50

MQ update

August 2003

In this issue

MQ Update

Published by
Xephon
27-35 London Road
Newbury
Berkshire RG14 1JL
England
Telephone: 01635 38126
From USA: 01144 1635 38126
Fax: 01635 38345
E-mail: info@xephon.com

North American office
Xephon/QNA
Post Office Box 350100
Westminster CO 80035-0100, USA
Telephone: (303) 410 9344
Fax: (303) 438 0290

Editor
Madeleine Hudson
E-mail: MadeleineH@xephon.com

Subscriptions and back-issues
A year's subscription to *MQ Update*, comprising twelve monthly issues, costs £255.00 in the UK; $380.00 in the USA and Canada; £261.00 in Europe; £267.00 in Australasia and Japan; and £265.50 elsewhere. In all cases the price includes postage. Individual issues, starting with the July 1999 issue, are available separately to subscribers for £22.50 ($33.75) each including postage.

Contributions
When Xephon is given copyright, articles published in *MQ Update* are paid for at the rate of £170 ($260) per 1000 words and £100 ($160) per 100 lines of code for the first 200 lines of original material. The remaining code is paid for at the rate of £50 ($80) per 100 lines. In addition, there is a flat fee of £30 ($50) per article. For more information about contributing an article you can download a copy of our *Notes for Contributors* from www.xephon.com/nfc.

MQ Update on-line
Code from *MQ Update*, and complete issues in Acrobat PDF format, can be downloaded from our Web site at www.xephon.com/mq; you will need to supply a word from the printed issue.

Real time JMS server implementation on WMQ

INTRODUCTION

This article illustrates the use of JMS (Java Messaging System), in a WMQ environment, for large-scale file replication in a cross-platform, cross-vendor environment. I will discuss the advantages and potential pitfalls and provide practical instructions for successfully implementing WMQ as a JMS server.

BACKGROUND

When we think of messaging solutions we probably envision a system that integrates different applications within the enterprise or even outside the enterprise (partner/customer applications) by a remote message invocation mechanism. Classic examples are homogeneous interfaces to heterogeneous backends and portals that delegate back-end processing of user requests and then reformat them for end-user presentation, or provide back-office details for a customer on front-office applications to provide a 360 degree view during customer interactions.

The common thread in messaging methods is the assumption that the messaging solution, while providing robust, highly available communication between systems, is fundamentally inefficient and should be used only as a last resort when communication to an outside system cannot be avoided. This view of messaging has prevailed from the onset of remote method calls (RMC) to more modern messaging solutions such as CORBA and DCOM, and it has limited the type of problem that messaging solutions are usually applied to.

Over the last decade there has been an increased understanding of the requirements of distributed systems. Emerging technologies, such as Java and .NET, have included code distribution as part of their fundamental programming model. In doing so, these technologies have incorporated high availability and fault tolerance into messaging, encouraging solution providers to deliver systems with performance characteristics applicable to a wider variety of problems.

On the Navy Marketing and Advertisement System that our company is building for Commander, Navy Recruiting Command, US Navy, I was asked to implement a file distribution and replication solution that previously would have required a custom system, integrating secure FTP, database replication, and other one-off solutions. Rather than choosing the custom development path I explored the possibility of applying state-of-the art messaging solutions to the problem. We found that not only did JMS provide the necessary infrastructure for transferring information but it also handled all of the infrastructure issues relating to quality of service, security, reliability, and performance that the system required. This article describes the challenges our team faced and how JMS, in the form of MQSeries, let us meet and exceed our customer requirements and expectations.

THE PROBLEM

Navy Recruiting Command had hired an outside firm to handle telephone calls. This firm had a number of call centres around the country where operators recorded interactions with leads. The recordings had to be quickly and reliably indexed and archived in remote data centres. The storage procedure was not allowed to affect the ability of the operator's system to record and store ongoing interactions.

The customer services firm had an existing system that included a combination of custom code, VPN, and other technologies. But the existing solution was falling far short of the performance and reliability goals and was a combination of technology that was hard to understand and expensive to maintain.

While developing a replacement system we had to base the messaging solution on WMQ as a part of the client's standards. Considering the security flaws with non-JMS solutions, particularly those based on FTP and secure copy (SCP), our team concluded that the development effort to base our solution on these non-JMS methods would be prohibitive and, therefore, settled on the JMS solution, which provided them 'out of the box'.

THE SOLUTION

We developed a WMQ-based JMS system that:

- Provided reliable archiving for recorded multi-media files.
- Would allow extendibility to let multiple data centres receive the files.
- Would allow additional data types to be archived.

The files that were in doubt were the larger ones (50K - 500K) that we had transferred in previous projects and which involved messaging solutions. Our first task was to ensure that the data sizes would not preclude a JMS solution. We tested the performance of the WMQ system's delivery with message payloads of various sizes. The results showed that, with appropriate configuration, messages of up to 1MB did not have a noticeable effect on overall system performance.

We laid out the system architecture; the existing infrastructure had a system on each client that created multi-media files in response to interactions between operators and users. These files needed to be archived. Our system starts a process running on each machine and looks for these files in known directories. When new files are detected they are packaged into a JMS payload and sent to a JMS server in one of the data centres for delivery. Once the JMS server acknowledges receipt the files are removed from the sender. The JMS server transfers the data to one of the available handlers in the data centres for archiving.

The system we built relies on point-to-point destinations called queues in JMS. The messages are actually delivered from the JMS broker to the queue and the MQ receiver client retrieves them from the queue.

It was important to our client to limit vendor lock-in, meaning we needed to design our code to minimize the impact of changing JMS vendors. Key advantages of JMS were the open standard it was based on and the broad industry support it receives, so that with properly designed code we could make the system work with any JMS system. The platform independence was easily accomplished by encapsulating all vendor-specific calls inside classes we called JMSProviders. These providers handled such vendor-specific issues as factory lookup, error handling, connection creation, and message property setting (see below for an example).

EXAMPLE ONE

```
Public QueueConnection createConnection() throws JMSException()
{
Return
  GetConnectionFactory().createQueueConnection(getUserName(),
  getPassword();
}
```

By leveraging the Java Naming and Directory Interface (JNDI) we stored the vendor-specific references. There is a small amount of vendor-specific code needed to handle some idiosyncrasies but it can be limited to some 'adaptor' classes, keeping it out of the application code. (Below is an example.) Because JMS is designed to work easily with JNDI it was another immediate advantage over other solutions – a centralized location for configuration information that not only could hold text-based information, but also could store configured objects.

EXAMPLE TWO

```
public final static string
CONNECTION_FACTORY_LOOKUP_NAME_KEY =
  "CONNECTION_FACTORY_LOOKUP_NAME";
public final static
 String FILE_TRANSFER_QUEUE_LOOKUP_NAME_KEY =
  "FINAL_TRANSFER_QUEUE_LOOKUP_NAME";
public final static String
  JMS_PROVIDER_CLASS_KEY = "JMS_PROVIDER_CLASS";
public void init() throws NamingException   {
   InitialContext jndi = createInitialContext();
   InitConnectionFactory(jndi);
```

```
    InitFileTransferQueue(jndi);
                                           }
public QueueConnection createConnection() throws JMSException {
    return
   GetConnectionFactory().createQueueConnection(getUserName(),getPassword());
                                                            }
public void initConnectionFactory(InitialContextjndi) throws
    NamingException   {
    SetConnectionFactory((QueueConnectionFactory)jndi.lookup
   (getProperties().getProperty(CONNECTION_FACTORY_LOOKUP_NAME_KEY)));
                      }
public void initFileTransferQueue(InitialContext jndi) throws
    NamingException   {
       SetFileTransferQueue((Queue) jndi.lookup
  (getProperties().getProperty(FILE_TRANSFER_QUEUE_LOOKUP_NAME_KEY)));
                      }
```

Out of the box, JMS solutions allow messages to be sent with a guarantee that, once a message is acknowledged as delivered to the JMS server, it will be delivered to the destination (queue) to which it was addressed. MQSeries is no exception. Once the code to send the message to the server is executed the client can be assured that the destination will eventually receive the message even if the server in question has a failure during processing (if the destination is temporarily unavailable, or the JMS server dies, etc). (See example three below.) The Class in the code below is responsible for actually executing the sending of data once it has been determined that it's necessary to send the file.

By configuring the message as persistent we can guarantee that once the message is received by the destination (queue) it will remain there until it is retrieved from the queue – even across system failures. Hence, once the message is safely delivered to the local JMS server it can be deleted. The value of overcoming system failures cannot be overestimated; handling of periodic system outages and failures is one of the biggest problems with developing distributed archiving solutions. The customer's existing system had complicated and brittle code to handle failure scenarios and failures were costly in terms of processing and maintenance. JMS allowed us to solve all these problems by delegating them to a robust, battle-tested, commercial solution.

EXAMPLE THREE

```
public void sendMessage(byte[] payload, Boolean persistent) throws
   SendFailedException  {
   QueueSender sender = null;
   try {
     Message message = createMessage(payload);
     Sender = createSender
     (persistent ? DeliveryMode.PERSISTENT:DeliverMode.NON_PERSISTENT);
     sender.send(message);
        getClient().getLogService().logInfo(getName() +
        "  sent message "  + message.getJMSMessageID()  +  ".");
       }  catch  (JMSException  exception  {
         getClient().getLogService().logError
         ("JMS exception processing  "  +  getName(), exception);
         stop();
         throw new SendFailedException ("JMS Message Send Failed");
       }
   try {
           sender.close();
       }   catch (JMSException ignore)   {
            getClient().getLogService().logInfo(getName())  + " failed
                to close sender.
             Processing will continue.");
                                          }
```

The key to this solution is configuring the JMS messages and server to provide both adequate performance and quality of service. The configuration options are defined by the JMS specification and are implemented by all commercial solutions; however, the exact method of configuration varies from vendor to vendor.

The architecture and system we created is general and powerful. However, there are a number of moving parts that must be configured and hooked up in just the right way. Let's now look at some potential pitfalls and ways to circumvent them to ensure a successful set-up of WMQ as a JMS server.

OVERVIEW

With MQSeries, first set up a JNDI server to retrieve the implementation-specific settings, which in this case comprise the JMS Connection Factory. There are many different ways to do this but a good all-purpose choice is a Lightweight Directory Access Protocol (LDAP) server. We chose to use Qualcomm

SLAPD. Once the server is installed and running, the WMQ admin tools can be set up to use it as the repository for WMQ object information. Also, during set-up it's important to pay close attention to the IBM documentation for setting up JMS on top of WMQ. The process involves creating queues and other objects that are specific for JMS usage and not part of the standard WMQ installation.

After setting up the JNDI/LDAP and JMS servers you are ready to configure your clients. The first step is to understand how JMS interacts with IBM's MQSeries implementation. Java clients of MQSeries can interact in one of two modes: client or bind mode. Client can be used only by Java applets, and bind mode relies on DLLs or object libraries on the client. Because of the peculiarities of the implementation you can only use bind mode when using an LDAP server for JMS connection information. Therefore, the user login and password are stored in a global location (*com.ibm.mq.MQEnvironment.class*) rather than passed in at connection time. To accommodate these vendor issues we created a subclass of our standard JmsProvider class called MQSeriesProvider. The only thing this class will do is override the way in which a connection is created. Instead of calling example one:

```
public QueueConnection createConnection() throws JMSException()
{
 Return
 GetConnectionFactory().createQueueConnection(getUserName(),getPassword();
}
```

we must call:

```
newQueueConnection = getConnectionFactory().createQueueConnection();
```

Finally, you need to supply the JMS-specified elements to the clients, such as the queues, queue managers, queue factories, and so on. Now the reason for using LDAP and JNDI becomes apparent: we use the LDAP server to store these elements and use external files to hold the keys to those LDAP objects. The LDAP server can act like a JNDI server and respond to name lookups by returning the objects we stored. This is what allows the code in example two to work. The name for a JMS element is obtained from a class static variable (for the default name) or an external file (to use something other than the default). In short, the LDAP server is asked for the object that is stored at the key in question and an object, in this case the JMS object that we are interested in, is returned.

Our JMS-based solution facilitated a uniform, cross-platform, and cross-vendor configuration environment, using off-the-shelf components. Now our code is as isolated as possible from platform-specific and vendor-specific settings.

HOW THE APPLICATION WORKS

There are two key components to the application: a sender and a receiver. The sender launches a background that polls a directory for files that need to be archived, while the receiver simply waits for JMS messages to be delivered, then archives the files contained in the messages. The JMS API lets us define these components with almost no regard for the specific JMS implementation that we are using. The sender consists of three main parts:

- A JMSProvider for creating connections.
- A ConnectionPool for obtaining existing, idle connections (which we will call JMSConnections).
- A poller to watch for files that need to be transferred.

At startup, the JMSProvider is used to create some ready connections to the JMS server. The connections are placed in the pool and then the poller is started. When the poller detects a file that needs to be transferred it creates a separate thread to process the file. In the separate thread the poller then obtains a JMSConnection from the connection pool, uses it to create a BytesMessage, and places the binary contents of the file into that message. Finally, the message is addressed to the receiver, sent to the JMS server, and then the JMSConnection is returned to the ConnectionPool.

The receiver is a simpler component; it starts a number of

FileListeners that wait for messages to be placed in the receiver queue. Example four below shows the code for setting up the processing by the FileListeners. JMS guarantees that a queue will deliver each message no more than once, so we can safely start many different FileListener threads and know that each message (and therefore each file) will be processed only once. This guarantee is another important advantage of using a JMS-based solution.

In our scenario the system screened the leads once they responded to a Navy Job offer, and qualified lead information was sent to the field recruiters using this mechanism, along with a similar package that was sent to external fulfillment centres, to provide appropriate material to the candidates.

EXAMPLE FOUR

```
Public void startOn(Queue,queue) {
   SetQueue(queue);
   CreateConnection();
   Try {
      createSession();
      createReceiver();
      getConnection().start() ;  // this starts the queue listener
       }  catch (JMSException exception)  {
            // handle the exception
                                      }
                        }
public void createReceiver() throws javax.jms.JMSException   {
   try {
      QueueReceiver  receiver  = getSession().
CreateReceiver(getQueue());
      Receiver.setMessageListener(this);
        }     catch (JMSException exception) {
           // handle the exception
                                        }
                                                  }
public void createSession() throws JMSException {
   setSession(getConnection().
CreateQueueSession(false, Session.AUTO_ACKNOWLEDGE));
                                                  }
public void createConnection()  {
   while(!hasconnection())   {
      try {
         setConnection(getClient().createConnection());
         } catch (JMSException exception) {
```

```
 // Connections drop periodically on the internet, log and try again.
         try {
               Thread.sleep(2ØØØ);
             }  catch
                (java.lang.InterruptedException ignored) {
                                 }
                        }
            }
  }
```

The message handling code is written in a callback, a method that JMS automatically invokes when a message is delivered to the FileListener. The code for this message is shown in Example five below.

EXAMPLE FIVE

```
   Public void on Message (Message message)   {
      BytesMessage byteMessage  =  ((BytesMessage) message);
      OutputStream stream =  new BufferedOutputStream(
         new FileOutputStream(getFilenameFor(message)));
         byte[]   buffer = new byte[getFileBufferSize()];
         Int length = Ø;
      try  {
         While((length = byteMessage.readBytes(buffer)) != -1) {
                Stream.write(buffer,Ø,length);
           }
                stream.close();
   } catch (JMSException exception)  {
         // handle the JMSException
   } catch (IOException exception)   {
         //handle the IOException
   }
                                    }
```

A trick to remember when setting up the receiver is to ensure that the initial thread that launches all the FileListeners continues to run after all of the FileListeners have started. This is necessary because some JMS implementations start QueueListeners in daemon threads. So the Java Virtual Machine might exit unexpectedly early if the only threads that are running are daemon threads. Example six below shows some simple code to prevent this from occurring.

EXAMPLE SIX

```
   public static void main(String[]) args) {
      ReceiverClient newReceiverClient = new ReceiverClient();
      NewReceiverClient.init();
      SetSoleInstance(newReceiverClient);
         While(!finished)  { // this prevents the VM from exiting early
   try {
           Thread.sleep(1ØØØ);
       }  catch (InterruptedException ex)  {
      }
     }
   }
```

CONCLUSION

After our initial implementation of the project we added features such as message compression, auto-recovery when sites become unreachable, federated message brokers, security, robust log-in, administration, and more. The elements were easy to add because of the open JMS model as well as the robustness of the architecture and features of WMQ. In addition to exceeding our customer expectations this messaging system proved that JMS is a viable solution not only for small, message-oriented applications but also for large-scale, mission-critical data transfer operations. We are moving ahead with using JMS with WebSphere in the EAI implementation in the Navy Marketing and Advertisement System.

Vikas Baruah (vikas.baruah@ams.com)
American Management Systems (USA)

WMQ groups and segments

Many features have been added to MQSeries since its release nearly a decade ago. Even for veteran MQSeries developers, learning the details of how to use a particular feature can be daunting. In my capacity as a senior middleware architect I am responsible for directing developers on the proper way of using the features of MQSeries. Features I often get asked about are message groups and segmentation.

In this article my aim is to help developers quickly start using a feature without getting bogged down in the *Programmer Reference and User Manual*. This text is not meant to be a replacement for reading the manuals. It is simply a quick tutorial on message groups and segmentation. I'll make some recommendations on how to use them and point out some of the things you should look out for. One general note is that, in all cases, these features require the version field of the MQMD to be set to MQMD_VERSION_2. I mention this because my examples are written in Java, which defaults to this version, so you won't see it set in the code.

WHAT ARE MESSAGE GROUPS AND SEGMENTS?

A message group allows an application to group related messages and ensure the retrieval order of messages. It is made up of one or more logical messages.

A logical message is a single unit of application information made up of one or more physical messages. When there are multiple physical messages in a logical message they are referred to as segments. A logical message does not need to be part of a message group.

A segment is used to split large messages that may not be able to be processed because of the configuration of a particular queue or queue manager maximum message size or because of application buffer limits.

A physical message is the smallest unit that you can PUT to or GET from a queue.

Figure 1 depicts the relationship among groups, segments, and physical messages. It shows a message group that contains two logical messages, where the first logical message is made up of a single physical message and the second logical message is segmented into two physical messages. The figure also shows a segmented logical message that is not a member of a group and a single physical message, also not a member of a group.

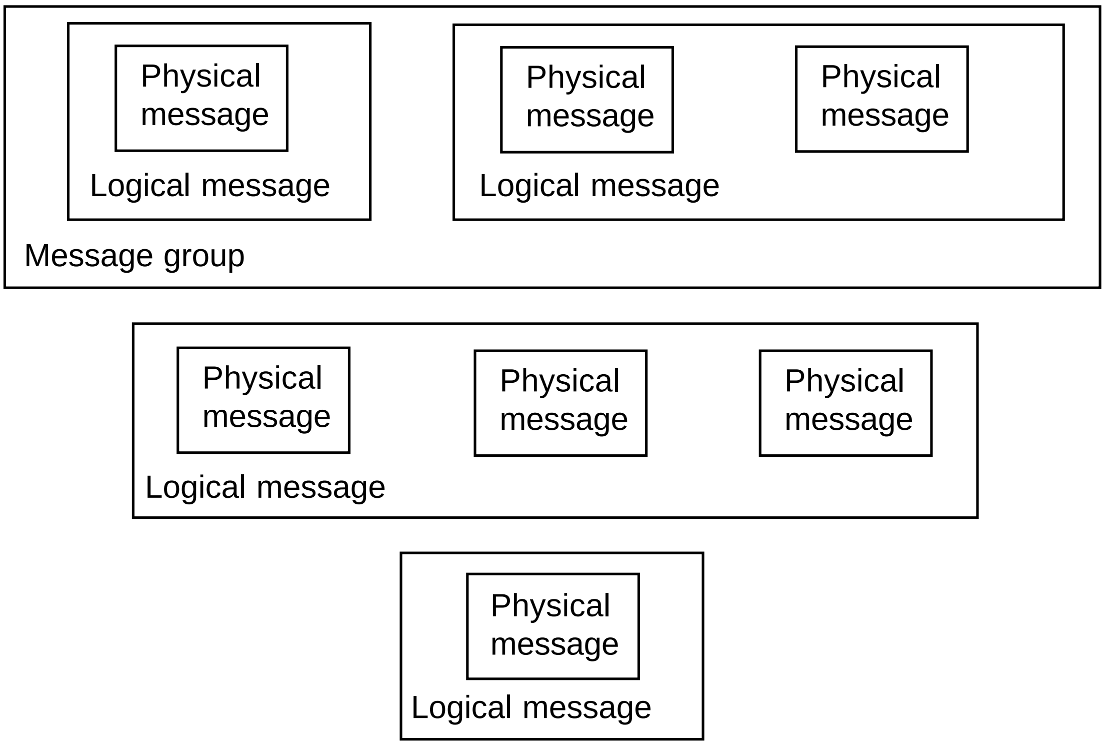

Figure 1: Message groups and segments

WHY USE MESSAGE GROUPS?

Message groups allow the application to ensure that related messages are processed together and in a particular order. While you can implement this by using the Msg-ID and Correl-ID fields in the message descriptor, message groups provide a semi-automated and consistent mechanism for implementing these capabilities. As we'll see, using message groups can help simplify design and reduce complexity.

HOW IS A MESSAGE GROUP SPECIFIED?

There are a number of fields in the MQMD that are used to manage message groups:

- Group-ID.
- MsgSeqNumber.
- MsgFlags.

There are also values in the PMO and GMO options field that are required for message groups.

To specify a message group the MsgFlags field in the MD structure is set to MQMF_MSG_IN_GROUP or MQMF_LAST_MSG_IN_GROUP. MQMF_MSG_IN_GROUP is set for all logical messages in the group until the last logical message, which is set to MQMF_LAST_MSG_IN_GROUP. It is always up to the application to manage these flags when putting messages to a queue. Once a group is started it must be completed before starting a new group. If the application tries to put a message to the queue that is not in the current group then a MQRC_INCOMPLETE_GROUP error is returned. When getting messages MsgFlags is an output field used to determine when you have completely retrieved the messages in the group.

The Group-ID field in MQMD is used to identify a group. This field can be manually set by the application or managed by the queue manager.

The MsgSeqNum field plus the Group-ID field are used to identify a logical message. The MsgSeqNum starts at 1 and increments by 1 for each logical message in the group. It can be manually set by the application or managed by the queue manager.

When putting message groups the recommended approach is to have the queue manager take responsibility for managing the Group-ID and MsgSeqNumber in the MD. This is accomplished by setting the MQPMO_LOGICAL_ORDER flag in the PMO's options field. Example one is a simple Java code fragment showing this method. In this example, the first PUT will generate a unique Group-ID and set the MsgSeqNumber number to 1. Subsequent calls will increment the value in MsgSeqNumber by 1.

Example one: PUT a message group to a queue

```
MQMessage putMsg = new MQMessage();
MQPutMessageOptions pmo = new MQPutMessageOptions();
pmo.options = MQC.MQPMO_NEW_MSG_ID |
              MQC.MQPMO_SYNCPOINT  |
              MQC.MQPMO_LOGICAL_ORDER;
```

```
putMsg.messageFlags = MQC.MQMF_MSG_IN_GROUP;
putMsg.clearMessage();
putMsg.writeString(msgDataA);
q1.put(putMsg,pmo);
putMsg.messageFlags = MQC.MQMF_MSG_IN_GROUP;
putMsg.clearMessage();
putMsg.writeString(msgDataB);
q1.put(putMsg,pmo);
putMsg.messageFlags = MQC.MQMF_LAST_MSG_IN_GROUP;
putMsg.clearMessage();
putMsg.writeString(msgDataC);
q1.put(putMsg,pmo);
        qMgr.commit();
```

It is also recommended that the application manage the syncpoint. This way there won't be a partial group on the queue if an error occurs during processing. Now, there are ways to put a message group onto a queue across multiple units of work, but the added complexity of the recovery logic in the code just doesn't warrant the effort, so try to avoid it. If you can't, then be aware that, when using Java, the queue manager will throw an MQRC_INCOMPLETE_GROUP exception on the close.

If the logical order flag is not specified it's up to you to set and manage Group-ID and MsgSeqNumber. If you must manage the fields yourself, at the very least you should have the queue manager generate a unique Group-ID for you. The potential for having logically different groups on the queue with the same group identifier can result in obvious problems when you attempt to get the messages.

To generate an identifier set the Group-ID field to MQGI_NONE for the first message put to the queue. If the PUT is successful the queue manager will return a unique value in Group-ID. Use this value on subsequent PUTs for the remaining messages in the group.

The real opportunity to simplify design and reduce code by using message groups comes when getting messages. Let's say that the requirement is to have a server application retrieve a number of related messages from an input queue as one message and that processing can only start when all the pieces of the transaction have arrived. This is a fairly common requirement. Before message groups you would have had to use Msg-ID and/or Correl-ID to indicate the relationship among messages. You would have also needed to identify the end of a group in some way. This would have to have been designed into the message in some way. Waiting for all the pieces to arrive before processing would require some fairly sophisticated processing, eg excessive browsing and moving data to temporary queues, etc.

What happens if the messages arrive out of order? This is not impossible in a complicated MQ network and in a large organization you can be sure that there would be a number of totally different implementations solving the same problem. By using message groups you can implement these requirements with some fairly simple code. The Java fragment in example two shows how this requirement could be implemented.

In example two the GMO options flag MQGMO_ALL_MSGS_AVAILABLE is used to wait for all the messages in the group to arrive before returning data from the get. The GMO options flag MQGMO_LOGICAL_ORDER forces the queue manager to retrieve the messages in MsgSeqNumber order, not by their position on the queue. The reason for the other options should be self-evident. Examine the ‘if-else’ after the get. Here, we are trying to determine the group disposition of the message. When it is the last message we commit the GETs. When in a group we set the GMO match options to MQMO_GROUP_ID to select the next message in the group. If not in a group, then process a single message.

It is important to note that you should check for the MQMF_LAST_MSG_IN_GROUP flag before checking the MQMF_MSG_IN_GROUP flag. This is because on the last message both flags are set. This is the case even if the message was put to the queue with only the last message flag set. The queue manager does this for some unknown reason, so be aware of it.

Example two: GET a message group

```
MQMessage getMsg = new MQMessage();
// Set the get message options to retrieve only complete messages
```

```
MQGetMessageOptions gmo = new MQGetMessageOptions();
gmo.options = MQC.MQGMO_ALL_MSGS_AVAILABLE |
              MQC.MQGMO_LOGICAL_ORDER |
              MQC.MQGMO_SYNCPOINT |
              MQC.MQGMO_WAIT;
// Get any message
gmo.matchOptions = MQC.MQMO_NONE;
boolean done = false;
do
{
   try
   {
      // Get the message
      q1.get(getMsg,gmo);
      if ((getMsg.messageFlags & MQC.MQMF_LAST_MSG_IN_GROUP) ==
                      MQC.MQMF_LAST_MSG_IN_GROUP)
      {
         // Reset the message flags to start getting
         // a new group or message.
         gmo.matchOptions = MQC.MQMO_NONE;
         qMgr.commit();
      }
      else if ((getMsg.messageFlags & MQC.MQMF_MSG_IN_GROUP) ==
                       MQC.MQMF_MSG_IN_GROUP)
      {
                           // Get the rest of the messages
         gmo.matchOptions = MQC.MQMO_MATCH_GROUP_ID;
      }
      else
      {
         // Message not in a group
         gmo.matchOptions = MQC.MQMO_NONE;
         qMgr.commit();
      }
   }
   catch (MQException ex)
   {
      if (ex.completionCode == MQException.MQCC_WARNING)
      {
         System.out.println("MQ warning: Completion code "+
                 ex.completionCode +
                 ", Reason code " + ex.reasonCode);
         done = true;
       }
       else
       {
         System.out.println("MQ error: Completion code "+
                  ex.completionCode +
                  ", Reason code " + ex.reasonCode);
          done = true;
```

```
            }
        }
    } while (!done);
```

WHY USE SEGMENTATION?

Segmentation allows large messages to be split up into smaller ones so that they can be processed. This is generally required when messages have to pass through an intermediate queue or queue manager before arriving at the target queue. Segmentation allows the applications on each end to send and receive messages that are larger than those that can be handled by the pass-thru nodes. Segmentation can also be used to provide target applications with the ability to process large messages when buffer space is limited. There are two types of segmentation available:

- System segmentation.
- Application segmentation.

SYSTEM SEGMENTATION

System segmentation is the simplest form of segmentation. It allows the application to put a large message to a queue and lets the queue manager decide where to split the message. The splitting of the message is transparent to the application. This is also referred to as ‘arbitrary segmentation’ because the queue manager splits the message on 16-byte boundaries without any consideration to the data layout of the message.

How are system segments specified?

Few changes are required to implement system segmentation. The code fragment in example three shows a simple PUT/GET sequence. The putting application needs simply to set the MsgFlags field in the MD to MQMF_SEGMENTATION_ALLOWED. This tells the queue manager to segment the message if it is too large for the queue or the queue manager to handle.

The get side of the example needs only to set the MQGMO_COMPLETE_MSG flag in the GMO. This option tells the queue manager to return only complete messages. The queue manager will reassemble the segments for you. Using this method assumes the application is capable of allocating a buffer large enough to hold the entire message. It is useful when the message has to pass through an intermediate that is not capable of handling large messages. You may also want to use this method to reduce the blocking effect on channel batching. Since a large message still counts as one message in the channel batch count but needs more time to transmit, a very large message sent unsegmented may affect the response time of other messages.

Example three: system segmentation

```
// Open the target queue
// NOTE: SMALL.MSG.QUEUE is defined with a
//       maximum message length of 16.
MQQueue q1 =
   qMgr.accessQueue("SMALL.MSG.QUEUE",
      openOptions);
System.out.println("Queue: " + q1.name +
                           " has max msg len of " +
                           q1.getMaximumMessageLength());
System.out.println();
// Define an MQ message and put message options
String msgData = new String("123456789Ø123456789Ø123456789");
MQMessage putMsg = new MQMessage();
MQPutMessageOptions pmo = new MQPutMessageOptions();
pmo.options = MQC.MQPMO_NEW_MSG_ID;
// Allow segmentation.
// NOTE: If this is not specified, then the
//       put will fail with CompCode = 2Ø3Ø,
//       MQRC_MSG_TOO_BIG_FOR_Q
putMsg.messageFlags  = MQC.MQMF_SEGMENTATION_ALLOWED;
putMsg.writeString(msgData);
q1.put(putMsg,pmo);
System.out.println("PUT message len: " + putMsg.getMessageLength());
System.out.println("PUT message data: " + msgData);
System.out.println();
// Now, lets see how many physical messages are on the queue
System.out.println("Current Queue Depth: " + q1.getCurrentDepth());
System.out.println();
// Get the logical message we just put to the queue
MQMessage getMsg = new MQMessage();
```

```
// Set the get message options to retrieve only complete messages
MQGetMessageOptions gmo = new MQGetMessageOptions();
gmo.options = MQC.MQGMO_COMPLETE_MSG;
// Get the message
q1.get(getMsg,gmo);
System.out.println("GET message len: " + getMsg.getMessageLength());
System.out.println("GET message data: " +
   getMsg.readString(getMsg.getMessageLength()));
System.out.println();
```

APPLICATION SEGMENTATION

This is a more complicated form of segmentation. It allows the application to control how the segments are put to and retrieved from a queue. There are various reasons why you would want to use this method. One case is when the sending or receiving application is not capable of allocating a buffer large enough to hold the entire message. Another case is when you need to control the boundaries of where the message data is to be split. Also, there is the case where you may want to begin sending segments before all the message data is available. These are the most likely reasons for using segmentation, though others may exist as well. You can also combine system segmentation with application segmentation. This way, one side can have the queue manager transparently perform the segmentation or reassembly, while the other side performs segmentation manually in the application.

How are application segments specified?

As with message groups, segments are specified and controlled by using fields in the MD and PMO/GMO structures. The two main fields used to define a segment in the MD are MsgFlags and Offset.

The offset field specifies the position of the data from the beginning of the logical message. It is manually or automatically set. Also, the options field in the PMO/GMO structures will be used again.

To specify a segment the MsgFlags field in the MD structure is set to MQMF_SEGMENT or MQMF_LAST_SEGMENT.

MQMF_SEGMENT is set for all segments in the message until the last segment, when you set MQMF_LAST_SEGMENT. As with message groups, once started, the segmented message should be completed before starting a new one.

When getting messages MsgFlags is an output field used to determine when you have completely retrieved the messages in the group. Once again it is recommended that the application manage the syncpoint. This way there won't be a partial message on the queue if an error occurs during processing.

It is recommended that you use automatic offset generation. To do this, set the PMO options flag MQPMO_LOGICAL_ORDER. This will instruct the queue manager to maintain the offset value for you. The code fragment in example four shows this in a simple sequence where the PUT side uses manual segmentation and the get side uses automatic reassembly.

Example four: application segmentation on the PUT

```
pmo.options = MQC.MQPMO_NEW_MSG_ID |
   MQC.MQPMO_SYNCPOINT |
   MQC.MQPMO_LOGICAL_ORDER;
// PUT 4 segments on to the queue
putMsg.messageFlags = MQC.MQMF_SEGMENT;
putMsg.writeString(msgDataA);
q1.put(putMsg,pmo);
System.out.println("PUT message len: " + putMsg.getMessageLength());
System.out.println("PUT message data: " + msgDataA);
putMsg.clearMessage();
putMsg.writeString(msgDataB);
q1.put(putMsg,pmo);
System.out.println("PUT message len: " + putMsg.getMessageLength());
System.out.println("PUT message data: " + msgDataB);
putMsg.clearMessage();
putMsg.writeString(msgDataC);
q1.put(putMsg,pmo);
System.out.println("PUT message len: " + putMsg.getMessageLength());
System.out.println("PUT message data: " + msgDataC);
putMsg.messageFlags = MQC.MQMF_LAST_SEGMENT;
putMsg.clearMessage();
putMsg.writeString(msgDataD);
q1.put(putMsg,pmo);
System.out.println("PUT message len: " + putMsg.getMessageLength());
System.out.println("PUT message data: " + msgDataD);
System.out.println();
```

```
qMgr.commit();
// Get the logical message we just put to the queue
MQMessage getMsg = new MQMessage();
// Set the get message options to retrieve only complete messages
MQGetMessageOptions gmo = new MQGetMessageOptions();
gmo.options = MQC.MQGMO_COMPLETE_MSG;
// Get the message
q1.get(getMsg,gmo);
System.out.println("GET message len: " + getMsg.getMessageLength());
System.out.println("GET message data: " +
   getMsg.readString(getMsg.getMessageLength()));
System.out.println();
```

If the logical order flag is not specified you must manually set the offset value on each PUT. But be warned; if your offset calculations are off, the queue manager will not indicate any problem when you PUT the segments of the logical message. However, you will not be able to retrieve the segments as a complete message by using a single GET that sets the MQGMO_COMPLETE_MSG flag. As I see it, the only reason for an application to maintain the offset manually is if there is a requirement to PUT segments to the queue out of sequence. Since this is such a rare situation, stay away from maintaining the offset yourself.

In my opinion there is little reason for using application segmentation. If you are going to implement application segmentation on both sides just use message groups. Why add the extra complexity of segmentation to the code? In this case message groups provide the same level of functionality as application segmentation. However, the use of system segmentation is another story. It can easily be used to route messages through intermediated systems and to reduce the system requirements for storing and transmitting very large messages.

Stephen L Marini
Information Infrastructures (USA)

Architecting for performance

This article is aimed at two types of reader:

- Those new to WMQ who want to understand the design of an WMQ system and who want to choose the best message attributes for various message processing scenarios.
- Those who understand the essence and practicalities of performance and want to improve the responsiveness of their queue managers.

Solving business problems with MQSeries requires input from various skilled MQ professionals, usually including a performance architect, a system administrator, and an application programmer. Successfully achieving a performance target with a suite of MQSeries application programs depends on the performance architect substituting lightweight resources (eg memory) for heavyweight resources (eg CPU, disk, network communications, and interprocess communication). Heavyweight resources should only be used where absolutely necessary, otherwise performance bottlenecks can occur.

All software systems have bottlenecks that ultimately limit their performance. The performance architect aims to use both lightweight and heavyweight resources, enabling the system to process the workload as efficiently as possible and optimize the utilization of those resources. To achieve this it is paramount that the performance architect understands the relationships and dependencies between resources labelled as lightweight and heavyweight.

DESIGNING FOR PARALLELISM AND SCALABILITY

Running parts of the message processing scenario in parallel can help increase utilization of CPUs, disks, and logical constructs inside the software system, thus providing enhanced scalability.

In order for messages to be retrieved from a queue in the same sequence as they were delivered to it (or to maintain integrity when retrieving messages by priority) it is necessary for the queue manager to service requests to that queue one at a time. This is controlled by the queue manager and is referred to as serialization. The required degree of serialization influences the way in which additional hardware can provide scalability in the number of messages per second that can be processed on a queue.

A particular request on a queue may require exclusive access to one or more system resources (this will be discussed later). The time elapsed for exclusive access servicing the request directly correlates to the performance limit of message throughput for that queue. A performance limit can be exposed by just a few message processing applications, according to the type of resource necessary to service a request. However, this may not be reached until thousands of messaging applications are serialized by the queue manager when attempting to access the same queue simultaneously. The MQSeries performance architect tries to ensure that messaging applications are designed to work in parallel with each other and with multiple instances of applications. This helps to achieve the highest message throughput that the system is capable of processing.

If an application processes messages synchronously, submitting a request message to a queue and then waiting for a response before submitting the next message, the application is serialized on itself (it is perfectly legitimate to design an application in this way). With the high degree of serialization in this scenario it is only possible to improve performance by using faster processors, faster disks, and/or faster network communications. This is because there is little parallelism involved in the various stages of processing the message.

As the message moves from one stage to the next the previous stage becomes inactive and the message must flow through to the last stage before a new message can be processed from the beginning. If each processing stage is utilized constantly by several new messages entering the system, all stages can operate in parallel with each other and performance can be improved significantly.

To sustain a high message throughput rate multiple instances of the same application or multiple threads inside a single application can be used. Either faster processors or more processors can provide a performance enhancement and an expensive disk subsystem (eg cached disks) may be unnecessary. For example, if processing persistent messages, faster processors allow more data to be written into the log buffer between successive updates to the log (this will be discussed later). Each log update happens more quickly and there can be a huge benefit in the responsiveness of the existing disk hardware.

Similar scalability characteristics apply to both server and client message processing applications. Multiple instances and/or multiple threads can be more beneficial to server applications since a single server application can be processing messages from several thousand client applications located on different workstations.

MESSAGE RECOVERY MODEL

MQSeries was first designed to advocate a ‘once-only assured delivery’ paradigm. The message application was able to place complete trust in MQSeries to deliver a mission-critical message at some point in the future. The message was logged on each and every MQSeries queue manager in the system as it flowed through the queue manager network, such that the message would survive a queue manager restart caused by either a software or hardware failure. A message with these properties is referred to as a persistent message.

An alternative type of message, which is less resource-intensive, is the non-persistent message. Non-persistent messages will be handled by the queue manager an order of magnitude quicker since no logging of the message is performed. Non-persistent messages do not survive a queue manager restart; however, the messages will arrive at their destination provided the queue manager continues to operate. Nevertheless, non-persistent messages are entirely suitable for user-driven synchronous applications where messages represent queries, and delays in receiving a response do not inhibit other business processes.

PERFORMANCE COMPENSATION FOR NON-RECOVERABILITY

Non-persistent messages flow faster through a queue manager system because they are not logged as they are moved between queues or queue managers. The queue manager initially stores non-persistent messages in volatile memory, overflowing the messages to the file system (buffers and disk) when there is a shortage of memory. Even if the message has been overflowed to the file system it still does not survive a queue manager restart. If the queue manager is operating normally and a non-persistent message does not arrive at its expected destination it can always be located *en route* on a transmission or dead letter queue, for example.

Normally, non-persistent messages are also moved faster over channels. The default non-persistent message speed (NPMSPEED) over server channels is 'FAST' and it allows non-persistent messages to be transmitted over the channel outside of message channel agent (MCA) syncpoint control. The MCA uses syncpoint control to guarantee the recoverability of persistent messages.

Non-persistent messages transmitted outside of syncpoint control appear on the destination queue as soon as the receiving MCA takes them off the channel. Persistent messages are not available to other application programs until the batch is closed. It should be noted that non-persistent messages still contribute to the number configured as the channel batch size attribute (BATCHSZ).

The speed and number of processors in a queue manager machine are the most important factors in message throughput. However, if other resources are utilized to their limits (eg the system is short of real memory, too much data is being written to the disks, a maximum data rate is being transmitted over a network interface, or network bandwidth is saturated by other systems), message throughput can bottleneck and the CPU resources cannot be used effectively.

PERFORMANCE OVERHEAD FOR RECOVERABILITY

Units of work

Persistent messages processed outside of syncpoint control (ie they are not included in a unit of work) must wait for the log record generated by the queue manager to be successfully written to the disk and the message placed on or retrieved from a queue before control is returned to the application.

The queue manager serializes requests on the queue while the log record is written to the log disk and the message is written to the queue file. This restricts a single queue to one update for each log write. Several application instances or threads can still process messages in parallel if they each use different queues. The queue manager can then combine all generated log records, for many queues, into a single write to the log disk.

Persistent messages processed inside of syncpoint control (ie they are included in a unit of work with other messages) also generate log records. However, the applications using syncpoint control are permitted to continue processing messages until they issue an MQCMIT call. Only then is the application blocked until the log records from the messages in the unit of work have been written to the log disk. Again, the queue manager can combine all generated log records for all the messages into a single write.

The queue manager serializes requests on a single queue in the same way for outside syncpoint control as inside syncpoint control. Inside of syncpoint control the serialization is of a much shorter duration. For each write to the log many updates can be made to a single queue (by a single application) inside of syncpoint control, whereas only one update to a queue is possible (by any application) outside of syncpoint control.

The WMQ performance architect should ensure that persistent messages are processed inside of syncpoint control where possible, to optimize data transfer to the log disk, thereby maximizing the utilization of this heavyweight resource.

Logging and checkpointing

The log records generated by processing persistent messages are first written into the log buffer. Inside of syncpoint control the MQCMIT call blocks the application from processing more messages (and thus generating more log records) but does not block other applications from processing messages. The new log records are appended to the log buffer, which allows the queue manager to continue collecting data to be written to the log disk while it is writing the data previously collected. This buffering allows the log disk to be used efficiently since the buffer is accumulating data ready to be written to the disk each time it completes a revolution.

The default number of log buffer pages (LogBufferPages) is only practical for a queue manager processing very few persistent messages each second when non-cached disks are holding the log files. The size of the log buffer should normally be configured to its maximum size of 2MB (512 pages).

Periodically, the queue manager rolls forward the point of recovery (referred to as checkpointing) in case the queue manager should need to be restarted. In order to restart as efficiently as possible, the queue manager synchronizes the queue files with the log. When the point of recovery is rolled forward log extents required for recovery (which were marked active for this purpose) are marked as inactive.

For circular logging the inactive log extents can then be reused. For linear logging the inactive logs can be archived. By default, checkpointing occurs every 10,000 log operations. Long running units of work that span checkpoints may cause several log extents to remain active. In these circumstances it may be necessary to configure more than the default number of three primary logs when the queue manager is created .

Speed and number of processors

Each time a write to the log disk completes, all the applications that were waiting for the log records in that I/O are unblocked. The queue manager then continues with writing the next log write for all the new log records accumulated during the previous log I/O. The number of log records that can be written into the log buffer each second is related to:

- The speed and number of processors.
- The number of application threads that are able to generate log records in parallel.
- The number of messages processed by the application instances or threads inside, as opposed to outside, of syncpoint control.

The number of log buffer pages written to the log disk each second is related to the speed and number of processors, the latency of the log disk, and the I/O subsystem bandwidth. A bottleneck in persistent messaging can be indicative of insufficient or ineffective parallelism in the application design and may not be attributable to either the queue manager or the log disk. Using a performance monitoring tool and detecting a high disk utilization (% busy) does not imply that disk throughput has reached a limit or that it is not possible further to increase persistent message throughput.

It is physically possible to perform only a certain number of I/Os to a disk each second. If each time the disk rotates a small amount of data is transferred there is maximum utilization but minimal transfer. This will show a statistic of 100% busy but the number of blocks written to the disk is small. A typical 10,000 rpm SCSI disk can sustain 8MB data transfer per second and rotates at a frequency of 166 times a second.

Persistent message throughput can be optimized by ensuring that there are always log records waiting in the buffer each time it is written to disk. To achieve this there must be CPU cycles available for writing log records to the log buffer and for each disk revolution there needs to be 50K of data in the log buffer. This represents a target that is rarely achieved in practice.

PHYSICAL LOG DEVICE

Speed and latency of the log disk

During the rotation time period of the log disk the number of log records written into the buffer primarily depends on the degree of parallelism achieved by the application architecture. Using a single application for processing messages outside of syncpoint, special disk hardware (eg RAID and/or software striped SCSI disks) does not help improve disk throughput. This is because the total size of the log write is insignificant compared with the minimum size of a stripe. However, cached disks (eg SSA with fast-write cache) can reduce the latency to between one and two milliseconds.

Unsuitability of IDE disks

IDE disks typically offer a non battery-backed cache, which gives no provision for power failure. In these situations the reliability of the message is severely restricted. The I/O latency is comparable to that of SCSI disks but the application only has the resilience offered by non-persistent messages, ie they are still delivered if there is no system failure.

Suitability of solid-state disks

Solid-state disks can be used but nominally provide several gigabytes of storage. In normal operation, with circular logs, the log is a write-only device so the benefit of being able to read information fast is not useful. They are more expensive than write-cacheing disks in financial terms though they do give a similar response time. They are useful for holding queue files if messages are frequently flushed from the file system buffers (when the non-persistent queue buffer overflows for example) and must be retrieved from disk some time later. However, the critical disk resource is normally the log, not the queue files.

Linear logs have three contenders for the log file where the log formatter and log archiver compete with the normal log data-stream. Solid-state disks can allow the archiving of the log extents to other media with minimum interference to the underlying messaging.

QUEUE MANAGER QUEUES

Queue depth and utilization

Queues are not suitable as databases and are designed for short term storage. A queue manager system can process the highest rate of non-persistent messages when there are only a few messages on the queue and they can all be contained inside the non-persistent queue buffer (default size of 64KB). Additional messages are overflowed to the file system buffers and onto the physical queue files.

Persistent messages are contained in the persistent queue buffer (default size of 128KB) and are always written to the physical log. Associating an application thread with its own private queue helps reduce contention on the queue at the expense of real memory. A limited degree of sharing queues processing a low throughput rate (eg ten persistent messages per second per queue) outside of syncpoint control, or a higher rate (eg 500 persistent messages per second per queue) inside of syncpoint control, will give similar performance behaviour.

Dynamic queues

Dynamic queues used as reply queues are often used for response messages only, and can have a very short lifetime. The first time a dynamic queue is opened the queue structure has to be constructed and typically involves many shared memory and file system operations. Closing a dynamic queue normally turns the object to a ‘ghost queue’. When the queue manager subsequently receives a request to create a dynamic queue it may be able to recreate one very quickly. If there is a suitable structure in the pool of ghost queues the queue manager can resurrect one and return a handle immediately. This allows dynamic queues to be created at a much higher frequency than prior to MQSeries V5.2.

Serialized I/O on queue files

It has been stated previously that persistent messages outside of syncpoint control block activity on the queue for the duration of the necessary I/O operations. This predicates that many application instances or threads processing messages outside of syncpoint control should be assigned their own dedicated queue to allow arbitrary permutations of queue updates in the same log I/O.

SUMMARY

We have shown how a queue manager system is sensitive to various optimizations that have a direct impact on throughput. If there are sufficient application instances or threads using the processing power of the system between each forced log write, contention can be reduced on the heavyweight log resource. This is achieved either by processing messages inside of syncpoint control and/or arranging for the application design to use a large enough set of queues. Otherwise, the system may not be able to exploit the CPU resources available. Furthermore, CPU utilization may be at a maximum but the number of cycles required to process each message may not be at a minimum.

Alexander Russell and Peter Toghill
IBM Hursley (UK)

Using the MQRFH2 message header

WebSphere Business Integration for Financial Networks (WBI for FN) is a financial network consolidation platform. It serves as a single point through which financial applications gain access to various networks and message processing services. WBI for FN is based on the WebSphere MQ Integrator or Integrator Broker (WMQI).

WMQI, the technological basis for the WebSphere Business Integration solutions, works with WMQ messaging, extending its basic connectivity and transport capabilities to provide a message broker solution driven by message flows. WBI for FN provides message processing services implemented using message flows and an associated message model. All programs and services external to WBI for FN communicate with WBI for FN services by means of WBI for FN messages. In addition to the mandatory control information in the message descriptor (MQMD) these messages use the Version 2 rules and formatting header (MQRFH2) to convey information that is not part of the message body, for example, attributes of the message, service requests and responses, and properties that represent configuration parameters.

This article provides a basic introduction to the MQRFH2 message header and its use, based on the experience gained through the development of WBI for FN.

PURPOSE

The MQRFH2 evolved from the rules and formatting header (MQRFH), which can be used to send string data in the form of name/value pairs. Unlike its predecessor, the MQRFH2 allows Unicode strings to be transported without translation and can carry numeric data types. It uses a syntax similar to that of the Extensible Markup Language (XML). A single message can potentially contain more than one MQRFH2 structure, though this is generally discouraged.

Currently, the MQRFH2 is exploited by the WMQI Broker and by the WMQ implementation of the Java Messaging Service (JMS) API. In conjunction with WMQI the MQRFH2 serves the following purposes:

- To define the message set to which the message format for the message body belongs.
- To define publish/subscribe flows.

- Through custom extensions, to allow client applications to define fields that can be accessed and processed by customized message processing nodes (such as those used in the WBI for FN services).

STRUCTURE

The MQRFH2 is composed of a fixed and a variable portion, each containing multiple fields. In the C and C++ programming languages its mandatory fixed component can be expressed through the following data structure:

```
 struct tagMQRFH2 {   MQCHAR4 StrucId; /* Structure identifier */
 MQLONG Version;        /* Structure version number */
 MQLONG StrucLength;    /* Total length of MQRFH2 including all
           NameValueLength and NameValueData fields */
 MQLONG Encoding;       /* Numeric encoding of data that follows last
           NameValueData field */
 MQLONG CodedCharSetId; /* Character set identifier of data that
           follows last NameValueData field */
 MQCHAR8 Format;        /* Format name of data that follows last
           NameValueData field */
 MQLONG Flags;          /* Flags */
 MQLONG NameValueCCSID; /* Character set identifier of NameValueData */
};
```

(This code fragment is taken from the *WebSphere MQ Application Programming Reference* manual, which should be consulted for further details.)

As with other MQ message headers, these fields are interpreted in the character set and encoding given by the character set identifier and numeric encoding specified in the preceding header structure or message descriptor.

EXTENSIONS

The fixed part of MQRFH2 is followed by any number of name/value pairs, meaning ordered pairs of the following fields in C/C++ syntax:

```
   MQLONG NameValueLength;  /* Length of NameValueData */
   MQCHARn NameValueData;   /* Name/value data */
```

Here, *n* is the length in bytes of the data in the NameValueData field, as specified in NameValueLength. The MQCHARn data type is only defined for the following values of *n*: 4, 8, 12, 20, 28, 32, 48, 64, 128, 256. If a different length is required, use a string or array construct of type MQCHAR.

The length of bytes of the NameValueData string must always be a multiple of four. If the size of the data that must be accommodated is not a multiple, that data must be padded with blanks to fill the string. You cannot use a null character to end the string prematurely as is common practice in C.

Unlike the fixed part of the structure and other MQ message headers, NameValueData is generally not interpreted in the character set specified in the preceding structure. Rather, its character set is given by the NameValueCCSID field in the fixed part of the MQRFH2, which is currently limited to the values 1200, 1208, 13488, and 17584.

Except when using the UTF-8 character set (CCSID 1208), which is independent of the numeric encoding, NameValueData, like all MQRFH2 fields, is interpreted in the encoding specified in the preceding structure. Although its encoding is observed, NameValueData is not converted when the message is retrieved from its local queue (for example, using the MQGET function call) even if MQGMO_CONVERT is specified as a message-data option.

FOLDER SYNTAX VERSUS XML

NameValueData is the most important and powerful MQRFH2 field because it contains structured data in a simple markup language comparable to XML. Each NameValueData field contains a folder, for example:

```
<ComIbmDni> ... </ComIbmDni>
```

The folder's boundaries, like those of an XML element, must be delimited by XML start and end tags (in the example, <ComIbmDni> and </ComIbmDni>). The XML empty-element tag cannot be used, although empty folders are allowed. The names of the start and end tags must match and denote the name of the folder (in the example, ComIbmDni), similar to an XML element type. In contrast, with an XML document the folder cannot begin with an XML declaration; because there can also be no document type declaration (and thus no document type) a folder can never be a valid XML document, although it might be well-formed.

The name of the folder used in the example (ComIbmDni) is constructed from the top-level domain name 'Com', followed by the company name 'Ibm' and the internal product identifier 'Dni'.

Any characters following the end tag in the NameValueData string must be blank. Therefore, each string cannot contain more than one folder. Additional folders can be specified in separate NameValueData fields, each preceded by its corresponding NameValueLength field.

Each folder, between its start and end tags, can contain any number of groups, properties, or both, in any order.

A group, like a folder, is delimited by XML start and end tags, for example:

```
<Dnf> ... </Dnf>
```

Again, the name of a group is determined by the name of its start and end tags (in the example, Dnf). Like folders, groups can contain properties or additional nested groups. Therefore, a group can be regarded and used as a 'sub-folder' for properties that are related by type or function. (In the example, the folder contains properties used to control the WBI for FN Extension for SWIFTNet, which has the component code Dnf.) There is no limit to the nesting level of groups and any number of groups can share the same name within an MQRFH2.

A property, like a folder or group, is delimited by start and end tags that define its name, but can contain an optional value (instead of groups or other properties), for example, where 'Failed' is the value:

```
<Code>Failed</Code>
```

A property and a group within a folder cannot share the same name independently of their relation. This restriction does not apply to properties and groups within different folders, each of which may be regarded as occupying a different name space.

A property value can consist of any characters in the character set specified in NameValueCCSID. As in most XML character data, the ampersand character (‘&’) and the left-angle bracket (‘<’) must be escaped using the strings ‘&amp;’ and ‘&lt;’ respectively. Since XML CDATA sections are not supported in folders there is no need to escape the right angle bracket (‘>’), single quote (‘'’), or double quote (‘"’). However, the escape sequences ‘&gt;’, ‘&apos;’, and ‘&quot;’ are supported in values. Blanks within values are considered significant and should not be removed by WMQ or its applications when processing the MQRFH2 structure. Other blanks within a folder are insignificant.

In addition to its value a property can contain an optional data type; for example, where boolean is the data type:

```
<IsStatus dt='boolean'>0</IsStatus>
```

The data type can be used to restrict the type of values that a property can take or to signal to the application programmer or user that a value is to be interpreted as being of a certain type. In the example, explicitly specifying the boolean data type elucidates the meaning of the value 0 to the reader although it will generally not affect the behaviour of an application that uses this property. Refer to the *WebSphere MQ Application Programming Reference* or *WebSphere MQ Integrator Programming Guide* for a complete list of supported data types and their use.

As in XML, a name (of a folder, group, or property) is a token beginning with a letter or one of a few allowed punctuation characters and continuing with letters, digits, hyphens, underscores, or full stops. Note that the colon (‘:’), which in XML is used for the namespace construct, is not allowed in a name. As in XML , names are case-sensitive. Again, please refer to the above publications for a detailed list of Unicode characters that can be used in names.

In spite of the similarity between folders and XML elements there are several XML constructs that cannot be used within folders and cannot easily be translated to the syntax allowed in folders. These unsupported constructs include:

- Comments.
- Processing instructions.
- CDATA sections.
- Declarations.
- White space handling.
- Attribute value normalization.
- Language identification.

Compared with general XML it is another fundamental restriction of the folder syntax that actual content can only be conveyed using property values. Intermixed markup and content as used in the XML element <tag1>value<tag2></tag2></tag1> does not conform to the folder syntax, because tag1 cannot be interpreted as a folder, group, or property.

USING THE MQRFH2 WITH WMQI

After receiving a message containing an MQRFH2, WMQI invokes its MQRFH2 parser to interpret the header. In addition to the fixed part of the MQRFH2 this parser lets you navigate and manipulate the folder structures contained in NameValueData fields. According to the conventions defined by WMQI the MQRFH2 can also carry a description of the message contents and publish/subscribe commands.

To refer to application data contained in a folder the logical message model provided by WMQI can be used. According to this model the MQRFH2 folders, groups, and properties are syntax elements of the logical message. (More precisely, folders and groups are name elements, whereas properties are value elements.) Because the relationship between these syntax elements is hierarchical the resulting structure is called the message tree.

The WMQI message flow nodes provide an interface called Extended Structured Query Language (ESQL) to query and update the message tree. For example, a reference to the ClientSend property in the folder shown in Listing one would be Root.MQRFH2.ComIbmDni.Dnf.SAG.SAGHeader.EnvMode.ClientSend (where Root is the root of the message tree).

Listing one: a sample ComIbmDni folder as used with WBI for FN

```
<ComIbmDni>
   <Version>1.Ø</Version>
   <OU>DNFSYSOU</OU>
   <Dnf>
         <Version>1.Ø</Version>
         <SAG>
               <SAGInstance>
                     <QMgr></QMgr>
                     <RequestQueue></RequestQueue>
               </SAGInstance>
               <SAGHeader>
                     <EnvMode>
                           <ClientSend>NoEnv</ClientSend>
                           <ClientReceive>NoEnv</ClientReceive>
                     </EnvMode>
               </SAGHeader>
         </SAG>
   </Dnf>
</ComIbmDni>
```

A complete discussion of WMQI and ESQL is outside the scope of this article. For details on using the MQRFH2 with WMQI please refer to *WebSphere MQ Integrator Working with Messages* and *WebSphere MQ Integrator ESQL Reference*.

Elmar Meyer zu Bexten
IBM Germany

Delete channel start process in WMQ 5.3 for z/OS

With versions up to WebSphere MQ V5.2 a process definition with APPLIDID('CSQX START') and USERDATA ('your.channel.name') is required for channel triggering.

By specifying this process and INITQ(SYSTEM.CHANNEL.INITQ), together with TRIGGER(YES), TRIGTYPE(FIRST) in the transmission queue definition, the channel is started by the MQSeries channel initiator (CHIN) when messages appear on the transmission queue. With WMQ V5.3 the process definition is no longer needed because the channel name can be specified directly in the TRIGDATA field of the transmission queue. For example:

```
DEFINE QLOCAL(your.xmit.queue) USAGE(XMITQ) TRIGTYPE(FIRST) +
TRIGGER(YES) INITQ(SYSTEM.CHANNEL.INITQ) TRIGDATA(your.channel.name)
```

The old way using a process is still supported so there is no essential need to change the definitions, except to get rid of some process definitions. Depending on the number of queuemanagers and channels there may be a lot of objects to change, maybe too many to do it manually. So I wrote a REXX program named CHLPROC to create the required commands for purging processes and changing transmission queues.

CHLPROC processes the output produced by CSQUTIL display commands and creates a ‘delete process’ and an ‘alter qlocal’ command when process and xmitq meet the following conditions:

- The process contains APPLICID(CSQX START) and a nonempty USERDATA.
- The qlocal is of USAGE(XMITQ), INITQ(SYSTEM. CHANNEL.INITQ), nonempty and existing PROCESS() that matches the above process conditions, empty TRIGDATA.

If QSGDISP(COPY) is found in the process or the xmitq definition the proper command will be created with QSGDISP(GROUP). In the case of incomplete or missing definitions CHLPROC will issue error messages. I recommend you do the following:

- Process the output of queue managers singly.
- Back up your object definitions using the CSQUTIL MAKEDEF function before applying the commands created by CHLPROC.
- Do not apply the commands if there is the chance of fallback from V5.3 to the previously used version.
- Check the COMMANDS first before processing them.

For more information about channel triggering check the *WebSphere MQ Intercommunication* manual, Chapter 24: *Preparing WebSphere MQ for z/OS*.

CHLPROC REXX

```
/* REXX */
/* CREATE COMMANDS TO GET RID OF CHANNEL START PROCESSES         */
/* COLLECT PROCESS INFORMATION FIRST */
/* */
SAY ''
PPROCESS  = 1
PQSGDISP  = 2
PAPPLICID = 3
PUSERDATA = 4
PARMP.Ø   = 4
PARMP.PPROCESS  = 'PROCESS('
PARMP.PQSGDISP  = 'QSGDISP('
PARMP.PAPPLICID = 'APPLICID('
PARMP.PUSERDATA = 'USERDATA('
/* */
SAY 'READING PROCESS LIST'
ADDRESS MVS 'EXECIO * DISKR ' PROCESSL
IF RC <> Ø THEN DO
  SAY 'ERROR DURING READ - RC=' RC
  EXIT
END
SAY 'RECORDS READ: ' QUEUED()
/* */
INERR = Ø
DO QUEUED()
  PULL CARD
  W1 = STRIP(CARD)
  /* GET OBJECT ATTRIBUTES */
  DO I = 1 TO PARMP.Ø
    IF INDEX(W1,PARMP.I) > Ø THEN DO
```

```
      P1 = INDEX(W1,PARMP.I) + LENGTH(PARMP.I)
      P2 = LENGTH(W1)
      VALUE.I = STRIP(SUBSTR(W1,P1,P2-P1))
      /* LAST ATTRIBUTE FOUND? */
      IF I = PUSERDATA THEN DO
        PNAME = VALUE.PPROCESS
        /* IS THIS A CHANNEL START PROCESS? */
        IF VALUE.PAPPLICID = 'CSQX START' THEN DO
          /* IS A CHANNELNAME SPECIFIED? */
          IF VALUE.PUSERDATA <> '' THEN DO
            /* REMEMBER VALUES */
            QSGDISP.PNAME = VALUE.PQSGDISP
            APPLICID.PNAME = VALUE.PAPPLICID
            CHANNEL.PNAME = VALUE.PUSERDATA
          END
          ELSE DO
            /* CSQX START BUT NO CHANNELNAME IN PROCESS DEFINITION */
            SAY 'NO CHANNELNAME SPECIFIED IN PROCESS 'PNAME
            INERR = INERR + 1
          END
        END
      END
    END
  END
END
/* */
/* NOW GET THE XMIT QUEUES */
/* */
QQUEUE    = 1
QQSGDISP  = 2
QUSAGE    = 3
QPROCESS  = 4
QINITQ    = 5
QTRIGDATA = 6
PARMQ.Ø   = 6
PARMQ.QQUEUE    = 'QUEUE('
PARMQ.QQSGDISP  = 'QSGDISP('
PARMQ.QUSAGE    = 'USAGE('
PARMQ.QPROCESS  = 'PROCESS('
PARMQ.QINITQ    = 'INITQ('
PARMQ.QTRIGDATA = 'TRIGDATA('
SAY 'READING QUEUE LIST'
ADDRESS MVS 'EXECIO * DISKR ' QUEUEL
IF RC <> Ø THEN DO
  SAY 'ERROR DURING READ - RC=' RC
  EXIT
END
SAY 'RECORDS READ: ' QUEUED()
CC = Ø
PURGE = Ø
```

```
ALTER = Ø
DO QUEUED()
  PULL CARD
  W1 = STRIP(CARD)
  /* GET OBJECT ATTRIBUTES */
  DO I = 1 TO PARMQ.Ø
    IF INDEX(W1,PARMQ.I) > Ø THEN DO
      P1 = INDEX(W1,PARMQ.I) + LENGTH(PARMQ.I)
      P2 = LENGTH(W1)
      VALUE.I = STRIP(SUBSTR(W1,P1,P2-P1))
      /* LAST ATTRIBUTE FOUND? */
      IF I = QTRIGDATA THEN DO
        /* IS THIS QLOCAL A XMITQ? */
        IF VALUE.QUSAGE = 'XMITQ' THEN DO
          QNAME = VALUE.QQUEUE
          /* DO NOT PROCESS THESE QUEUES (WILL RESULT IN ERROR MSGS) */
          IF QNAME = 'SYSTEM.CLUSTER.TRANSMIT.QUEUE' !  ,
             QNAME = 'SYSTEM.QSG.TRANSMIT.QUEUE' THEN DO
             ITERATE
          END
          /* IS THE INITQ SYSTEM.CHANNEL.INITQ FOR THIS XMITQ? */
          IF VALUE.QINITQ = 'SYSTEM.CHANNEL.INITQ' THEN DO
            /* IS THERE A PROCESS NAME? */
            IF VALUE.QPROCESS <> '' THEN DO
              /* IS THE TRIGGER DATA EMPTY? */
              IF VALUE.QTRIGDATA =  '' THEN DO
                /* CHECK FOR RELATED PROCESS */
                PNAME = VALUE.QPROCESS
                IF APPLICID.PNAME = 'CSQX START' THEN DO
                  /* BUILD COMMANDS */
                  ZW3 = ''
                  IF QSGDISP.PNAME = 'COPY' THEN DO
                    ZW3 = ' QSGDISP(GROUP) '
                  END
                  CC = CC + 1
                  CMD.CC = 'DELETE PROCESS('!!PNAME!!')'!!ZW3
                  PURGE = PURGE + 1
                  CC = CC + 1
                  CMD.CC = 'ALTER QLOCAL('!!QNAME!!') FORCE + '
                  CC = CC + 1
                  CMD.CC = '  PROCESS('' '') + '
                  ZW3 = ''
                  IF VALUE.QQSGDISP = 'COPY' THEN DO
                    ZW3 = ' QSGDISP(GROUP) '
                  END
                  ELSE DO
                    IF VALUE.QQSGDISP = 'SHARED' THEN DO
                      ZW3 = ' QSGDISP(SHARED) '
                    END
                  END
                  CC = CC + 1
                  CMD.CC = '  TRIGDATA('''!!CHANNEL.PNAME!!''')'!!ZW3
                  CC = CC + 1
                  CMD.CC = '*'
                  ALTER = ALTER + 1
                END
                ELSE DO
                  SAY 'NO VALID PROCESS FOUND FOR XMITQ '!!QNAME
                  INERR = INERR + 1
                END
              END
              ELSE DO
                SAY 'TRIGDATA NOT EMPTY FOR XMITQ '!!QNAME
                INERR = INERR + 1
              END
            END
            ELSE DO
              SAY 'NO PROCESS SPECIFIED IN XMITQ '!!QNAME
              INERR = INERR + 1
            END
          END
          ELSE DO
            SAY 'SYSTEM.CHANNEL.INITQ NOT SPECIFIED IN XMITQ '!!QNAME
            INERR = INERR + 1
          END
        END
      END
    END
  END
END
/* */
SAY 'WRITING COMMAND FILE'
CMD.Ø = CC
'EXECIO * DISKW COMMANDS (STEM CMD. FINIS'
IF RC <> Ø THEN DO
  SAY ' ERROR DURING WRITE - RC = 'RC
  EXIT
END
/* */
SAY ''
SAY 'PROCESS PURGE COMMANDS CREATED: 'PURGE
SAY 'XMITQ ALTER COMMANDS CREATED:   'ALTER
SAY 'XMITQ/PROCESS IN ERROR          'INERR
SAY ''
EXIT
```

CHLPROC EXECUTION JCL

```
// your jobcard goes here
//PROCESS  EXEC PGM=CSQUTIL,PARM='QMGR'
```

```
//SYSPRINT DD DSN=&&TEMPP,DISP=(NEW,PASS),SPACE=(CYL,(8,8))
//SYSIN    DD *
COMMAND DDNAME(CMD) TGTQMGR(QMGR)
//CMD      DD *
DISPLAY PROCESS(*) APPLICID USERDATA
//QUEUES   EXEC PGM=CSQUTIL,PARM='QMGR'
//SYSPRINT DD DSN=&&TEMPQ,DISP=(NEW,PASS),SPACE=(CYL,(8,8))
//SYSIN    DD *
COMMAND DDNAME(CMD) TGTQMGR(QMGR)
//CMD      DD *
DISPLAY QLOCAL(*) USAGE PROCESS TRIGDATA INITQ
//CHLPROC  EXEC PGM=IKJEFTØ1,DYNAMNBR=3Ø,REGION=ØM
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSIN    DD DUMMY
//PROCESSL DD   DSN=&&TEMPP,DISP=(OLD,DELETE)
//QUEUEL   DD   DSN=&&TEMPQ,DISP=(OLD,DELETE)
//COMMANDS DD   SYSOUT=*
//SYSTSIN  DD *
 EXEC 'your.library(CHLPROC)'
/*
```

SAMPLE CHLPROC OUTPUT

```
READING PROCESS LIST
RECORDS READ:  26Ø
NO CHANNELNAME SPECIFIED IN PROCESS QMA.CHANNEL.PROCESS
READING QUEUE LIST
RECORDS READ:  3Ø55
SYSTEM.CHANNEL.INITQ NOT SPECIFIED IN XMITQ QMB
NO PROCESS SPECIFIED IN XMITQ QMC
NO VALID PROCESS FOUND FOR XMITQ QMD
WRITING COMMAND FILE
PROCESS PURGE COMMANDS CREATED: 3
XMITQ ALTER COMMANDS CREATED:   3
XMITQ/PROCESS IN ERROR          4
Sample COMMANDS output for local, group and shared definitions:
DELETE PROCESS(QM2.CHANNEL.PROCESS)
ALTER QLOCAL(QM2) FORCE +
  PROCESS(' ') +
  TRIGDATA('QM1.TO.QM2')
DELETE PROCESS(QM3.CHANNEL.PROCESS) QSGDISP(GROUP)
ALTER QLOCAL(QM3) FORCE +
  PROCESS(' ') +
  TRIGDATA('GROUP.TO.QM3') QSGDISP(GROUP)
DELETE PROCESS(QM4) QSGDISP(GROUP)
ALTER QLOCAL(QM4) FORCE +
  PROCESS(' ') +
  TRIGDATA('QSG.TO.QM4') QSGDISP(SHARED)
```

CSQUTIL EXECUTION WITH COMMANDS

```
// your jobcard goes here
//CSQ      EXEC PGM=CSQUTIL,PARM='QMGR'
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
COMMAND DDNAME(CMD) TGTQMGR(QMGR)
//CMD      DD DISP=SHR,DSN=commands.file.created.by.chlproc
/*
```

Stefan Raabe (stefan.raabe@t-online.de)
Consultant (Germany)

MQ news

iWay, the Information Builders unit, has announced expanded support for IBM's WebSphere product line, including Application Server and the former MQSeries products, WebSphere MQ and WebSphere MQ Integrator (WMQI), with its Universal Adapter Framework.

JCA 1.0-based adapters now support all leading J2EE app servers (from BEA, IBM, Fujitsu, Oracle, and Sun) and the company is introducing JCA 1.5 support with the release of iWay 5.5.

For more information contact:
iWay Software, Two Penn Plaza, New York, NY 10121-2898, USA.
Tel: +1 212 330-1700
Fax: +1 212 564-1726
Web: http://www.iwaysoftware.com

* * *

IBM has launched a portfolio of offerings and programs designed and priced specifically for medium-sized business customers.

The portfolio, IBM says, will consist of new hardware, software, services, solutions and financing, all designed to meet specific criteria for medium-sized businesses with respect to function, ease of use and management, and price – a set of brand attributes delivered under the name Express.

WebSphere Commerce Express is a new software offering which, claims the company, makes it faster, easier, and less expensive for medium-sized businesses to create and manage an e-commerce site, allowing businesses to start building an online store in as little as one hour.

WebSphere MQ – Express is designed to make it easy to connect a variety of different applications together so businesses can efficiently share critical data across their IT infrastructure. A medium-size business can install WebSphere MQ – Express in just five clicks and can be up and running in as little as 10 minutes, says IBM.

For more information contact your local IBM representative.

* * *