQueueManager_new()`. Los pasos de configuración son los siguientes:
- La aplicación tiene que registrar un alias de normas, enlazándolo con el punto de entrada y el nombre del módulo de normas, mediante mqeClassAlias_add(), por ejemplo:

```
#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);
```

- Este alias de normas se tiene que incluir en los parámetros de inicio del gestor de colas que se pasan a `mqeAdministrator_QueueManager_create()` o bien a `mqeQueueManager_new()`, por ejemplo:

```
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);
```

- El usuario debe proporcionar un punto de entrada o una función de inicialización. A continuación se muestra un ejemplo de una función de inicialización para una implementación de normas. Los miembros de las estructuras de los parámetros están documentados en la consulta de programación en C de MQe.

```
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;
}
```

El módulo de normas se descarga cuando se libera el gestor de colas. Tenga en cuenta que, a diferencia del código base en Java, la implementación de normas está vinculada al ciclo de vida de ejecución de un solo gestor de colas y no se puede sustituir en el curso de este ciclo de vida.

Utilización de las normas de gestor de colas

En este tema se describen algunos ejemplos de utilización de las normas de gestor de colas.

En la base de código en Java, un usuario proporciona una implementación de un método de normas mediante la creación de subclases de la clase `MQeQueueManagerRule`.

En la base de código en C, un usuario correlaciona las funciones de las normas con los punteros de función de las normas relevantes. Estos punteros se pasan a la función de inicialización de las normas, que es también el punto de entrada del módulo de normas del usuario.

Para obtener una descripción de todos los parámetros que se pasan a las funciones de las normas en la base de código en C, consulte la consulta de programación en C de MQe.

Ejemplo de norma de transferencia de mensajes: Este primer ejemplo muestra una norma de transferencia de mensajes que insiste en que cualquier mensaje que se transfiera a una cola utilizando este gestor de colas debe contener un campo de ID de mensaje de MQe:

Base de código Java

```
/* 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" );
    }
}
```

Base de código en C

```
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);
  }
}
```

Observe la manera en que se recupera la instancia del bloque de excepción desde la estructura del parámetro de salida y después se establece con los códigos de razón y de retorno adecuados. Así se comunica la función de la norma con la aplicación, lo que modifica el comportamiento de la aplicación.

Ejemplo de norma de obtención de mensajes:

La siguiente norma de ejemplo es una norma de obtención de mensajes que insiste en que debe suministrarse una contraseña antes de permitir que se procese una petición de obtención de mensajes en la cola llamada OutboundQueue. La contraseña se incluye como un campo en el filtro de mensaje que se ha pasado al método `getMessage()`.

Base de código Java

```
/* 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" );
           }
       }
   }
}
```

Base de código en C

```
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.
                        }
                    }
                }
            }
        }
    }
```

Esta norma es un ejemplo muy sencillo de cómo proteger una cola. Sin embargo, para una mayor seguridad, se recomienda que utilice un autentificador. Un autentificador permite que una aplicación cree listas de control de accesos y determine quién podrá obtener mensajes de las colas.

Ejemplo de norma de eliminación de colas:

Se llama a la siguiente norma de ejemplo cuando una petición de administración del gestor de colas intenta suprimir una cola. A la norma se le pasa una referencia del objeto para el proxy de la cola en cuestión. En este ejemplo, la norma comprueba el nombre de la cola que se pasa y si la cola se denomina PayrollQueue, se rechazará la petición de eliminar la cola.

Base de código Java

```
/* 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" );
    }
}
```

Base de código en C

Esta norma no se implementa en la base de código en C.

Normas de transmisión

Un mensaje que se transfiere a una cola remota que está definida como síncrona, se transmitirá inmediatamente. Los mensajes que se transfieren a las colas remotas definidas como asíncronas se almacenan en el gestor de colas local hasta el momento en que se active el gestor de colas para transmitirlos. Una aplicación puede activar directamente el gestor de colas. El proceso se puede modificar o supervisar mediante las normas de transmisión del gestor de colas.

Las normas de transmisión son un subconjunto de las normas del gestor de colas. Las dos normas que permiten controlar la transmisión de los mensajes son las siguientes:

`triggerTransmission()`

Esta norma determina si se permite la transmisión del mensaje en el momento en el que se

invoca la norma. Se puede utilizar para prohibir o permitir la transmisión de todos los mensajes, es decir, se permite la transmisión de todos los mensajes o de ninguno.

transmit()
Esta norma toma una decisión que permite la transmisión por colas de colas remotas asíncronas. Por ejemplo, permite que sólo se transmitan los mensajes de las colas que están consideradas como de alta prioridad. Sólo se invoca la norma `transmit()` si la norma `triggerTransmission()` devuelve un valor correcto.

Ejemplo de norma de transmisión de desencadenante

MQe invoca la norma `triggerTransmission` cuando se activa la transmisión. Esto sucede cuando el método o la función `triggerTransmission` del gestor de colas se llama de manera explícita desde una aplicación o una norma. Además, en la base de código en Java, la norma se puede invocar cuando un mensaje se transmite a otra cola asíncrona remota. El comportamiento de la norma predeterminada tanto en Java como en C permite que continúe el intento de transmitir los mensajes pendientes. Por ejemplo, es la norma predeterminada Java en com.ibm.mqe.MQeQueueManagerRule:

```
/* default trigger transmission rule -
      always allow transmission */
public boolean triggerTransmission(int noOfMsgs,
                  MQeFields msgFields ){
    return true;
}
```

El código de retorno de esta norma indica al gestor de colas si se han de transmitir o no los mensajes pendientes. Un código de retorno true significa ″transmitir″, mientras que un código de retorno false significa ″no transmitir ahora″.

El usuario puede alterar temporalmente el comportamiento predeterminado al implementar su propia norma triggerTransmission(). Una norma más compleja puede decidir si la transmisión se realiza inmediatamente o no según el número de mensajes que esperan ser transmitidos o las colas remotas asíncronas. El siguiente ejemplo muestra una norma que sólo permite que la transmisión continúe si hay más de 10 mensajes pendientes de transmisión.

Base de código Java

```
/* 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 */
    }
}
```

Base de código en C

```
/* 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);
            }
        }
```

Norma transmit

Sólo se invoca la norma `transmit()` si la norma `triggerTransmission()` permite la transmisión. Devuelve un valor de true o `MQERETURN_OK` . Se invoca la norma `transmit()` para cada definición de cola remota que contiene mensajes que esperan ser transmitidos. Esto significa que la norma puede decidir qué mensajes se deben transmitir cola a cola.

Una extensión razonable de esta norma puede permitir la transmisión de todos los mensajes en horas de poca actividad. Durante las horas de máxima actividad sólo se permite transmitir los mensajes de las colas de alta prioridad.

Norma transmit - ejemplo 1 en Java:

La norma de ejemplo siguiente solamente permite transmitir mensajes desde una cola si su prioridad predeterminada es mayor que 5. Si no se ha asignado una prioridad a un mensaje antes de transferirlo a una cola, se le asigna la prioridad predeterminada de la cola.

```
public boolean transmit( MQeQueueProxy queue )    {
    if ( queue.getDefaultPriority() > 5 )    {
        return (true);
    }
    else    {
        return (false);
    }
}
```

Norma transmit: ejemplo 1 en C:

La norma de ejemplo siguiente solamente permite transmitir mensajes desde una cola si su prioridad predeterminada es mayor que 5. Si no se ha asignado una prioridad a un mensaje antes de transferirlo a una cola, se le asigna la prioridad predeterminada de la cola.

```
/* 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);
    }
}
```

Ejemplo de norma transmit más complejo

En el ejemplo siguiente (en Java y en C) se presupone que la transmisión de los mensajes se realizará a través de una red de comunicaciones que cobra por la duración de la transmisión. También asume que hay una franja horaria de tarifa económica en que el coste por unidad es más bajo. Las normas bloquean la transmisión de los mensajes hasta la franja horaria de tarifa económica. Durante la franja horaria de tarifa económica, el gestor de colas se activa con regularidad.

Norma transmit - ejemplo 2 en Java:

El ejemplo siguiente asume que la transmisión de los mensajes se realizará a través de una red de comunicaciones que cobra por la duración de la transmisión. También asume que hay una franja horaria de tarifa económica en que el coste por unidad es más bajo. Las normas bloquean la transmisión de los mensajes hasta la franja horaria de tarifa económica. Durante la franja horaria de tarifa económica, el gestor de colas se activa con regularidad.

```
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 */
}
```

Las funciones cheapRatePeriodStart y cheapRatePeriodEnd definen la duración de esta franja horaria de tarifa económica. En este ejemplo, la franja horaria de tarifa económica se ha definido a partir de las 18:00 horas de la tarde hasta las 09:00 horas de la mañana siguiente.

La constante MILLISECS_BETWEEN_TRIGGER_TRANSMITS define el período de tiempo, en milisegundos, entre cada activación del gestor de colas. En este ejemplo, el intervalo de activación se ha definido en 15 segundos.

La activación del gestor de colas está manejada por una hebra subordinada que se activa al final del período triggerInterval. Si la hora actual está dentro de la franja horaria de tarifa económica, invoca el método MQeQueueManager.triggerTransmission() para iniciar un intento de transmisión de todos los mensajes que esperan ser transmitidos. La hebra subordinada se crea mediante la norma queueManagerActivate() y se detiene mediante la norma queueManagerClose(). El gestor de colas invoca estas normas cuando se activa y cierra respectivamente.

```
/**
* 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.
        }
    }
}
```

El código que maneja la hebra subordinada es similar al siguiente:

```
/**
* 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();
          }
        }
    }
}
```

El propietario de la variable se define mediante la clase `MQeRule`, que es el elemento superior de `MQeQueueManagerRule`. Como parte de su proceso de arranque, el gestor de colas activa las normas del gestor de colas y pasa una referencia a sí misma al objeto de normas. Dicha referencia se almacena en el propietario de la variable.

La hebra forma un bucle indefinidamente, puesto que la norma `queueManagerClose()` la detiene, y permanece inactiva hasta que finaliza el período del intervalo de `MILLISECS_BETWEEN_TRIGGER_TRANSMITS`. Al final de este intervalo, si no se le ha dicho que se detenga, invoca el método `timeToTransmit()` para comprobar si la hora actual se encuentra dentro del período de transmisión de tarifa económica. Si este método se ejecuta correctamente, se invoca la norma `triggerTransmission()` del gestor de colas. El método `timeToTransmit` se muestra en el código siguiente:

```
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 */
    }
}
```

Norma transmit: ejemplo 2 en C:

El ejemplo en C emula el ejemplo de base de código en Java. Aunque la base de código en C nativa está compuesta en su totalidad por una única hebra, se puede escribir el código específico de plataforma en el que se creen las hebras. En este ejemplo de una norma de activación de gestores de colas escrita por un usuario, se genera una hebra que forma bucles y está inactiva durante un período de tiempo que se define en una variable `triggerInterval` y, a continuación, siempre que no se le haya pedido que se detenga, comprueba que estamos en un período de tarifa económica antes de intentar activar la transmisión. Los datos, que son necesarios entre las invocaciones de normas, se almacenan en la estructura de datos privados de la norma. La función de norma de cierre del gestor de colas se utiliza para proporcionar la condición de terminación de la hebra, estableciendo un conmutador booleano, `carryOn`, en `MQE_FALSE`. Este conmutador se puede inicializar en `MQE_TRUE` en la función de inicialización de las normas. Esta función espera hasta que la hebra se suspende antes de devolver el control a la aplicación.

La estructura de datos privados que se pasa entre las invocaciones de normas es la siguiente:

```
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);
        }
    }
}
```

La función `timeToTrigger` proporciona las funciones equivalentes del método `run()` en el ejemplo Java. Observe la utilización de la variable de datos privados `carryOn` y escriba `MQEBOOL`, como una de las condiciones para que el bucle 'while' continúe. Cuando esta variable tenga un valor de `MQE_FALSE`, el bucle 'while' terminará, lo que hará que la hebra termine al salir de la función.

```
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;
}
```

La función `timeToTransmit()` devuelve un valor booleano para indicar si nos encontramos o no en un período de transmisión económico:

```
MQEBOOL timeToTransmit() {

    SYSTEMTIME timeInfo;
    GetLocalTime(&timeInfo);
```

```
    if (timeInfo.wHour >= 18 || timeInfo.wHour < 9) {
        return MQE_TRUE;
    } else {
        return MQE_FALSE;
    }
}
```

Probablemente sería mejor definir las constantes de las horas límite del intervalo de tarifa económica y transportarlas también a la estructura de datos privados de las normas, pero no se ha hecho aquí por razones de claridad.

La función devuelve MQE_TRUE para sugerir que nos encontramos en un período de tarifa económica, es decir, entre las 18:00 y las 09:00. Un valor de retorno de MQE_TRUE es uno de los requisitos previos para que se active la transmisión en timeToTrigger(). Por último, se utiliza la norma de cierre del gestor de colas para terminar la hebra. Tenga en cuenta que una de las condiciones de terminación de la función timeToTrigger() es que la variable booleana carryOn tenga un valor de MQE_FALSE. En la función de cierre, el valor de carryOn se establece en false. Sin embargo, todavía es posible que transcurra mucho tiempo entre el momento en que este valor se establece en MQE_FALSE y el momento en que se sale de la función timeToTrigger(). El valor de triggerInterval + el tiempo que se tarda en realizar una operación triggerTransmission. Además, esperamos a que la hebra termine en su función. También invocamos triggerTransmission() una vez más en caso de que todavía haya mensajes pendientes.

```
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);
 }
}
```

Activación de las definiciones de colas remotas asíncronas

El gestor de colas puede activar sus definiciones de colas remotas asíncronas y las colas del servidor local durante el arranque. En la base de código en Java, la activación de las definiciones de colas remotas asíncronas hace que se intenten transmitir todos los mensajes que contienen, mientras que la activación de colas del servidor local hace que se intenten obtener todos los mensajes que están esperando en la cola de almacenamiento y envío que tienen asignada. La norma activateQueues() permite la configuración de este comportamiento.

La norma predeterminada devuelve un valor 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;
    }
}
```

Si `activateQueues()` devuelve un valor false, las definiciones de colas remotas sólo se activan cuando se les transfiere un mensaje. Las colas del servidor local se pueden activar invocando el método `triggerTransmission()` del gestor de colas.

En la base de código en C, la activación de colas de servidores locales y colas asíncronas no hace que se intenten transmitir o desplegar los mensajes pendientes. Sólo las llamadas explícitas a la función `triggerTransmission()` del gestor de colas tienen este resultado. No hay ninguna implementación de la norma activateQueues en la base de código en C. La activación de las colas se produce cuando se inicia el gestor de colas.

Normas de colas

En la base de código en Java, cada cola tiene su propio conjunto de normas. Una solución puede ampliar el comportamiento de estas normas. Todas las normas de colas deben descender de la clase com.ibm.mqe.MQeQueueRule.

En la base de código en C, sólo se carga un conjunto de normas. Un usuario puede implementar diferentes normas para diferentes colas cargando otros módulos de normas del módulo 'maestro'. A continuación, las funciones de las normas maestras pueden invocar las funciones correspondientes en todos los demás módulos según sea necesario.

Se llama a las normas de colas cuando:

- Se activa la cola.
- Se cierra la cola.
- Se coloca un mensaje en la cola mediante una operación de transferencia (sólo base de código en Java).
- Se elimina un mensaje de la cola mediante una operación de obtener.
- Se suprime un mensaje de la cola mediante una operación de suprimir (sólo base de código en Java).
- Se examina la cola.
- Se efectúa una operación de deshacer en un mensaje de la cola.
- Se añade un escucha de mensajes a la cola (sólo base de código en Java).
- Se elimina un escucha de mensajes de la cola (sólo base de código en Java).
- Caduca un mensaje.
- Se intentan modificar los atributos de una cola, es decir, el autentificador, el cifrador, el compresor (sólo base de código en Java).
- Se transfiere un mensaje duplicado a una cola.
- Se transmite un mensaje desde una cola asíncrona remota.

Utilización de las normas de colas

Este apartado describe algunos ejemplos de utilización de las normas de colas.

En el primer ejemplo se muestra un uso posible de la norma de mensajes caducados y una copia del mensaje se transfiere a una cola de mensajes no entregados. Tanto las colas como los mensajes pueden tener fijado un intervalo de caducidad. Si este intervalo se supera, el mensaje se marcará como caducado. En este punto, se invoca la norma messageExpired(). Cuando se devuelve esta norma, se suprime el mensaje caducado.

En el primer ejemplo se envían los mensajes caducados a la cola de mensajes no entregados del gestor de colas, cuyo nombre se define mediante la constante `MQe.DeadLetter_Queue_Name` en la base de código en Java y `MQE_DEADLETTER_QUEUE_NAME` en la base de código en C. El gestor de colas rechaza la transferencia de un mensaje que se ha transferido anteriormente a otra cola. Esto evita que se introduzca un mensaje duplicado en la red de MQe. Por lo tanto, antes de pasar el mensaje a la cola de mensajes no entregados, la norma debe establecer su indicador de reenvío. Esto se lleva a cabo añadiendo el campo en Java `MQe.Msg_Resend` o en C `MQE_MSG_RESEND` al mensaje.

El campo de tiempo de caducidad del mensaje debe suprimirse antes de trasladar el mensaje a la cola de mensajes no entregados.

Normas de colas: ejemplo 1 en Java:

En este ejemplo se muestra un uso posible de la norma de mensajes caducados y una copia del mensaje se transfiere a una cola de mensajes no entregados. Tanto las colas como los mensajes pueden tener fijado un intervalo de caducidad. Si este intervalo se supera, el mensaje se marcará como caducado. En este punto, se invoca la norma messageExpired(). Cuando se devuelve esta norma, se suprime el mensaje caducado.

```
/* 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);
}
```

Normas de colas: ejemplo 1 en C:

En este ejemplo se muestra un uso posible de la norma de mensajes caducados y una copia del mensaje se transfiere a una cola de mensajes no entregados. Tanto las colas como los mensajes pueden tener fijado

un intervalo de caducidad. Si este intervalo se supera, el mensaje se marcará como caducado. En este punto, se invoca la norma messageExpired(). Cuando se devuelve esta norma, se suprime el mensaje caducado.

```
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);
                    }
                }
            }
        }
    }
}
```

Normas de colas: ejemplo 2 en Java:

El ejemplo siguiente muestra cómo anotar un suceso que se ha producido en la cola. El suceso que tiene lugar es la creación de un escucha de mensajes.

En el ejemplo la cola tiene su propio archivo de registro, pero del mismo modo se puede tener un archivo de registro central para que lo utilicen todas las colas. La cola tiene que abrir el archivo de registro durante su activación y cerrar el archivo de registro durante el cierre. Se pueden utilizar las normas de colas, `queueActivate` y `queueClose` para ello. Es necesario que la variable `logFile` sea una variable de clase de forma que ambas normas puedan acceder al archivo de registro.

```
/* 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 );
    }
}
```

La norma `addListener` se muestra en el código siguiente. Utiliza el método `MQe.log` para añadir un suceso `Event_Queue_AddMsgListener`.

```
/* 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() );
}
```

Normas de colas: ejemplo 2 en C:

El ejemplo siguiente muestra cómo anotar un suceso que se ha producido en la cola. El suceso que tiene lugar es una petición de transferencia de mensaje.

En este ejemplo, se configura un archivo central de registro para todas las colas que utilizan las normas de activación y cierre de colas. A continuación, este archivo de registro se utiliza para realizar un seguimiento de todas las operaciones putMessage. Puesto que el archivo de registro se comparte entre invocaciones de normas, la información necesaria para acceder al archivo de registro se almacena en la estructura de datos privados de las normas. En este caso, la estructura de datos privados contiene un manejador de archivo para pasar entre las invocaciones de normas:

```
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;
```

En la función de activación de colas de las normas , el archivo se abre y la activación de la cola se anota:

```
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);
             }
        }
    }
}
```

En la función de cierre de colas de las normas, el archivo se cierra después de anotar el cierre de la cola:

```
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;
    }
}
```

La función de transferencia de mensajes de las normas garantiza que se anotan todas las operaciones de transferencia de mensajes:

```
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);
}
```

Normas de puentes

Aunque las normas de colas también se pueden aplicar a las colas puente, se pueden aplicar también los siguientes tipos distintos de normas a los puentes:

UndeliveredMessageRules

Estas normas se pueden aplicar al escucha puente y se pueden utilizar para determinar la acción que se debe realizar cuando no se puede entregar un mensaje de MQ a la pasarela de MQe. La norma predeterminada que utiliza MQe detendrá el escucha puente después de un número establecido de intentos de entrega del mensaje. Se proporcionan dos normas de ejemplo:

examples.mqbridge.rules.MQeUndeliveredMessageRule

Copia de la norma predeterminada

examples.mqbridge.rules.UndeliveredMQMessageToDLQRule
Descartará el mensaje o lo moverá a la cola de mensajes no entregados de MQ en función del campo de informe del mensaje de MQ original.

Normas StartUp
Estas normas se pueden utilizar para controlar el inicio de los objetos contenidos en el puente para que, por ejemplo, el puente esté en estado detenido cuando se inicie la pasarela. Se proporciona un ejemplo: examples.mqbridge.rules.MQeStartupRule.

Normas SyncQueuePurger
Estas normas se pueden utilizar para fines administrativos a fin de borrar registros antiguos que a veces pueden quedar en el gestor de colas de MQ. No obstante, esto sólo suele suceder si el mensaje de MQe correspondiente se ha suprimido. Se proporcionan dos ejemplos:

examples.mqbridge.rules.MQeSyncQueuePurgerRule
Invoca el rastreo con una sentencia de información cuando descubre mensajes anteriores a la hora especificada.

examples.mqbridge.rules.DestructiveMQSyncQueuePurgerRule
Suprime los mensajes anteriores a la hora especificada.

Java Message Service (JMS)

Las clases de MQe para Java Message Service (JMS) son un conjunto de clases Java que implementan las interfaces JMS de Sun para permitir que los programas JMS accedan a sistemas MQe. En este tema se describe cómo utilizar las clases de MQe para JMS.

El release inicial de las clases JMS para MQe Versión 2.1, da soporte al modelo punto a punto de JMS, pero no da soporte al modelo de publicación o suscripción.

La utilización de JMS como API para escribir aplicaciones de MQe tiene una serie de ventajas, puesto que JMS es un estándar abierto:

- Protección de la inversión, tanto en conocimientos específicos como en código de aplicación
- Disponibilidad de personal formado para la programación de aplicaciones JMS
- Posibilidad de escribir aplicaciones de mensajería que son independientes de las implementaciones de JMS

Puede obtener más información acerca de la API de JMS en el sitio web de Sun, en la dirección siguiente: http://java.sun.com.

Utilización de JMS con MQe

En este apartado se describe cómo configurar el sistema para que ejecute los programas de ejemplo, incluido el ejemplo de Prueba de verificación de la instalación (IVT) que verifica la instalación de JMS para MQe.

Para utilizar JMS con MQe debe tener los siguientes archivos jar, además de MQeBase.jar, en la variable CLASSPATH:

jms.jar
Ésta es la definición de la interfaz de Sun para las clases JMS

MQeJMS.jar
Es la implementación de JMS en MQe.

Obtención de archivos jar

MQe no se suministra con la definición de la interfaz JMS de Sun, que se encuentra en el archivo jms.jar, sino que se tiene que descargar antes de utilizar JMS. En el momento de la escritura, se puede descargar de forma gratuita de http://java.sun.com/products/jms/docs.html. Se exige el archivo jar de JMS Versión 1.0.2b.

Además, si hay que almacenar y recuperar los objetos administrados por JMS mediante JNDI (Java Naming and Directory Interface), las clases javax.naming.* tienen que estar en la variable CLASSPATH. Si se utiliza Java 1, por ejemplo, un JRE 1.1.8, se tiene que obtener el archivo jndi.jar y añadirlo a la variable CLASSPATH. Si se utiliza Java, JRE 1.2 o posterior, es posible que el JRE contenga estas clases. Puede utilizar MQe sin JNDI, pero a cambio de depender un poco de un proveedor. Se tienen que utilizar clases específicas de MQe para los objetos ConnectionFactory y Destination. Puede descargarse los archivos jar de JNDI de http://java.sun.com/products/jndi.

Prueba de la variable classpath de JMS

Puede utilizar el programa de ejemplo `examples.jms.MQeJMSIVT` para probar la instalación de JMS. Antes de ejecutar este programa, necesita un gestor de colas de MQe que tenga una cola `SYSTEM.DEFAULT.LOCAL.QUEUE`. Además de los archivos jar de JMS que se han mencionado anteriormente, también necesita los siguientes archivos jar o equivalentes en la variable CLASSPATH para ejecutar `examples.jms.MQeJMSIVT`:

- MQeBase.jar
- MQeExamples.jar

Puede ejecutar el ejemplo desde la línea de mandatos escribiendo:

```
java examples.jms.MQeJMSIVT -i
 <nombre de archivo ini>
```

donde <nombre archivo ini> es el nombre del archivo de inicialización (ini) del gestor de colas de MQe. De manera opcional, puede añadir un distintivo ″-t″ para activar el rastreo:

```
java examples.jms.MQeJMSIVT -t -i
    <nombre de archivo ini>
```

El programa de ejemplo comprueba que los archivos jar necesarios se encuentren en la variable CLASSPATH comprobando las clases que contienen. El programa de ejemplo crea una *QueueConnectionFactory* y la configura utilizando el nombre de archivo ini que entró en la línea de mandatos. A continuación, inicia una conexión, que:

1. Inicia el gestor de colas de MQe
2. Crea una cola JMS que representa la cola `SYSTEM.DEFAULT.LOCAL.QUEUE` del gestor de colas
3. Envía un mensaje a la cola JMS
4. Vuelve a leer el mensaje y lo compara con el mensaje que envió

La cola `SYSTEM.DEFAULT.LOCAL.QUEUE` no debe contener ningún mensaje antes de ejecutar el programa, de lo contrario, la nueva lectura del mensaje no será la que el programa envió. La salida del programa debe ser similar a ésta:

```
utilización del archivo ini '<nombre de archivo .ini>'
      para configurar la conexión
se está comprobando la variable CLASSPATH
se han encontrado las clases de la interfaz de JMS
se han encontrado las clases JMS de MQe
se han encontrado las clases base de MQe
Se está creando y configurando QueueConnectionFactory
Se está creando la conexión
a partir de los datos de conexión, el proveedor de JMS
  es IBM MQe Versión 2.0.0.0
```

```
Se está creando la sesión
Se está creando la cola
Se está creando el emisor
Se está creando el receptor
Se está creando el mensaje
Se está enviando el mensaje
Se está recibiendo el mensaje


CAMPOS DE CABECERA
----------------------------------------
 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

 CAMPOS DE PROPIEDADES (sólo lectura)
----------------------------------------
 JMSXRcvTimestamp : 1032184400133

 CUERPO DEL MENSAJE (sólo lectura)
----------------------------------------
Un mensaje de texto sencillo del programa MQeJMSIVT

El mensaje recuperado es un mensaje de texto (TextMessage); ahora se están buscando
coincidencias con el mensaje enviado
Los mensajes son iguales. Bien.
Se está finalizando la conexión
conexión cerrada
IVT finalizada
```

Ejecución de otros programas de ejemplo de JMS en MQe

MQe proporciona otros dos programas de ejemplo para las clases JMS. El programa `examples.jms.PTPSample01` es parecido a los ejemplos de IVT que se han descrito anteriormente, pero existe un argumento de línea de mandatos para indicarle que no utilice la *Java Naming and Directory Interface* (JNDI) y no tiene las mismas comprobaciones en la variable CLASSPATH. El programa exige los mismos archivos jar de JMS y MQe en la variable CLASSPATH que `examples.jms.MQeJMSIVT`, es decir, jms.jar, MQeJMS.jar, MQeBase.jar y MQeExamples.jar. También requiere el archivo jndi.jar, aunque no utilice JNDI, porque el programa importa javax.naming. En el apartado acerca de la utilización de JNDI se ofrece más información acerca del archivo jndi.jar. Puede ejecutar el ejemplo desde la línea de mandatos escribiendo:

```
java examples.jms.PTPSample01 -nojndi -i <nombre de archivo ini>
```

donde <nombre archivo ini> es el nombre del archivo de inicialización (ini) del gestor de colas de MQe. De forma predeterminada, el programa utilizará la cola `SYSTEM.DEFAULT.LOCAL.QUEUE` en este gestor de colas. Puede especificar una cola diferente utilizando el distintivo -q:

```
java examples.jms.PTPSample01 -i <nombre de archivo ini> -q <nombre de cola>
```

También puede activar el rastreo añadiendo el distintivo -t:

```
java examples.jms.PTPSample01 -t -i <nombre de archivo ini> -q <nombre de cola>
```

El programa `examples.jms.PTPSample02` utiliza filtros y escuchas de mensajes. Este programa crea un receptor de colas (*QueueReceiver*) con un filtro ″azul″ y también un escucha de mensajes. Crea un segundo receptor de colas (QueueReceiver) con un filtro ″rojo″ y un escucha de mensajes. A continuación, envía cuatro mensajes a una cola, dos con el color de propiedad del filtro establecido en azul y dos con el color

de propiedad del filtro establecido en rojo, y comprueba que los escuchas de mensajes reciban los mensajes correctos. El programa tiene los mismos parámetros de línea de mandatos que `examples.jms.PTPSample01`.

Escritura de programas JMS

Presenta el modelo de JMS y proporciona información sobre cómo escribir aplicaciones JMS para MQe.

En este apartado se proporciona información sobre cómo escribir aplicaciones JMS para MQe. Ofrece una breve introducción del modelo JMS e información acerca de la programación de algunas tareas comunes que es probable que deban realizar los programas de aplicación.

El modelo de JMS

JMS define una vista genérica de un servicio de mensajes. Es importante que comprenda esta vista y cómo se correlaciona con el sistema de MQe subyacente. El modelo de JMS genérico se basa en las interfaces siguientes que se definen en el paquete javax.jms de Sun:

Connection
Proporciona una conexión con el servicio de mensajería subyacente y se utiliza para crear *Sessions* (sesiones).

Session
Proporciona un contexto para producir y consumir mensajes, incluidos los métodos que se utilizan para crear *MessageProducers* (productores de mensajes) y *MessageConsumers* (consumidores de mensajes).

MessageProducer
Se utiliza para enviar mensajes.

MessageConsumer
Se utiliza para recibir mensajes.

Destination
Representa el destino de un mensaje.

Nota: Una conexión es de hebra segura, pero las sesiones, los productores de mensajes y los consumidores de mensajes no lo son. Aunque la especificación de JMS permite que una sesión sea utilizada por más de una hebra, es responsabilidad del usuario asegurarse de que los recursos de la sesión no los utilicen varias hebras de manera simultánea. La estrategia que se recomienda consiste en utilizar una sesión por hebra de aplicación.

Por lo tanto, en términos de MQe:

Connection
Proporciona una conexión con un gestor de colas de MQe. Todas las conexiones de una JVM se deben conectar al mismo gestor de colas, puesto que MQe da soporte a un solo gestor de colas por JVM. La primera conexión creada por una aplicación intentará conectarse a un gestor de colas que ya se esté ejecutando y si no lo consigue intentará iniciar un gestor de colas por sí misma. Las conexiones posteriores se conectarán al mismo gestor de colas que la primera conexión.

Session
No tiene un equivalente en MQe.

Message producer y message consumer
No tienen equivalentes directos en MQe. MessageProducer invoca el método `putMessage()` del gestor de colas. MessageConsumer invoca el método `getMessage()` del gestor de colas.

Destination
Representa una cola de MQe.

JMS para MQe puede transferir mensajes a una cola local o a una cola remota asíncrona y puede recibir mensajes de una cola local. No puede transferir ni recibir mensajes de una cola remota síncrona.

Las interfaces JMS genéricas se subclasifican en versiones más específicas para el comportamiento de punto a punto y de publicación o suscripción. MQe implementa las subclases punto a punto de JMS. Las subclases punto a punto son:

QueueConnection
Amplía la conexión (Connection)

QueueSession
Amplía la sesión (Session)

QueueSender
Amplía el productor de colas (MessageProducer)

QueueReceiver
Amplía el consumidor de colas (MessageConsumer)

Queue
Amplía el destino

Se recomienda escribir programas de aplicación que sólo utilicen referencias a las interfaces de javax.jms. Toda la información específica del proveedor se encapsula en implementaciones de:

- QueueConnectionFactory
- Queue

Estas implementaciones se conocen como ″objetos administrados″, es decir, objetos que se pueden administrar y almacenar en un espacio de nombres JNDI. Una aplicación JMS puede recuperar estos objetos del espacio de nombres y utilizarlos sin necesidad de conocer el proveedor que ha proporcionado la implementación. Sin embargo, es posible que en dispositivos pequeños que buscan objetos en un espacio de nombres JNDI sea poco práctico o represente un gasto innecesario. Por lo tanto, se proporcionan dos versiones de las clases `QueueConnectionFactory` y `Queue`.

Las clases superiores, `MQeQueueConnectionFactory.class` y `MQeJMSQueue.class`, proporcionan la funcionalidad básica de JMS pero no se pueden almacenar en JNDI, mientras que las subclases MQeJNDIQueueConnectionFactory.class y MQeJMSJNDIQueue.class incorporan la funcionalidad necesaria para poderlas almacenar y recuperar de JNDI.

Creación de una conexión:

Normalmente, las conexiones no se crean directamente, sino mediante una fábrica de conexiones. Un espacio de nombres JNDI puede almacenar una fábrica configurada, aislando así la aplicación JMS de la información específica del proveedor. Consulte el apartado Utilización de JNDI, que aparece más adelante, para obtener información acerca de cómo almacenar y recuperar objetos mediante JNDI.

Si no hay ningún espacio de nombres JNDI disponible, puede crear objetos de fábrica durante la ejecución. No obstante, esto reduce la portabilidad de la aplicación JMS puesto que requiere referencias a clases específicas de MQe. El código siguiente crea una fábrica de conexión de colas (QueueConnectionFactory). La fábrica utiliza un gestor de colas de MQe que se configura con un archivo de inicialización (ini):

```
QueueConnectionFactory factory;
factory = new com.ibm.mqe.jms.MQeJNDIQueueConnectionFactory();
((com.ibm.mqe.jms.MQeJNDIQueueConnectionFactory)factory).
setIniFileName(<initialisation file>)
```

Utilización de la fábrica para crear una conexión:

Utilice createQueueConnection() para crear una conexión de colas (QueueConnection):

```
QueueConnection connection;
connection = factory.createQueueConnection();
```

Inicio de la conexión:

En la especificación de JMS, las conexiones no se activan al crearlas. Hasta que la conexión no se inicia, los consumidores de mensajes (MessageConsumers) que están asociados con la conexión no pueden recibir ningún mensaje. Utilice el siguiente mandato para iniciar la conexión:

```
connection.start();
```

Obtención de una sesión:

Después de crear una conexión, puede utilizar el método `createQueueSession()` de la conexión de colas (QueueConnection) para obtener una sesión. El método toma dos parámetros:

1. Un booleano que determina si se trata de una sesión ″ejecutada″ o ″no ejecutada″.
2. Un parámetro que determina la modalidad de ″reconocimiento″. Este parámetro se utiliza cuando la sesión es ″no ejecutada″.

El caso más sencillo es aquel en el que JMS utiliza y maneja reconocimientos con `AUTO_ACKNOWLEDGE`, tal como se muestra en el siguiente fragmento de código:

```
QueueSession session;
boolean transacted = false;
session = connection.createQueueSession(transacted, Session.AUTO_ACKNOWLEDGE);
```

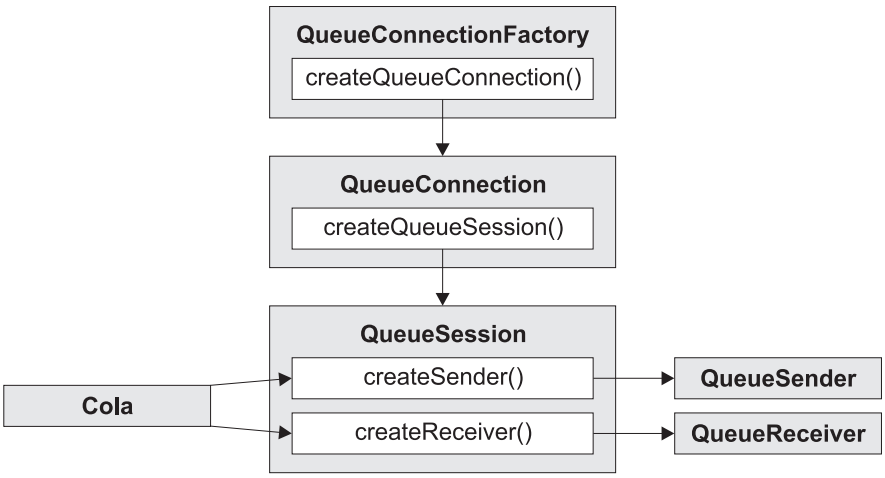

Figura 73. Obtención de una sesión después de crear una conexión

Envío de un mensaje:

Los mensajes se envían mediante un productor de mensajes (MessageProducer). Para el modelo punto a punto se trata de un emisor de colas (QueueSender) que se crea mediante el método `createSender()` en la sesión de colas (QueueSession). Normalmente, se crea un emisor de colas (QueueSender) para una determinada cola (Queue), de modo que todos los mensajes que se han enviado mediante ese emisor se envían al mismo destino. Los objetos de cola se pueden crear durante la ejecución o bien se pueden crear y almacenar en un espacio de nombres JNDI. Consulte “Utilización de JNDI (Java Naming and Directory Interface)” en la página 154, para obtener información detallada sobre cómo almacenar y recuperar objetos mediante JNDI.

JMS proporciona un mecanismo para crear una cola (Queue) durante la ejecución que minimiza el código específico de la implementación en la aplicación. Este mecanismo utiliza el método QueueSession.createQueue(), que toma un parámetro de serie de caracteres que describe el destino. En sí misma, la serie de caracteres todavía se encuentra en un formato específico de la implementación, pero es una propuesta mucha más flexible que si se hace referencia directamente a las clases de implementación.

Para JMS en MQe, la serie de caracteres es el nombre de la cola de MQe. De manera opcional puede contener el nombre del gestor de colas. Si se incluye el nombre del gestor de colas, el nombre de cola se separa del nombre del gestor de colas mediante un símbolo de suma '+', por ejemplo:

```
ioQueue = session.createQueue("myQM+myQueue");
```

Esto creará una cola de JMS que representa la cola de MQe "myQueue" en el gestor de colas "myQM". Si no se especifica ningún nombre de gestor de colas, se utiliza el gestor de colas local, es decir, el único al que está conectado JMS. Por ejemplo:

```
String queueName = "SYSTEM.DEFAULT.LOCAL.QUEUE";

...

ioQueue = session.createQueue(queueName);
```

Esto creará una cola de JMS que representa la cola de MQe `SYSTEM.DEFAULT.LOCAL.QUEUE` en el gestor de colas que utiliza la conexión de JMS.

Tipos de mensajes:

JMS proporciona varios tipos de mensajes y, cada uno de ellos, incorpora alguna información de su contenido. Para evitar que se haga referencia a los nombres de clase específicos de la implementación para los tipos de mensaje, en el objeto Session se facilitan métodos para la creación de mensajes. En el programa de ejemplo, se crea un mensaje de texto del modo siguiente:

```
System.out.println("Creating a TextMessage");
TextMessage outMessage = session.createTextMessage();
System.out.println("Adding Text");
outMessage.setText(outString);
```

Los tipos de mensajes que se pueden utilizar son:

- BytesMessage
- ObjectMessage
- TextMessage

Recepción de un mensaje:

Los mensajes se reciben mediante un receptor de colas (QueueReceiver). Se crea a partir de una sesión (Session) mediante el método `createReceiver()`. Este método toma un parámetro de cola (Queue) que define desde donde se reciben los mensajes. Consulte el apartado anterior "Envío de un mensaje" para obtener información detallada sobre cómo crear un objeto de cola (Queue). En el programa de ejemplo se crea un receptor y se puede leer el mensaje de prueba con el código siguiente:

```
QueueReceiver queueReceiver = session.createReceiver(ioQueue);
Message inMessage = queueReceiver.receive(1000);
```

El parámetro de la llamada de recepción es un tiempo de espera en milisegundos. Este parámetro define el tiempo que debe esperar el método si no hay ningún mensaje disponible inmediatamente. Puede omitir este parámetro, en cuyo caso, la llamada se bloquea de modo indefinido. Si no desea que haya retardo, utilice el método `receiveNoWait()`. Los métodos de recepción reciben un mensaje del tipo adecuado. Por ejemplo, si se transfiere un mensaje de texto (TextMessage) a una cola, cuando se recibe el mensaje el objeto que se devuelve es una instancia de un mensaje de texto (TextMessage). Para extraer el contenido del cuerpo del mensaje, se debe efectuar una transformación CAST desde la clase Message genérica, que

es el tipo de retorno declarado de los métodos de recepción, a la subclase más específica como, por ejemplo, TextMessage. Si el mensaje recibido es de tipo desconocido, puede utilizar el operador ″instanceof″ para determinar de qué tipo es. Por lo general, suele ser conveniente probar la clase del mensaje antes de enviarlo ya que, de este modo, los errores inesperados pueden manejarse correctamente. En el código siguiente se muestra la utilización de ″instanceof″ y la extracción del contenido de un mensaje de texto (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");
}
```

Manejo de errores:

Las excepciones informan de todos los errores de ejecución de una aplicación JMS. La mayoría de los métodos de JMS generan JMSExceptions para indicar errores. Una práctica de programación recomendada consiste en capturar estas excepciones y manejarlas adecuadamente. A diferencia de las excepciones normales de Java, una JMSException puede incluir una excepción adicional. Para JMS, puede constituir una forma muy adecuada de pasar detalles importantes del transporte subyacente. Cuando se genera una JMSException como resultado de la emisión de una excepción por parte de MQe, generalmente ésta se muestra como una excepción incluida en la JMSException. La implementación estándar de JMSException no incorpora la excepción incluida en la salida de su método `toString()`. Por consiguiente, se debe comprobar de forma explícita si hay una excepción incluida e imprimirla, tal como se muestra en el fragmento siguiente:

```
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);
    }
}
```

Escucha de excepciones:

Para la entrega de mensajes asíncrona, el código de la aplicación no puede capturar las excepciones que se emiten al producirse anomalías en la recepción de mensajes. Esto se debe a que el código de la aplicación no realiza llamadas explícitas a métodos `receive()`. Para hacer frente a esta situación, se puede registrar un escucha de excepciones (ExceptionListener), que es una instancia de una clase que implementa el método `onException()`. Cuando se produce un problema grave, se invoca este método, en el que se pasa la `JMSException` como único parámetro. Puede obtener información más detallada en la documentación de JMS de Sun.

Mensajes de JMS:

Los mensajes JMS constan de las siguientes partes:

Cabecera
: Todos los mensajes dan soporte al mismo conjunto de campos de cabecera. Los campos de cabecera contienen valores que utilizan tanto los clientes como los proveedores para identificar y direccionar mensajes.

Propiedades
: Cada mensaje contiene un recurso incorporado para ofrecer soporte para los valores de propiedad que define la aplicación. Las propiedades facilitan un mecanismo eficaz para filtrar los mensajes que define la aplicación.

Cuerpo

JMS define varios tipos de cuerpos de mensaje que abarcan la mayoría de los estilos de envío de mensajes que se utilizan actualmente. JMS define cinco tipos de cuerpos de mensaje:

Texto Mensaje que contiene una `java.lang.String`

Objeto
Mensaje que contiene un objeto Java serializable

Bytes Corriente de datos de bytes no interpretados para codificar un cuerpo de modo que coincida con un formato de mensaje existente

Corriente de datos
Corriente de valores primitivos de Java que se rellenan y leen de manera secuencial y que no reciben soporte en esta versión de JMS para MQe.

Correlación
Conjunto de pares nombre-valor, donde los nombres son series de caracteres y los valores son tipos primitivos de Java. Se puede acceder a las entradas de modo secuencial o aleatorio por nombre. El orden de las entradas no está definido. En esta versión de JMS para MQe no se da soporte a la correlación.

El campo de cabecera `JMSCorrelationID` se utiliza para enlazar un mensaje con otro. Generalmente, enlaza un mensaje de respuesta con su mensaje de solicitud.

Selectores de mensajes:

Un mensaje contiene un recurso incorporado para ofrecer soporte para los valores de propiedad que define la aplicación. De hecho, proporciona un mecanismo que permite añadir campos de cabecera específicos de la aplicación a un mensaje. Las propiedades permiten que una aplicación tenga, a través de los selectores de mensajes, un proveedor de JMS que seleccione o filtre mensajes en su nombre, utilizando criterios específicos de la aplicación. Las propiedades que define la aplicación deben cumplir las normas siguientes:

- Los nombres de propiedad deben cumplir las normas de un identificador de selector de mensajes.
- Los valores de propiedad pueden ser boolean, byte, short, int, long, float, double y string.
- Los prefijos de nombre JMSX y JMS_ están reservados.

Los valores de las propiedades se establecen antes de enviar un mensaje. Cuando un cliente recibe un mensaje, las propiedades del mensaje son de sólo lectura. Si un cliente intenta establecer propiedades en este punto, se genera una MessageNotWriteableException. Si se invoca `clearProperties()`, las propiedades pueden ser de lectura y escritura.

Un valor de propiedad puede duplicar un valor del cuerpo de un mensaje, o puede no hacerlo. JMS no define ninguna política que determine qué debe o no convertirse en una propiedad. Sin embargo, para obtener un mejor rendimiento, las aplicaciones sólo deben utilizar propiedades del mensaje cuando sea necesario personalizar la cabecera de un mensaje. La razón fundamental para ello es ofrecer soporte para la selección personalizada de mensajes. Un selector de mensajes JMS permite que un cliente especifique los mensajes en los que está interesado utilizando la cabecera de mensaje. Sólo se entregan los mensajes cuyas cabeceras coinciden con el selector. Los selectores de mensajes no pueden hacer referencia a valores del cuerpo del mensaje. Un selector de mensajes coincide con un mensaje cuando el selector se evalúa como verdadero (true) cuando se sustituyen los valores de propiedad y el campo de cabecera del mensaje por sus identificadores correspondientes en el selector.

Un selector de mensajes es una serie de caracteres, que puede contener:

Literales

- Un literal de serie de caracteres se incluye entre comillas simples. Una comilla simple doble representa una comilla simple. Por ejemplo, 'literal' y, para indicar el genitivo sajón inglés, 'literal''s'. Como en los literales de serie de caracteres de Java, utilizan la codificación de caracteres Unicode.
- Un literal numérico exacto es un valor numérico sin coma decimal como, por ejemplo, 57, -957, +62. Se ofrece soporte para los números del rango de número largo de Java.
- Un literal numérico aproximado es un valor numérico en notación científica como, por ejemplo, 7E3 o -57.9E2, o un valor numérico con un decimal, como, por ejemplo, 7, -95,7 o +6,2. Se ofrece soporte para los números del rango doble de Java. Tenga en cuenta que los errores de redondeo pueden afectare l funcionamiento de los selectores de mensajes incluidos los literales numéricos aproximados.
- Los literales booleanos `TRUE` y `FALSE`.

Identificadores

- Un identificador es una secuencia de longitud ilimitada de letras Java y dígitos Java, en la que en primer lugar debe haber una letra Java. Una letra es cualquier carácter para el que el método `Character.isJavaLetter` devuelve un valor `true`. Esto incluye "_" y "$". Una letra o dígito es cualquier carácter para el que el método `Character.isJavaLetterOrDigit` devuelve un valor `true`.
- Los nombres `NULL`, `TRUE` o `FALSE` no pueden ser identificadores.
- `NOT`, `AND`, `OR`, `BETWEEN`, `LIKE`, `IN` e `IS` no pueden ser identificadores.
- Los identificadores son referencias de campo de cabecera o referencias de propiedades.
- Los identificadores son sensibles a las mayúsculas y las minúsculas.
- Las referencias de los campos de cabecera de mensaje están restringidas a:
  - JMSDeliveryMode
  - JMSPriority
  - JMSMessageID
  - JMSTimestamp
  - JMSCorrelationID
  - JMSType

  Los valores JMSMessageID, JMSTimestamp, JMSCorrelationID y JMSType pueden ser nulos y, en este caso, se tratan como un valor NULL.
- Todos los nombres que empiecen por "JMSX" son nombres de propiedades definidos por JMS.
- Todos los nombres que empiecen por "JMS_" son nombres de propiedades específicos del proveedor.
- Todos los nombres que no empiecen por "JMS" son nombres de propiedades específicos de la aplicación.
- Si hay alguna referencia a una propiedad que no exista en un mensaje, su valor es NULL. Si existe, su valor es el de la propiedad correspondiente.

Espacio en blanco

Es el mismo que se ha definido para Java: espacio, separador horizontal, salto de página y terminador de línea.

Operadores lógicos

Actualmente sólo se da soporte a `AND`.

Operadores de comparación

- Actualmente, sólo se da soporte al operador de igual ('=').
- Sólo se pueden comparar valores del mismo tipo.
- Si se intentan comparar diferentes tipos, el selector siempre es falso.
- Dos series de caracteres son iguales si contienen la misma secuencia de caracteres.

- El operador de comparación `IS NULL` comprueba una valor de campo de cabecera nulo o un valor de propiedad no encontrado. No se da soporte al operador de comparación `IS NOT NULL`.

Tenga en cuenta que actualmente no se da soporte a los operadores aritméticos.

El siguiente selector de mensajes selecciona mensajes con un tipo de mensaje de coche (car) y color azul (blue): `"JMSType ='car' AND colour ='blue'"`

Al seleccionar campos de cabecera, MQe interpretará literales numéricos exactos de modo que coincidan con el tipo del campo en cuestión, es decir, un selector que compruebe los campos de cabecera JMSPriority o JMSDeliveryMode interpretará un literal numérico exacto como int, mientras que un selector que compruebe JMSExpiration o JMSTimestamp interpretará un literal numérico exacto como número largo. Sin embargo, al seleccionar las propiedades del mensaje, MQe siempre interpretará un literal numérico exacto como un número largo y un literal numérico exacto como un doble. Por lo tanto, las propiedades específicas de la aplicación que se van a utilizar para la selección de mensajes se deben establecer mediante los métodos `setLongProperty` y `setDoubleProperty` respectivamente.

Limitaciones en esta versión de MQe

Esta versión de JMS para MQe implementa el subconjunto de punto a punto de JMS con un par de limitaciones. No implementa ninguna de las clases opcionales:

- Las clases ConnectionConsumer, ServerSession y ServerSessionPool del servidor de aplicaciones
- Las clases XA:
  - XAConnection
  - XAConnectionFactory
  - XAQueueConnection
  - XAQueueConnectionFactory
  - XAQueueSession
  - XASession
  - XATopicConnection
  - XATopicConnectionFactory
  - XATopicSession

No implementa la clase TemporaryQueue, lo que significa que la clase QueueRequestor no funcionará ni tampoco las clases MapMessage y StreamMessage.

En la clase `QueueConnectionFactory`, no se implementa el método `createQueueConnection()` que toma un nombre de usuario y una contraseña como parámetros, puesto que MQe no tiene el concepto de usuario. Se implementa el método sin parámetros.

Cuando se lee un mensaje de una cola pero no se reconoce, éste se devuelve a la cola para volverlo a entregar. En este caso, se debe establecer el campo de cabecera JMSRedelivered en el mensaje. JMS para MQe no establece este campo de cabecera.

JMS para MQe puede transferir mensajes a una cola local o a una cola remota asíncrona y puede recibir mensajes de una cola local. No puede colocar ni recibir mensajes de una cola remota síncrona.

Utilización de JNDI (Java Naming and Directory Interface)

Una de las ventajas que supone utilizar JMS es la posibilidad de escribir aplicaciones que son independientes de las implementaciones de JMS, lo que le permite conectar una implementación de JMS que sea adecuada a su entorno. Sin embargo, hay que configurar determinados objetos JMS de manera específica a la implementación de JMS que haya elegido. Estos objetos son las fábricas de conexiones, los

destinos y las colas, a menudo denominados ″objetos administrados″. Para mantener los programas de aplicación independientes de la implementación de JMS, hay que configurar estos objetos fuera de los programas de aplicación. Normalmente, se configurarán y almacenarán en un espacio de nombres JNDI. La aplicación buscará los objetos en el espacio de nombres y podrá utilizarlos directamente, puesto que ya se han configurado.

Es posible que haya situaciones como, por ejemplo, en un dispositivo pequeño, en las que no sea deseable utilizar JNDI. En estos casos los objetos se podrían configurar directamente en la aplicación. No utilizar JNDI implicaría que la aplicación dependiera en cierta medida de la implementación.

Almacenamiento y recuperación de objetos con JNDI

Antes de utilizar JNDI para almacenar o recuperar objetos, hay que configurar un ″contexto inicial″, tal como se muestra en este fragmento que se ha tomado del programa de ejemplo MQeJMSIVT_JNDI:

```
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 );
```

donde:

icf define una clase de fábrica para el contexto JNDI. Esto depende del proveedor JNDI que utilice. La documentación que entrega el proveedor JNDI debe indicarle qué valor tiene que utilizar para esto. Consulte también los ejemplos que aparecen a continuación.

url define la ubicación del espacio de nombres. Esto dependerá del tipo de espacio de nombres que utilice. Si utiliza el sistema de archivos, será un url de archivo que identifique un directorio de su sistema de archivos. Si utiliza LDAP, será un url de ldap que identifique un servidor LDAP y su ubicación en el árbol de directorios de dicho servidor. La documentación que entrega el proveedor JNDI debe describir el formato correcto del url.

Para obtener información más detallada sobre la utilización de JNDI, consulte la documentación de JNDI de Sun.

Nota: Algunas combinaciones de paquetes JNDI y suministradores de servicio LDAP pueden producir un error LDAP 84. Para resolver el problema, inserte la línea siguiente antes de la llamada a InitialContext.

```
environment.put(Context.REFERRAL,"throw");
```

Después de obtener un contexto inicial, los objetos se pueden almacenar y recuperar del espacio de nombres. Para almacenar un objeto, utilice el método `bind()`:

```
ctx.bind(nombre_entrada, objeto);
```

donde 'nombre_entrada' es el nombre con el que quiere almacenar el objeto y 'objeto' es el objeto que hay que almacenar. Por ejemplo, para almacenar una fábrica con el nombre ″ivtQCF″:

```
ctx.bind("ivtQCF", factory);
```

Para almacenar un objeto en un espacio de nombres JNDI, el objeto tiene que satisfacer la interfaz javax.naming.Referenceable o bien la interfaz java.io.Serializable, según el proveedor JNDI que utilice. Las clases `MQeJNDIQueueConnectionFactory` y `MQeJMSJNDIQueue` implementan estas dos interfaces. Para recuperar un objeto del espacio de nombres, utilice el método `lookup()`:

```
object = ctx.lookup(nombre_entrada);
```

donde nombre_entrada es el nombre con el que quiere almacenar el objeto. Por ejemplo, para recuperar una QueueConnectionFactory almacenada con el nombre ″ivtQCF″:

```
QueueConnectionFactory factory;
 factory = (QueueConnectionFactory)ctx.lookup("ivtQCF");
```

Utilización de los programas de ejemplo con JNDI

Se puede utilizar el programa de ejemplo examples.jms.MQeJMSIVT_JNDI para probar la instalación mediante JNDI. Es muy parecido al programa examples.jms.MQeJMSIVT, con la excepción de que utiliza JNDI para recuperar la fábrica de conexiones y la cola que utiliza. Antes de ejecutar este programa, tiene que almacenar estos dos objetos administrados en un espacio de nombres JNDI:

Tabla 4. Objetos administrados de un espacio de nombres JNDI

Nombre de entrada	Clase Java	Descripción
ivtQCF	MQeJNDIQueueConnectionFactory	QueueConnectionFactory configurada para utilizar un gestor de colas de MQe
ivtQ	MQeJMSJNDIQueue	Cola configurada para representar una cola de MQe que sea local para el gestor de colas que utiliza la entrada ivtQCF

Para crear estas entradas, se puede utilizar el programa examples.jms.CreateJNDIEntry o la herramienta MQeJMSAdmin, que se explica en el siguiente apartado. Las instalaciones más grandes pueden tener un directorio *Lightweight Directory Access Protocol (LDAP)* disponible, pero para las instalaciones más pequeñas es posible que sea más adecuado un espacio de nombres del sistema de archivos. Cuando haya decidido un espacio de nombres, deberá obtener los archivos de la clase JNDI correspondientes para dar soporte al espacio de nombres y añadirlos a la variable CLASSPATH. Éstos variarán según el espacio de nombres que haya elegido y la versión de Java que utilice.

Siempre debe tener las clases javax.naming.* en la variable CLASSPATH. Si utiliza Java 1 (por ejemplo, JRE 1.1.8), tiene que obtener una copia del archivo jndi.jar y añadirlo a la variable CLASSPATH. Si utiliza Java 2 (JRE 1.2 o posterior), JRE puede contener estas clases.

Si desea utilizar un directorio LDAP, tiene que obtener las clases JNDI que dan soporte a LDAP, por ejemplo, ldap.jar de Sun o ibmjndi.jar de IBM, y añadirlas a la variable CLASSPATH. Es posible que algunos JRE de Java 2 ya contengan las clases de Sun para LDAP. Consulte también la sección siguiente sobre el soporte de LDAP para las clases Java.

Si desea utilizar un directorio de sistemas de archivos, tiene que obtener las clases JNDI que dan soporte al sistema de archivos, por ejemplo, fscontext.jar de Sun (que también requiere providerutil.jar) y añadirlas a la variable CLASSPATH. El programa de ejemplo CreateJNDIEntry requiere que el archivo MQeJMS.jar se encuentre en la variable CLASSPATH, además de los archivos jar de JNDI. Toma los siguientes argumentos de la línea de mandatos:

```
java examples.jms.CreateJNDIEntry -url<providerURL>
    [-icf<initialContextFactory>][-ldap]
     [-qcf<entry name><MQe queue manager ini file>]
    [-q<entry name><MQe queue name>]
```

Otro argumento que se puede utilizar es:

```
java  examples.jms.CreateJNDIEntry -h
```

En los dos ejemplos anteriores:

-url<providerURL>
El URL del contexto inicial de JNDI (parámetro obligatorio)

-icf<initialContextFactory>
La fábrica de contexto inicial de JNDI que toma de forma predeterminada el sistema de archivos: `com.sun.jndi.fscontext.RefFSContextFactory`

-ldap Se debe especificar si utiliza un directorio LDAP

-qcf<entry name><MQe queue manager ini file>
El nombre de una entrada de JNDI que hay que crear para una `QueueConnectionFactory` de JMS y el nombre del archivo de inicialización (ini) de un gestor de colas de MQe que hay que utilizar para configurarla.

-h Muestra un mensaje de ayuda

Se tiene que especificar el url, -url, y también una QueueConnectionFactory (-qcf) o una Queue (-q), o bien ambas. La fábrica de contexto, -icf, es opcional y toma de forma predeterminada un directorio del sistema de archivos. Se debe especificar el distintivo LDAP, -ldap, si se utiliza un directorio LDAP, lo que añade al nombre de entrada un prefijo ″cn=″, que es necesario para LDAP.

Por ejemplo, si existe un gestor de colas con el archivo de inicialización *d:\MQe\exampleQM\exampleQM.ini* y utiliza un directorio JNDI basado en el sistema de archivos de *d:\MQe\data\jndi*, escriba (todo en una línea):

```
java  examples.jms.CreateJNDIEntry -url file://d:/MQe/data/jndi -qcf  ivtQCF
d:\MQe\exampleQM\exampleQM.ini
```

Tenga en cuenta que se utilizan barras inclinadas en el url, aunque el sistema de archivos utilice barras invertidas. El directorio url ya debe existir. Para añadir una entrada de la cola, debería escribir (todo en una línea):

```
java  examples.jms.CreateJNDIEntry -url file://
d:/MQe/data/jndi -q ivtQ SYSTEM.DEFAULT.LOCAL.QUEUE
```

Podría utilizar otra cola local en vez de la cola SYSTEM.DEFAULT.LOCAL.QUEUE.

También podría especificar el nombre de cola como `GC_ejemplo+SYSTEM.DEFAULT.LOCAL.QUEUE`, donde `GC_ejemplo` es el nombre del gestor de colas. Si no se especifica el nombre del gestor de colas, se utiliza el gestor de colas local.

Las dos entradas se pueden añadir al mismo tiempo escribiendo:

```
java  examples.jms.CreateJNDIEntry
             -url file://d:/MQe/data/jndi -qcf ivtQCF
 d:\MQe\exampleQM\exampleQM.ini -q ivtQ
                SYSTEM.DEFAULT.LOCAL.QUEUE
```

De nuevo, debe escribir todo este mandato en una línea. Se puede añadir un máximo de una fábrica de conexiones y una cola a la vez.

Cuando se hayan creado las entradas de JNDI, podrá ejecutar el programa `example .jms.MQeJMSIVT_JNDI`. Este programa requiere que en la variable CLASSPATH se encuentren los mismos archivos jar que tiene el programa MQeJMSIVT, es decir:

- `jms.jar`, la definición de la interfaz de Sun para las clases JMS
- `MQeJMS.jar`, la implementación de JMS en MQe
- `MQeBase.jar`
- `MQeExamples.jar`

También requiere los archivos jar de JNDI, tal como se utilizan para el programa de ejemplo `CreateJNDIEntry`. Se puede ejecutar el ejemplo desde la línea de mandatos escribiendo:

```
java  examples.jms.MQeJMSIVT_JNDI
     -url<providerURL>
```

donde <providerURL> es el URL especificado del contexto inicial de JNDI. De forma predeterminada el programa utiliza el contexto del sistema de archivos de JNDI:

```
com.sun.jndi.fscontext.RefFSContextFactory
```

Si es necesario, puede especificar un contexto alternativo:

```
java  examples.jms.MQeJMSIVT_JNDI -url<providerURL>
    -icf<initialContextFactory>
```

De manera opcional, puede añadir un distintivo `-t` para activar el rastreo:

```
java  examples.jms.MQeJMSIVT_JNDI -url<providerURL>
    -icf<initialContextFactory> -t
```

Para utilizar las entradas del directorio del sistema de archivos creadas en el ejemplo CreateJNDIEntry anterior, escriba:

```
java examples.jms.MQeJMSIVT_JNDI -url file://d:/MQe/data/jndi
```

El programa de ejemplo comprueba que los archivos jar necesarios se encuentren en la variable CLASSPATH comprobando las clases que contienen. Busca en la fábrica de conexiones de colas (QueueConnectionFactory) y en la cola (Queue) del directorio JNDI. Inicia una conexión, que a su vez inicia el gestor de colas de MQe, envía un mensaje a la cola (Queue), vuelve a leer el mensaje y lo compara con el mensaje que se envió. La cola no debe contener ningún mensaje antes de ejecutar el programa, de lo contrario, la nueva lectura del mensaje no será la que el programa envió. Las primeras líneas de la salida del programa deben ser similares a éstas:

```
se está utilizando la fábrica de contexto
         'com.sun.jndi.fscontext.RefFSContextFactory' para el directorio
se está utilizando el url del directorio 'file://d:/MQe/data/jndi'
se está comprobando la variable CLASSPATH
se han encontrado las clases de la interfaz de JMS
se han encontrado las clases JMS de MQe
se han encontrado las clases base de MQe
se han encontrado las clases jndi.jar
se han encontrado las clases com.sun.jndi.fscontext.RefFSContextFactory
Se está buscando la fábrica de conexiones en jndi
Se está buscando la cola en jndi
Se está creando la conexión
```

El resto de la salida debe ser similar a la del ejemplo sin JNDI. También puede ejecutar los otros dos programas de ejemplo, `examples.jms.PTPSample01` y `example .jms.PTPSample02`, utilizando JNDI. Estos programas exigen que en la variable CLASSPATH se encuentren los mismos archivos jar de JMS y MQe que para el programa `MQeJMSIVT_JNDI`, es decir:

- `jms.jar`
- `MQeJMS.jar`
- `MQeBase.jar`
- `MQeExamples.jar`

También requieren el archivo jndi.jar y los archivos jar del proveedor JNDI que utilice, por ejemplo, el sistema de archivos o LDAP. Los ejemplos se pueden ejecutar desde la línea de mandatos escribiendo:

```
java  examples.jms.PTPSsample01 -url<providerURL>
```

Como en el ejemplo anterior, `URL_proveedor` es el URL del contexto inicial de JNDI. De forma predeterminada, el programa utiliza el contexto del sistema de archivos para JNDI, es decir, `com.sun.jndi.fscontext.RefFSContextFactory`. Si es necesario, puede especificar un contexto alternativo:

```
java examples.jms.PTPSsample01 -url<providerURL>
              -icf<initialContextFactory>
```

De manera opcional, puede añadir un distintivo ″-t″ para activar el rastreo: java examples.jms. PTPSsample01 -url <providerURL><-icf initialContextFactory> -t . Para utilizar las entradas del directorio del sistema de archivos creadas en el ejemplo CreateJNDIEntry anterior, debería escribir:

```
java examples.jms.PTPSample01 -url file://d:/MQe/data/jndi
```

El programa `examples.jms.PTPSample02` utiliza filtros y escuchas de mensajes. Crea un `QueueReceiver` con un filtro ″colour=’blue’″ y también un escucha de mensajes. Crea un segundo `QueueReceiver` con un filtro ″colour=’red’″ así como un escucha de mensajes. Envía cuatro mensajes a una cola, dos con la propiedad ″colour″ establecida en ″red″ y dos con la propiedad ″colour″ establecida en ″blue″, y comprueba que los escuchas de mensajes reciban los mensajes correctos. El programa tiene los mismos parámetros de línea de mandatos que el programa `PTPSample01` y se puede ejecutar de la misma manera. Simplemente sustituya `PTPSample02` por `PTPSample01`.

Correlación de mensajes de JMS con mensajes de MQe

En este apartado se describe cómo se correlaciona la estructura de mensajes de JMS con un mensaje de MQe. Puede resultar interesante para los programadores que deseen transmitir mensajes entre aplicaciones de MQe tradicionales y JMS.

Tal como se ha descrito anteriormente, la especificación de JMS define un formato de mensaje estructurado que se compone de una cabecera, tres tipos de propiedades y cinco tipos de cuerpos de mensajes, mientras que MQe define un solo objeto de mensaje sin formato, `MQeMsgObject`. MQe define algunos nombres de campos de constantes que necesitan las aplicaciones de mensajería, como UniqueID, MessageID y Priority, mientras que las aplicaciones pueden transferir datos a un mensaje de MQe como pares `<nombre, valor>`.

Para enviar mensajes de JMS mediante MQe, se define un formato de constante para almacenar la información que contiene un mensaje de JMS en un `MQeMsgObject`. Así se añaden tres campos de nivel superior y cuatro objetos `MQeFields` a un `MQeMsgObject`, tal como se muestra en el siguiente ejemplo.

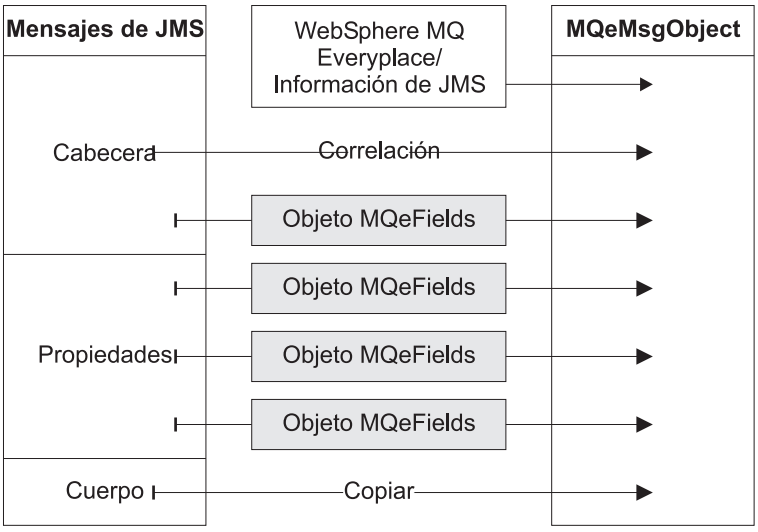

Figura 74. Correlación de un mensaje de JMS con un MQeMQeMsgObject

En los siguientes apartados se describe el contenido de estos campos:

Denominación de campos MQeMsgObject

Un `MQeMsgObject` almacena los datos como un par `<nombre, valor>`. Los nombres de campo que se utilizan para correlacionar datos de mensajes JMS con el MQeMsgObject se definen en com.ibm.mqe.MQe y com.ibm.mqe.jms.MQeJMSMsgFieldNames:

MQeJMS field names

```
MQe.MQe_JMS_VERSION
MQeJMSMsgFieldNames.MQe_JMS_CLASS
```

Nombres de campos de mensajes de JMS

```
MQeJMSMsgFieldNames.MQe_JMS_HEADER
MQeJMSMsgFieldNames.MQe_JMS_PROPERTIES
MQeJMSMsgFieldNames.MQe_JMS_PS_PROPERTIES
MQeJMSMsgFieldNames.MQe_JMSX_PROPERTIES
MQeJMSMsgFieldNames.MQe_JMS_BODY
```

Nombres de campos de cabeceras de JMS

```
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
```

Información de JMS para MQe

Se añaden directamente al `MQeMsgObject` dos pares de <nombre, valor> que contienen la información necesaria para que MQe vuelva a crear el mensaje de JMS:

`MQe.MQe_JMS_VERSION`
Contiene un tipo de datos *short* que describe el número de la versión de la implementación de JMS en MQe que se utiliza para almacenar el mensaje. El número de la versión actual es 1. La presencia o ausencia de un campo denominado `MQe.MQe_JMS_VERSION` se utiliza para determinar si un `MQeMsgObject` contiene un mensaje de JMS para MQe.

MQeJMSMsgFieldNames.MQe_JMS_CLASS
Contiene un tipo de datos de serie de caracteres (*String*) que describe el tipo de cuerpo de mensaje JMS almacenado en el `MQeMsgObject`. Las series de caracteres se definen en la tabla siguiente:

Tabla 5. Series de caracteres de MQeJMSMsgFieldNames.MQe_JMS_CLASS

Tipo de mensaje JMS	MQe.MQe_JMS_CLASS
Mensaje de bytes	jms_bytes
Mensaje de correlación	jms_map
Mensaje nulo	jms_null
Mensaje de objeto	jms_object
Mensaje de corriente de datos	jms_stream
Mensaje de texto	jms_text

Archivos de cabecera de JMS

Los campos de cabecera JMS se almacenan en un `MQeMsgObject` utilizando las siguientes normas:

1. Si un campo de cabecera JMS es idéntico a un campo MQeMsgObject definido, el valor de la cabecera se correlaciona directamente con el campo adecuado en el MQeMsgObject.
2. Si un campo de cabecera de JMS no se correlaciona directamente con un campo definido, pero se puede representar mediante campos existentes definidos por MQe, el valor de la cabecera se convierte según sea conveniente y, a continuación, se establece en el MQeMsgObject.
3. Si por entonces MQe no ha definido un campo equivalente, el campo de cabecera se almacenará en un objeto MQeFields, que posteriormente se incluye en el MQeMsgObject. Así se garantiza que el campo de cabecera JMS en cuestión se pueda restaurar cuando se vuelva a crear el mensaje JMS.

Los campos de cabecera que se correlacionan directamente con los campos MQeMsgObject son:

Tabla 6. Campos de cabecera que se correlacionan directamente con los campos MQeMsgObject

Campo de cabecera JMS	Campo definido MQeMsgObject
JMSTimestamp	MQe.Msg_Time
JMSCorrelationID	MQe.Msg_CorrelID
JMSExpiration	MQe.Msg_ExpireTime
JMSPriority	MQe.Msg_Priority

Los campos de cabecera JMS JMSReplyTo y JMSMessageID se convierten antes de almacenarlos en campos MQeMsgObject.

JMSReplyTo se divide entre MQe.Msg_ReplyToQMgr y MQe.Msg_ReplyToQ, mientras que JMSMessageID es la serie de caracteres String "ID:" seguida por un código hash de 24 bytes generado a partir de una combinación de MQe.Msg_OriginQMgr y MQe.Msg_Time.

Los tres campos de cabecera JMS restantes, JMSDeliveryMode, JMSRedelivered y JMSType, no tienen equivalentes en MQe. Estos campos se almacenan en un objeto MQeFields de la siguiente manera:

- Como un campo de enteros denominado MQe.MQe_JMS_DELIVERYMODE
- Como un campo booleano denominado MQe.MQe_JMS_REDELIVERED
- Como un campo de serie de caracteres denominado MQe.MQe_JMS_JMSTYPE

A continuación, este objeto MQeFields se almacena en el MQeMsgObject como MQe.MQe_JMS_HEADER. Por último, se vuelve a crear JMSDestination cuando se recibe el mensaje y, por lo tanto, no es necesario almacenarlo en el MQeMsgObject.

Propiedades de JMS

Cuando se almacenan campos de propiedades JMS en un MQeMsgObject, el formato <nombre, valor> que utilizan las propiedades JMS se corresponde con el formato de datos de un objeto MQeFields:

Tabla 7. Campos de propiedades JMS y el objeto MQeFields

Tipo de propiedad	Objeto MQeFields correspondiente
Específica de la aplicación	MQe.MQe_JMS_PROPERTIES
Estándar (nombre_JMSX)	MQe.MQe_JMSX_PROPERTIES
Específico del proveedor (nombre_proveedor_JMS)	MQe.MQe_JMS_PS_PROPERTIES

Se utilizan tres objetos MQeFields, correspondientes a los tres tipos de propiedades JMS, específica de la aplicación, estándar y específica del proveedor, para almacenar los pares <nombre, valor> almacenados como propiedades de mensajes JMS.

A continuación, estos tres objetos MQeFields se incluyen en el MQeMsgObject con los siguientes nombres:

- MQe.MQe_JMS_PROPERTIES, específico de la aplicación
- MQe_MQe_JMSX_PROPERTIES, propiedades estándar
- MQe.MQe_JMS_PS_PROPERTIES, específico del proveedor

Tenga en cuenta que actualmente MQe no establece ninguna propiedad específica del proveedor. Sin embargo, este campo se utiliza para activar MQe de modo que maneje mensajes JMS de otros proveedores, como MQ.

El siguiente fragmento de código crea un mensaje de texto JMS de MQe al añadir los campos necesarios a un 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);
```

Ahora, utilice JMS para recibir el mensaje e imprimirlo:

```
// 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());
```

Esto ofrece:

```
   CAMPOS DE CABECERA
  ----------------------------
    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

   CAMPOS DE PROPIEDADES (sólo lectura)
  ----------------------------
    JMSXRcvTimestamp : 1032876532537
    anInt : 12345

   CUERPO DEL MENSAJE (sólo lectura)
-----------------------------------------------------------------
  A test message to MQe JMS
```

Tenga en cuenta que JMS establece algunos de los campos de mensajes JMS internamente, por ejemplo, `JMSMessageID` y `JMSXRcvTimestamp`.

Cuerpo de los mensajes de JMS:

Independientemente del tipo de mensaje de JMS, MQe almacena internamente el cuerpo del mensaje de JMS como una matriz de bytes. Para los tipos de mensajes que actualmente reciben soporte, esta matriz de bytes se crea como se indica a continuación:

Tabla 8. Cuerpo de los mensajes de JMS

Tipo de mensaje JMS	Conversión
Mensaje de bytes	ByteArrayOutputStream.toByteArray();
Mensaje de objeto	<serialized object>.toByteArray();
Mensaje de texto	String.getBytes("UTF-8");

Cuando el cuerpo del mensaje JMS se almacena en un `MQeMsgObject`, esta matriz de *bytes* se añade directamente al `MQeMsgObject` con el nombre `MQe.MQe_JMS_BODY`.

Clases de JMS de MQe

Las clases de MQe para Java Message Service constan de un número de clases e interfaces Java que se basan en el paquete de interfaces y clases javax.jms de Sun. Estas clases se encuentran en el paquete com.ibm.mqe.jms. Se proporcionan las siguientes clases:

Tabla 9. Clases de JMS de MQe

Clase	Implementa
MQeBytesMessage	BytesMessage

Tabla 9. Clases de JMS de MQe (continuación)

Clase	Implementa
MQeConnection	Connection
MQeConnectionFactory	ConnectionFactory
MQeConnectionMetaData	ConnectionMetaData
MQeDestination	Destination
MQeJMSEnumeration	Java.util.Enumeration de QueueBrowser
MQeJMSJNDIQueue	Queue
MQeJMSQueue	Queue
MQeMessage	Mensaje
MQeMessageConsumer	MessageConsumer
MQeMessageProducer	MessageProducer
MQeObjectMessage	ObjectMessage
MQeQueueBrowser	QueueBrowser
MQeQueueConnection	QueueConnection
MQeJNDIQueueConnectionFactory	QueueConnectionFactory
MQeQueueConnectionFactory	QueueConnectionFactory
MQeQueueReceiver	QueueReceiver
MQeQueueSender	QueueSender
MQeQueueSession	QueueSession
MQeSession	Session
MQeTextMessage	TextMessage

Tenga en cuenta que las aplicaciones implementan el MessageListener y el ExceptionListener.

Errores y manejo de errores

Visión general de los errores y manejo de los errores en Java y C

En este capítulo se describe lo que ocurre si se produce un error en las bases de código en Java y en C .

Manejo de errores en Java

Los errores en la base de código en Java se manejan mediante excepciones. En la consulta de programación en Java de MQe se documentan todos los códigos de excepción que el código en Java de MQe puede devolver en las clases siguientes:

- com.ibm.mqe.MQeExceptionCodes
- com.ibm.mqe.mqbridge.MQeBridge.ExceptionCodes

Manejo de errores en C

La base de código en C indica los errores mediante códigos de *retorno* y de *razón*. El código C no dispone de ninguna excepción que maneje el mecanismo como en el caso de C++. MQe no utiliza las funciones de manejo de errores del sistema operativo. Un `MQeExceptBlock` maneja errores y devuelve valores de las funciones. Una aplicación puede, si lo desea, instalar todos los manejadores de excepciones del sistema operativo que necesite.

La naturaleza específica de una condición de error se devuelve utilizando dos valores, MQERETURN y MQEREASON. MQERETURN determina el área general en el que se ha producido una anomalía en la aplicación y hace una distinción entre avisos y errores. Los avisos se pueden ignorar, pero los errores no se deben ignorar. Con errores, la aplicación tiene que resolver el problema para continuar de forma segura.

MQERETURN y MQEREASON se devuelven en MQeExceptBlock. El valor MQERETURN es también el valor de retorno de la función.

Estructura de los códigos

El archivo de cabecera MQe_nativeReturnCodes.h lista todos los códigos de retorno y de razón. Están divididos en áreas de función y, a continuación, por error o aviso. Por ejemplo, MQERETURN_QUEUE_MANAGER_ERROR y MQERETURN_QUEUE_MANAGER_WARNING. Los avisos indican que una situación se puede ignorar.

Bloque de excepciones

La estructura de MQeExceptBlock se utiliza para devolver al usuario el código de retorno y de razón, generado por una llamada de función. Si una llamada de función no devuelve MQERETURN_OK, utilice la macro ERC para obtener el código de razón.

MQe se entrega con dos macros:

EC Esta macro indica el código de retorno en la estructura del bloque de excepciones.

ERC Esta macro indica el código de razón en la estructura del bloque de excepciones.

El convenio en MQe es que un puntero que señala a un bloque de excepciones se pase en primer lugar a una función nueva. Un puntero que señala al manejador de objeto se pasa en segundo lugar, seguido de cualquier parámetro adicional. Las siguientes llamadas, el manejador de objeto es el primer parámetro que se pasa y el puntero que señala al bloque de excepción es el segundo, seguido de cualquier parámetro adicional.

La estructura del bloque de excepciones, tal como se muestra en el siguiente ejemplo, es MQeExceptBlock_st.

```
struct MQeExceptBlock_st
  {
      MQERETURN ec;
                  /* return code*/
      MQEREASON erc;
                  /* reason code*/
      MQEVOID*      reserved;
                  /* reserved for internal use only*/
  }
```

Se recomienda asignar el bloque de excepciones en la pila, en lugar de hacerlo en el almacenamiento dinámico. Esto simplifica las posibles asignaciones de memoria, aunque no exista ninguna restricción en la asignación de espacio en el almacenamiento dinámico. En el siguiente código se muestra cómo hacerlo:

```
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 {
}
```

Todas las llamadas a API necesitan tomar bloques de excepciones. La base de código de enlaces C permite que el valor NULL se pase a una llamada a API. Sin embargo, esta función se ha dejado de utilizar en la base de código en C y, por lo tanto, no se recomienda.

Se debe utilizar un bloque de excepción distinto para cada hebra en la aplicación.

Nota: Si un error no se corrige, las siguientes llamadas API pueden transferir el sistema a un estado impredecible.

Macros útiles

Un número de macros ayuda a acceder al bloque de excepciones:

SET_EXCEPT_BLOCK
Establece los códigos de retorno y de razón para especificar valores, por ejemplo:

```
 MQeExceptBlock exceptBlock;
SET_EXCEPT_BLOCK(&exceptBlock,
        MQERETURN_OK,
        MQEREASON_NA);
```

SET_EXCEPT_BLOCK_TO_DEFAULT
Establece códigos de retorno y de razón en valores de no error, por ejemplo:

```
 MQeExceptBlock exceptBlock;
SET_EXCEPT_BLOCK_TO_DEFAULT(&exceptBlock);
```

EC Accede al código de retorno, por ejemplo:

```
 MQeExceptBlock exceptBlk;
/*MQe API call */
 MQERETURN returncode;
returnCode = EC(&exceptBlock);
```

ERC Accede al código de razón, por ejemplo:

```
 MQeExceptBlock exceptBlk;
/*MQe API call*/
 MQEREASON reasoncode;
 MQEREASON reasonCode = ERC(&exceptBlock);
```

NEW_EXCEPT_BLOCK
Puede crear un bloque de excepciones temporal. Esto es útil para operaciones de borrado temporal.

Índice

P

R

S