MQ update

52

October 2003

In this issue

3 Trouble-shooting using output channel status

7 Using WMQ in J2EE, part 1

24 Advanced MQ channel configuration

28 WBI Message Broker V5.0 Toolkit: an introduction

35 WMQ clusters and shared queues in z/OS

45 Using the XML Transform node in WMQI

49 MQ news

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

Disclaimer

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 £100 ($160) per 1000 words and £50 ($80) per 100 lines of code for the first 200 lines of original material. The remaining code is paid for at the rate of £20 ($32) per 100 lines. To find out more about contributing an article, without any obligation, please 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.

Trouble-shooting using output channel status

In a distributed queueing environment encompassing lots of Windows, Unix, and z/OS queue managers, the channel status is a good source of information when trouble-shooting. Looking at empty queues it is not possible to determine whether an application has already processed the messages that were in the message flow or whether there was ever a message flow at all.

The channel status log can help because it supplies information about the number of messages that have been transferred, a start date and time for the channel, the date and time for the last message that was transferred, and other attributes.

Unfortunately most of the channel status information is available only when the channel is 'running' and is lost when the channel is terminated. Since many people work in international environments, where different time zones are commonplace, it is probable that many messages are transferred when no-one is in the office to trap errors.

For these reasons I thought it would be useful to save some of the channel status information at channel termination time to assist with the analysis of potential errors.

I decided to use a security exit program because the security exit is called for all types of channel. This applies to send and receive exits as well, but these are also called during message transfer, whereas the security exit is called only during channel connection and termination.

CHLSTAT PROGRAM LOGIC

1 At channel termination the exit is driven with the exit reason MQXR_TERM. You should check the proper exit point and the MQCXP/MQCD fields.

2 Get the channel name and conname from MQCD and the partner queue manager name from MQCXP.

3 Open a temporary dynamic queue named SYSTEM.CHLSTAT.*, which will be used as a reply queue for commands, using the supplied model queue SYSTEM.COMMAND.REPLY.MODEL.

4 Put a 'DISPLAY CHSTATUS(channelname) ALL' message to the SYSTEM.COMMAND.INPUT. queue.

5 Read the replies and check for CSQM42 messages. If these are found you should get the status information from the exit and use *wto* to display the messages in the CHIN log if the conname and queue manager name match the CD/CXP values.

6 Repeat step 5 above, until the reply queue is empty.

7 Close the reply queue with the option 'delete-purge, clean-up'.

Other solutions, aside from writing the channel status information to the CHIN log, are possible, eg keeping the reply messages in a queue without reading them or processing the reply messages with message tools. Which method you use will depend upon your own environment and preferences.

I prefer to have all related and required information available in one place so I chose the CHIN log because it is checked for channel start and stop messages.

Use

The exit can be used in any type of channel, regardless of whether it is a sender or a receiver channel. I tested it with SDR, RCVR, SVRCONN, CLUSSDR, and CLUSRCVR channels, but it should work for other channel types too. There are some things to be aware of when using the CHLSTAT exit:

- Seurity exits cannot be chained in the channel definition. If you already use a security exit you have to handle the chaining yourself.
- If there are multiple connections by a single application (SupportPac MO71 works this way, using a type SVRCONN

channel) you get duplicate output because multiple channels with the same name and the same conname (IP address) exist during the termination of one channel.

The exit finds multiple reply messages as a result of the **display chstatus** command and creates an output for each one. Unfortunately I did not find a way to determine which chstatus suppresses the duplication.

- If a sender channel is in retry state the exit is also driven. The exit tries to find a channel status with a proper queue manager name but because there was no connection it does not find any and will issue an error message. Because there is no flag to indicate that the channel is in retry state this error message will be issued wrongly at every retry interval.

  That's why I decided not to show an error message when no queue manager name is found in the channel status. If you miss a channel status output for a terminating channel you may switch on the trace (described later) to determine why it is not shown. Maybe it was in error and suppressed because of the reason described above.

- If you define the exit into a cluster receiver channel the channel definition will be distributed to other cluster queuemanagers, depending on your cluster setup and usage.

  The exit is used to build implicit defined cluster sender channels (CLUSSDRA) also holding the SECEXIT(CHLSTAT) attribute. These implicit defined channels will only work if the queue manager is on z/OS and if the exit is available. If the exit is not available or the queue manager is running on a different platform (eg Windows or Unix) the channel is unable to start because the exit name is specified in a different format and the exit program does not exist. You will need a channel autodefinition exit to remove the security exit name.

- If you want to use the exit in implicit defined cluster sender channels on z/OS you will need a channel autodefinition exit to put the SECEXIT(CHLSTAT) attribute into the channel definition (if it was not specified in the proper cluster receiver channel).

- A **ping channel(..)** command will result in a channel status output. You will also see messages CSQX500I and CSQX501I (Channel ... started/is no longer active) in the CHIN log.
- I get a 2009 returncode (connection broken) in the security exit when disconnecting from the queue manager during the termination processing of a SVRCONN channel. I was not able to find out why, so I just ignored it.
- For testing purposes I included a ‘trace’ function in the exit. Just put the word ‘TRACE’ at the beginning of the channel security exit user data and you will get additional information in the CHIN log, such as MQCXP and MQCD areas, returncodes, replyqname, and so on. To get rid of the trace function within the loadmodule set the ‘&TRACE’ variable to ‘OFF’ at the beginning of the source program and create a new load module. To eliminate the tracing from the source erase all *do_trc* macro calls, the macro itself, and all source code that is marked to be used only for tracing (check the comments within the program source).

COMPILE AND LINK

There is nothing special to be aware of for compilation, just use your favourite Assembler/MQ compile job. For linkedit you have to include CSQXSTUB, which can be found in the SCSQLOAD library.

My sample SYSLIN statements

```
//SCSQLOAD DD DSN=........SCSQLOAD,DISP=SHR
//SYSLIN DD *
  ENTRY CHLSTAT
  INCLUDE SCSQLOAD(CSQXSTUB)
  INCLUDE SYSLIB(CHLSTAT)
  MODE AMODE(31),RMODE(ANY)
  NAME CHLSTAT(R)
/*
```

Sample output from CHLSTAT exit in the CHIN log

```
+CHLSTATIØ1- >>>>> CHANNEL STATUS OUTPUT <<<<<
```

```
+CHLSTATIØ2- CHANNELNAME(QM1.TO.QMA          )
+CHLSTATIØ3-        TYPE(RECEIVER  )   STATUS(STOPPING   )
+CHLSTATIØ4-     CONNAME(19Ø.69.69.69    )
+CHLSTATIØ5-    QMGRNAME(QM1                            )
+CHLSTATIØ8-  START TIME(14:47:3Ø)       DATE(19.Ø3.2ØØ3)
+CHLSTATIØ9- LASTMSGTIME(14:47:3Ø)       DATE(19.Ø3.2ØØ3)
+CHLSTATI1Ø-    LSTSEQNO(        11) CURSEQNO(        11)
+CHLSTATI11-     INDOUBT(NO        )     MSGS(         3)
+CSQX545I MQFT CSQXRESP Channel QM1.TO.QMA closing because
     disconnect interval expired
```

For more information about channel exits please refer to *MQSeries Intercommunication,* Chapter 38*, Channel-exit programs*.

The program CHLSTAT can be found at www.xephon.com/extras/chlstat.txt.

Stefan Raabe
Consultant (Germany)

Using WMQ in J2EE, part 1

OVERVIEW

This article describes the use of WMQ in the Java 2 Enterprise Environment (J2EE) and is part one of two. In this first part the J2EE standard interface for messaging will be described as will the WMQ implementation. In part two, which will be published in the November issue of *MQ Update*, the J2EE will be discussed in more detail as will the integration of WMQ with the messaging options available in IBM's J2EE platform, the WebSphere Application Server.

THE JAVA MESSAGE SERVICE

The Java Message Service (JMS) is a vendor-independent messaging API based on the Java programming language. The specification is owned by Sun Microsystems and was developed with input from all the major providers of messaging software, including IBM. The JMS guarantees provider independence but not provider interoperability.

JMS messaging domains

The JMS has two messaging domains:

- Point-to-point is the type of messaging familiar to all users of WMQ, where the message is received in one location only, from a defined sender.
- Publish/subscribe is more of a broadcast, where subscribers register their interest in topics (a method of partitioning the messages) with a central Broker. The topic space is hierarchical in structure and the use of wildcards is supported when specifying topics to publish or subscribe to.

  Publishers send messages (on a specified topic) to the Broker, which then forwards a copy of the message to every subscriber that has registered an interest in that topic. A message published by an application can, therefore, be received by many subscribers and the subscribers do not have a direct connection to the publisher.

The JMS is structured around a simple class hierarchy, where abstract classes are extended into each messaging domain. This ensures that the programming model for both point-to-point and publish/subscribe is the same.

JMS Administered Objects

The JMS requires some kind of abstraction of the proprietary nature of a provider's messaging software and this is accomplished using JMS Administered Objects:

- The JMS ConnectionFactory is the object that a JMS application uses to create a connection to a JMS provider. The ConnectionFactory is extended by the messaging domains to the QueueConnectionFactory and the TopicConnectionFactory.

- The JMS Destination is the object that a JMS application uses as a target for sending messages and a source for receiving messages. The Destination is extended by the messaging domains to the Queue and the Topic.

JMS Administered Objects are stored in a JNDI namespace by an administrator who must have enough knowledge of the JMS provider specifics to be able to configure them properly. A JMS application will access and use these objects by looking them up in JNDI and, therefore, the provider will remain transparent to the application.

JMS class hierarchy

The JMS class hierarchy is very simple and cascades from the ConnectionFactory:

- JMS Connections are created from JMS ConnectionFactory objects once retrieved from JNDI. They are multi-threaded and encapsulate an active connection to a JMS provider. The Connection is extended by the messaging domains to the QueueConnection and the TopicConnection.
- JMS Sessions are created from JMS Connections. They are single-threaded and create the context for sending and receiving messages with the various transactional and delivery options that this entails. The Session is extended by the messaging domains to the QueueSession and the TopicSession.
- JMS MessageProducers and MessageConsumers are created from JMS Sessions and are used for sending and receiving messages respectively. Multiple producers and consumers can be created from one Session but their operations are serialized because of the single-threaded nature of the Session.

  The MessageProducer is extended by the messaging domains to the QueueSender and the TopicPublisher. The MessageConsumer is extended by the messaging domains to the QueueReceiver and the TopicSubscriber.

- JMS messages to be sent are created from JMS Sessions, whereas Message objects are returned from the various receive methods on the MessageConsumers.

JMS messages

Messages in JMS have a format similar to that of a WMQ message in that specific portions of the message are allocated for carrying context and data separately.

There are three sections to a JMS message:

- The header fields, which contain values used by both providers and applications to identify and route messages.
- The properties, which are used to add optional header fields to a message. There are some JMS standard properties and applications, and providers can also add specific fields to the message header.
- The data portion of the message.

Three of the most important message header fields (instantly recognizable to a WMQ programmer) have been further centralized to the programming model by giving them default values on the MessageProducer. These are:

- DeliveryMode, which determines whether the message is persistent or non-persistent. The JMS specification defines ‘persistent’ to mean that a JMS provider should take extra care to ensure that the message is not lost due to a provider failure. Persistent messages should also be delivered once and only once.
- Priority, which determines whether the message should be expedited over others. Priorities are in the range zero to nine and the JMS specification specifies that messages of priority five and above should be delivered before messages of priority four and below.
- TimeToLive, which determines when the message will expire, starting from the time the message was sent.

There are five types of JMS Message, extended from an abstract message parent class:

- TextMessage, where the message body contains a Java String.
- BytesMessage, where the message body contains raw bytes (provided for legacy code compatibility).
- StreamMessage, where the message body contains an ordered sequence of typed data fields.
- MapMessage, where the message body contains an indexed array of typed data fields.
- ObjectMessage, where the message body contains a serialized Java Object.

Message selectors

A selector can be specified when creating a JMS MessageConsumer, which then retrieves only messages that match the selector. Message selectors are strings containing logical constructs based on the SQL92 syntax and expressions for the examination of message header and property field values.

A message selector matches a message when the selector (with the message header and property field identifiers substituted for their values in the message) evaluates to 'true'. The selector can be empty, in which case all messages are matched.

Acknowledgement, transactions, and durability

A JMS Session can be created to be either non-transacted or transacted. A transacted Session uses a series of transactions, each of which groups messages' send and receive operations into an atomic unit of work. When a transaction is completed it can be either committed or rolled back. In the first case all message operations are finalized, whereas in the second case any messages sent or received are moved back to their previous state. The completion of a transaction is controlled by the application, with the JMS provider maintaining integrity if the application terminates unexpectedly.

If a JMS Session is non-transacted, another message receipt control level is used, message acknowledgement, of which there are three types:

- *Automatic acknowledgement*; each message received is automatically acknowledged to the provider. This is transparent to the application.
- *Duplicates-OK acknowledgement*; each message may be received more than once. This is transparent to the application.
- *Client acknowledgement*; message acknowledgement is controlled by the application. In this case, the application calls an acknowledgement method on a received message, which confirms the receipt of all messages (since the last acknowledgement call) to the JMS provider.

  The message acknowledgement is based on that point in time not on message delivery order. The client can also call a recovery method on the JMS Session, which will cause the redelivery of all unacknowledged messages.

JMS work can also be encapsulated into distributed transactions, according to the X/Open document *Distributed Transaction Processing: The XA Specification*. Details of this will be covered in the second part of this article, published next month.

There is also the concept of robustness – peculiar to JMS publish/subscribe – the durable subscription. Normally in a publish/subscribe framework a subscribing application will deregister its subscriptions with the Broker when it is closed, thus ensuring that no attempt is made to send messages to it while it is off-line. This is a non-durable subscription.

A durable subscription is not deleted at the Broker when the application is terminated (it must be explicitly closed). This means that an application owning a durable subscription will not miss any publications on the topic it is registered against during any downtime.

Asynchronous message delivery

Asynchronous message delivery is a feature of JMS that can be misleading to WMQ practitioners because all WMQ messaging is asynchronous. A better term for 'asynchronous' in this case would be 'passive' and the reasons why will become clear in this section.

Asynchronous message delivery allows the JMS application to register a class (which must implement the MessageListener interface) with the JMS provider. The only method in the MessageListener interface is the onMessage method, which is called by the JMS provider when a message arrives for the receiving application.

This is the passive nature of the message delivery; instead of the application calling a synchronous blocking receiving method on the MessageConsumer, the JMS provider passes the message into the onMessage method of the MessageListener.

Another way of describing this would be to say that the JMS provider drives the receiving application with messages, instead of the receiving application looping an attempted message receive operation.

THE WMQ IMPLEMENTATION OF THE JMS

An important point to be emphasized immediately is the fact that the JMS specification defines no wire format for JMS implementations, either in terms of messages or protocols. This means that different JMS implementations will not communicate directly, thus implying that while JMS applications are portable between different JMS providers the different providers themselves are not compatible at the implementation level.

Underlying WMQ technology

The WMQ implementation of the JMS is built upon the WMQ Classes for Java, which is a proprietary Java API for WMQ. This is transparent in WMQ JMS terms in that there is almost no WMQ Java configuration that can be done through the WMQ JMS interface.

The only other point worth mentioning here is that WMQ Java provides a Connection Pooling facility, which is used transparently by WMQ JMS. WMQ Java Connection Pooling is turned on by default in WMQ JMS but can be turned off at ConnectionFactory level. The pool contains underlying queue manager MQHCONNs, which are pooled and reused to improve performance. The WMQ JMS Session is where the MQHCONN is located; see below for more details of how MQI calls fit into WMQ JMS calls.

As has been previously stated, point-to-point messaging is familiar to all WMQ practitioners and because of this the WMQ JMS point-to-point implementation sits easily on top of standard WMQ. However, using WMQ JMS publish/subscribe needs more in the way of provider configuration.

There are several choices of publish/subscribe mechanism from WMQ and these can all be used for WMQ JMS publish/subscribe:

- The MA0C WMQ Product Extension, which provides add-on publish/subscribe Broker function to a WMQ queue manager.
- The WMQ Event Broker or Integrator Broker products both provide publish/subscribe engines which can be used with WMQ JMS publish/subscribe applications.

In product release terms the WMQ JMS implementation is included in WMQ V5.3 and above and is available as the MA88 Product Extension for MQSeries V5.2.

Underlying MQI calls

The JMS programming model is very simple when looking down from the top but it is often the case that some knowledge is required about the timing of the underlying WMQ calls. Table 1 describes when the standard MQI calls happen during JMS operations.

WMQ JMS Administered Objects

JMS Administered Objects are where the JMS provider masks the proprietary nature of the provider software from the JMS

MQI call	JMS operation
MQCONN	The MQHCONN is held at the WMQ JMS Session level, so the MQCONN would normally occur when calling the createSession method on the Connection. This may not be the case if WMQ Java Connection Pooling is being used, in which case the MQCONN will be called when the pool does not have a connection available and has to create one.
MQDISC	The use of the MQDISC call corresponds to the MQCONN call above. The MQHCONN is held in the WMQ JMS Session so when the Session is closed the MQDISC call would normally be issued. If the Connection Pool is active the MQDISC will happen when the pool is purged of active connections (either through timeout or shutdown).
MQOPEN	WMQ queues are opened either for send or receive (or both) and the implication may (correctly) be drawn that the MQOPEN call is issued when a WMQ JMS MessageProducer or MessageConsumer is created.
MQCLOSE	The converse of MQOPEN; when a WMQ JMS MessageProducer or MessageConsumer is closed the MQCLOSE call will be issued.
MQPUT	MQPUT is called when the send or publish method of a WMQ JMS MessageProducer is invoked.
MQGET	MQGET is called when the receive method of a WMQ JMS MessageConsumer is invoked or the onMessage method of a WMQ JMS MessageListener is called.
MQCMIT	MQCMIT is called when the current unit of work running in a WMQ JMS Session is committed.
MQBACK	MQBACK is called when the current unit of work running in a WMQ JMS Session is rolled back.

Table 1: Timing of standard MQI calls during JMS operations

application. As previously stated, these objects are located in a JNDI namespace (they can be created programmatically but this will destroy the JMS portability of the application). WMQ JMS provides a text-based tool called JMSAdmin for the administration of WMQ JMS Administered Objects, which resembles the **runmqsc** command line tool for WMQ queue manager administration. This tool is driven using the configuration file *JMSAdmin.config,* which contains the following three essential pieces of information to access the JNDI namespace:

- PROVIDER_URL contains the location of the JNDI server in standard format, which varies according to the Initial Context Factory type.
- INITIAL_CONTEXT_FACTORY contains the name of the class that will be used to instantiate the JNDI administration. This will be provided by the JNDI implementation and can be based on a basic file system context, an LDAP context, a WebSphere CosNaming context, etc.
- SECURITY_AUTHENTICATION contains any user authentication information required to access the JNDI server.

There are two useful JNDI tools available for download:

- A GUI interface for the JMSAdmin tool, the MS0N Product Extension.
- A Queue Manager InitialContextFactory implementation, the ME01 Product Extension.

WMQ implementations of the JMS Administered Objects are as follows:

- MQQueue and MQTopic, representing the JMS Destination objects.
- MQQueueConnectionFactory and MQTopicConnectionFactory, representing the JMS ConnectionFactory objects.
- Extensions of the ConnectionFactory objects required for

WAS V3.5.3 and V4.*x* (JMSAdmin is not required for WAS V5.0):

– JMSWrapMQXAQueueConnectionFactory

– JMSWrapMQXATopicConnectionFactory.

WMQ JMS programming techniques

Before proceeding with WMQ-specific JMS programming there are several very important provider-independent JMS programming points that must be stressed. These are listed below.

- The JMS Session is the most important of the JMS classes. It is an edict of the JMS specification that the Session is single-threaded and this rule must be adhered to rigorously.
- The JMS Connection, Session, MessageProducer, and MessageConsumer objects should all be closed by the application programmer (in the reverse order to which they were created) because they may use provider resources whose automatic removal by garbage collection cannot be guaranteed.
- If a JMS MessageListener object has been specified to receive messages asynchronously, either with a JMS Session or a JMS MessageConsumer, the entire Session is marked for asynchronous delivery only.

  Calling any of the synchronous methods for receiving messages on any JMS MessageConsumer created from the JMS Session in question is a programming error and will result in an Exception. This does not affect the sending of messages.
- It is always recommended best practice to register a JMS ExceptionListener against all JMS Connections but especially when JMS MessageListeners are being used for passive message delivery. In this case there will be no way for the application to know if there is a serious JMS error because there will be no synchronous method calls happening, and hence no exceptions can be thrown. The ExceptionListener

allows the JMS provider to notify a passive application that there has been a serious error on the JMS Connection.

- Message selectors can quickly become bloated and cumbersome and this can severely affect performance. If the header of every JMS Message has to be parsed and compared with a complicated selector then message delivery rate will be slowed. Careful planning of the use of selectors is recommended to avoid this.

Some of the advice above becomes more obvious when put into a WMQ-specific context. The first two points in particular should be considered from the WMQ programming standpoint as well.

- The WMQ implementation of the JMS Session holds a base WMQ MQMessage object (from the WMQ Java base classes). This is reused for each message sent and received from all MessageProducers and MessageConsumers created from the Session.

  This gives a considerable performance improvement (as opposed to creating a new MQMessage object each time) and is perfectly safe, providing that the Session is not accessed from multiple threads at once. There is no safeguard against this – no exception will be thrown if it is attempted – but if operations on the Session are not serialized they will interfere with each others' use of the WMQ MQMessage object, with unpredictable results.

- There is another facet to the single-threading of JMS Sessions and this is to do with concurrent sending and receiving of messages. If both senders and receivers are created from the same JMS Session, calls to their methods must be serialized – that is to say a message cannot be sent if a receive call is underway and *vice versa*.

  This is not a problem if synchronous receive is being used because calls to the send and receive methods can be controlled from within the application. However, if a JMS MessageListener is being used it is recommended that messages are not sent from that JMS Session because there is no control (from the application) over when a message will be received.

- As Table 1 illustrates, the WMQ implementations of all the JMS objects contain handles to WMQ queue manager resources. If the WMQ JMS objects are not closed correctly then these WMQ resources will be left locked and will not be cleaned up immediately. If the load on WMQ JMS is high it is not uncommon, because of lack of care when closing JMS objects, for the queue manager to overflow agent processes.

  Closing a WMQ JMS Connection object should cause the closure to cascade to all JMS objects created from that Connection, but best practice is to close explicitly all JMS objects in the reverse order to which they were created. This is even more important when publish/subscribe is being used because stale subscriptions will persist on the Brokers indefinitely unless they are manually deleted.

There is a similarity between WMQ messages and JMS messages in that there is a defined boundary between message context information held in the header and the message payload. However, an examination of the JMS specification will show that there are some parts of the JMS message header that do not correspond directly to fields in the MQMD.

The WMQ implementation of JMS messages uses the MQRFH2 extended header to hold the extra JMS information. As a result it may be that WMQ JMS applications cannot communicate with WMQ legacy applications if they are not designed to use the MQRFH2.

This is resolved by a configuration option of the WMQ JMS Destination, the 'Target Client' property, which can be set to either of the following two values:

- MQJMS_CLIENT_JMS_COMPLIANT: this is the default value and specifies that the messages should be put to the Destination with the MQRFH2 (and hence all JMS information) intact.

- MQJMS_CLIENT_NONJMS_MQ: this specifies that the messages should be put to the Destination with the MQRFH2 removed. This also removes some of the JMS information and leads to the message either being output with an MQMD Format of MQFMT_STRING (in the case of a JMS TextMessage) or with an MQMD Format of MQFMT_NONE (in the case of all other messages).

This parameter only affects messages being output from WMQ JMS – it has no bearing on messages input into WMQ JMS. If WMQ JMS receives a message that does not have an MQRFH2 a best effort is made to construct as much JMS information as possible by extrapolation from the existing message data. For instance, an incoming message with an MQMD Format of MQFMT_STRING will be reconstituted with WMQ JMS as a JMS TextMessage.

WMQ JMS Publish/Subscribe

There are many other options on the WMQ JMS Destination and ConnectionFactory objects but the only other ones that will be discussed here are those relating to publish/subscribe. There are several WMQ implementations of the publish/subscribe function:

- WMQ Publish/Subscribe is a WMQ Product Extension (MA0C). It is a set of programs that installs into the WMQ executable directory and manages publications and subscriptions by using specially defined system queues. This product will go out of service at the end of 2004.
- WMQ Integrator Broker is a member of the WMQ product family that provides application integration by offering message transformation and routing. As part of the routing function it implements publish/subscribe capability in the same manner as the MA0C Broker (using special system queues on the underlying queue manager).
- WMQ Event Broker is similar to Integrator Broker except that the complex message transformation function is not included. It offers the same publish/subscribe method as Integrator

Broker and also offers direct IP connections to the Broker from WMQ JMS. This offers much greater performance at the expense of some reliability (the publish/subscribe paradigm lends itself to this kind of messaging).

WMQ understands publication and subscription messages by using the MQRFH and MQRFH2 extended messaging headers. MA0C understands the MQRFH only, whereas WMQ Integrator Broker and Event Broker understand the MQRFH2 directly (although they are backwards-compatible with the MQRFH).

This means that the WMQ JMS TopicConnectionFactory must have a configuration option to specify which header to use. This is the 'Broker Version' parameter, which can take the following two values:

- MQJMS_BROKER_V1: this is 'compatibility mode' and it specifies that the Broker requires the MQRFH. In this case WMQ JMS messages will contain an MQRFH with the publication or subscription information immediately followed by an MQRFH2 containing JMS information. This option must be set if the MA0C Broker is to be used but it can also be used for WMQ Integrator Broker or Event Broker.
- MQJMS_BROKER_V2: this is 'native mode' and it specifies that the Broker requires the MQRFH2 only. In this case, WMQ JMS messages will contain only the MQRFH2, which will contain publication or subscription information as well as all JMS information. This option cannot be used with the MA0C Broker and is recommended for WMQ Integrator Broker or Event Broker.

As previously mentioned, WMQ Event Broker offers a direct IP connection for WMQ JMS Publish/Subscribe. This is configured by the 'Transport Type' parameter on the WMQ TopicConnectionFactory, which can take one of the following three values:

- MQJMS_TP_BINDINGS_MQ: this specifies that the application will connect directly to a queue manager (for publish/subscribe or not) on the same machine.

- MQJMS_TP_CLIENT_MQ_TCPIP: this specifies that the application will connect to a remote queue manager (for publish/subscribe or not) over a TCP/IP connection. If this option is set, a channel name, a hostname, and a port also need to be specified.
- MQJMS_TP_DIRECT_TCPIP: this specifies that the application will connect directly to an Event Broker over TCP/IP without using any intermediate queues. If this option is set, a hostname and a port also need to be specified.

The direct IP connection with WMQ Event Broker does not support durable subscriptions. This is because of the latency possible with a durable subscription: if a durable subscriber is not connected the messages it is to receive need to be held and this requires a queue. Persistent messages are also not supported with direct IP connections to Event Broker – this is because there is no logging of the message involved and the message will not be hardened to disk.

WMQ JMS also offers the option of shared or exclusive queues for JMS subscribers. These can be mixed on the same Broker and their use will be determined by a performance *versus* resources assessment. If subscriptions share a queue for messages, less resource will be used than if some (or all) subscriptions use one queue exclusively. These options are configured at the WMQ JMS TopicConnectionFactory and Topic levels. This function is not relevant with direct IP connections to WMQ Event Broker.

WMQ JMS environment configuration

The environment for WMQ JMS can be fairly complicated. What follows is a basic description of the minimum environment variable settings on a Windows platform.

The PATH must contain the following:

- C:\Program Files\IBM\WebSphere MQ\Java\lib;

The CLASSPATH must contain the following:

- C:\Program Files\IBM\WebSphere MQ\Java\lib;
- C:\Program Files\IBM\WebSphere MQ\Java\lib\jndi.jar;
- C:\Program Files\IBM\WebSphere MQ\Java\lib\providerutil.jar;
- A path to a JNDI provider implementation, such as fscontext.jar or ldap.jar.
- C:\Program Files\IBM\WebSphere MQ\Java\lib\jms.jar;
- C:\Program Files\IBM\WebSphere MQ\Java\lib\jta.jar;
- C:\Program Files\IBM\WebSphere MQ\Java\lib\connector.jar;
- C:\Program Files\IBM\WebSphere MQ\Java\lib\com.ibm.mq.jar;
- C:\Program Files\IBM\WebSphere MQ\Java\lib\com.ibm.mqjms.jar;

These all assume the default installation directory on the Windows platform. On the Unix platforms the CLASSPATH has similar settings but the inclusion to the PATH variable above should be added to the LIBPATH or the LD_LIBRARY_PATH, depending on the platform.

WMQ JMS problem determination

WMQ JMS offers tracing and logging to aid with problem determination or debugging. These are both enabled from the command line with the following options:

- MQJMS_TRACE_LEVEL: this can be set to ‘on’ or ‘base’. The former will trace JMS calls only and the latter will trace JMS calls and the underlying WMQ Java calls.
- MQJMS_TRACE_DIR: this determines the directory to store the trace files.
- MQJMS_LOG_DIR: setting this parameter will redirect serious errors (usually related to configuration rather than programming)

to a log file. These errors are usually written to standard error output.

An example of using these options is as follows:

```
C:> java -DMQJMS_TRACE_LEVEL=on -DMQJMS_TRACE_DIR=C:\temp -
DMQJMS_LOG_DIR=C:\temp JmsProg
```

(This article continues next month, in the November issue of *MQ Update*, with a more detailed discussion of the J2EE and the integration of WMQ with the WebSphere Application Server (WAS) messaging options.)

Ewan Withers
IBM Hursley (UK)

Advanced MQ channel configuration

Channels are one of the first topics any new MQ administrator encounters, and exchanging messages over a textbook case of a receiver/sender channel pair is the MQ equivalent of 'Hello World'. Once this is mastered attention usually turns to the topics of tuning, securing, and triggering the channels.

Well-tuned and triggered receiver/sender pairs are the workhorses of the MQ world and in many cases these are all that are ever needed. As a result requester and server channels are often overlooked and misunderstood. This article will attempt to explain some of the finer points of requester and server channels as well as dispel some commonly held misconceptions about them.

So let's clear the biggest hurdle right out of the gate; although it is customary to discuss channels in terms of receiver/sender and requester/server pairs it is perfectly acceptable to pair up a receiver and a server or a requester and a sender. Feel free to mix and match any inbound channel with any outbound channel as required. The key is to understand exactly how the channel types work in order to avoid any unintended consequences.

INBOUND CHANNELS

Receiver and requester channels are the same in most respects. Both can be started remotely by either a sender or a server channel and both support multiple simultaneous instances attached to different remote nodes. The main difference is that a requester channel can be used to start its remote partner definition, regardless of whether that definition is of a sender or a server type. This is a point that is often misunderstood so it bears repeating – a requester channel can be used with and will start either a server or a sender channel.

The advantages of requester channels are most obvious when the MQ administrator has no access to the remote node. Perhaps the remote node is in another department, inside the DMZ, or even at a business partner site. In these cases when the channel refuses to start it is difficult to diagnose the problem. The ability to start the channel from the receiving side assists the diagnosis and, in the case of a triggering problem, can provide a workaround to get the messages quickly flowing again.

Functionally, a receiver channel is a subset of a requester channel. If all the receivers in a shop were replaced with requesters the network would operate exactly as before and it's possible that nobody would notice the difference. (If you do this remember to reset the sequence numbers on all the senders and servers!) The shop would gain the ability to start the channels from both ends and the CONNAME parameters would provide an additional element of documentation on the configuration of the network.

OUTBOUND CHANNELS

So if a requester can start either a sender or a server what is the difference between them? Not much if they are started locally. In this case both channel types will attempt to establish communication to an MQ node at the address and port specified in the CONNAME. Since they require exclusive use of their XMITQ both channel types are limited to one running instance at a time. The real difference between sender and server channels is observed when a requester starts the channel from the remote node.

To illustrate, consider three queue managers, QMA, QMB, and QMC. One instance each of sender and server channel types are created on QMA, each with CONNAME(QMB). Matching requester channels with CONNAME(QMA) are created on both QMB and QMC. The result of starting the channels on QMB is illustrated in Figure 1. Both channels on QMA connect to QMB as expected. The interesting behaviour occurs when we use a requester channel on QMC to start the sender or server channels on QMA. We'll examine the two cases separately, beginning with the sender.

When the QMA.QMB.SDR channel is started on QMC the channel contacts QMA and passes the start request to it. When the request comes in to QMA the sender channel is started using the address defined locally in the CONNAME. Even though the request came from QMC, the channel will connect to QMB.

Figure 2 shows the interaction between the channels. This behaviour can be useful. For example, consider a shop with a dozen queue managers and a requirement to start all the channels every Monday morning at 7am. This would normally be accomplished with scripted automation across all twelve queue managers. Using requester channels the automation could be consolidated onto a single queue manager.

Figure 3 shows what happens when the server channel is started from QMC. Unlike the sender channel the server will respond to the node that started it. In this case the server channel on QMA connects to QMC despite the fact that the CONNAME points to QMB. This characteristic allows the server channel to connect to any number of remote nodes without a configuration change, although it is limited to a single running instance at any given moment.

This might be useful if QMC is a contingency server for QMB. In the event that QMB dies, QMC can take over the server channel at QMA. Be aware though that *any* queue manager can start that server channel. A rogue queue manager could hijack the channel and divert messages away from their intended destination.

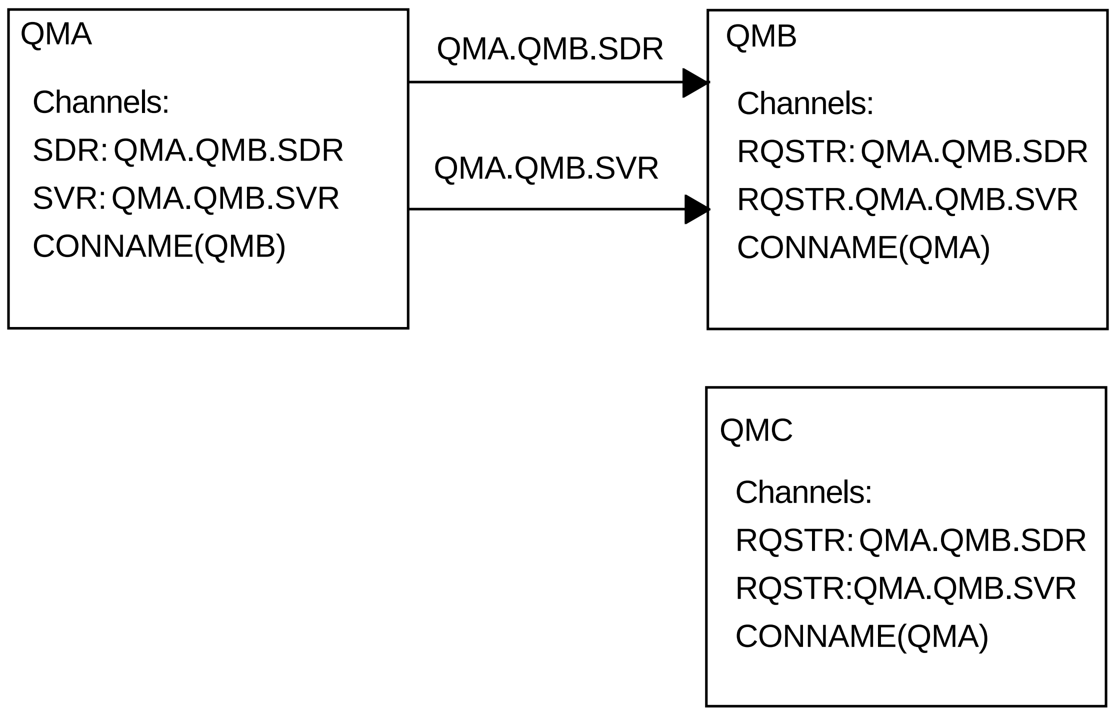

Figure 1: Starting the QMB channels

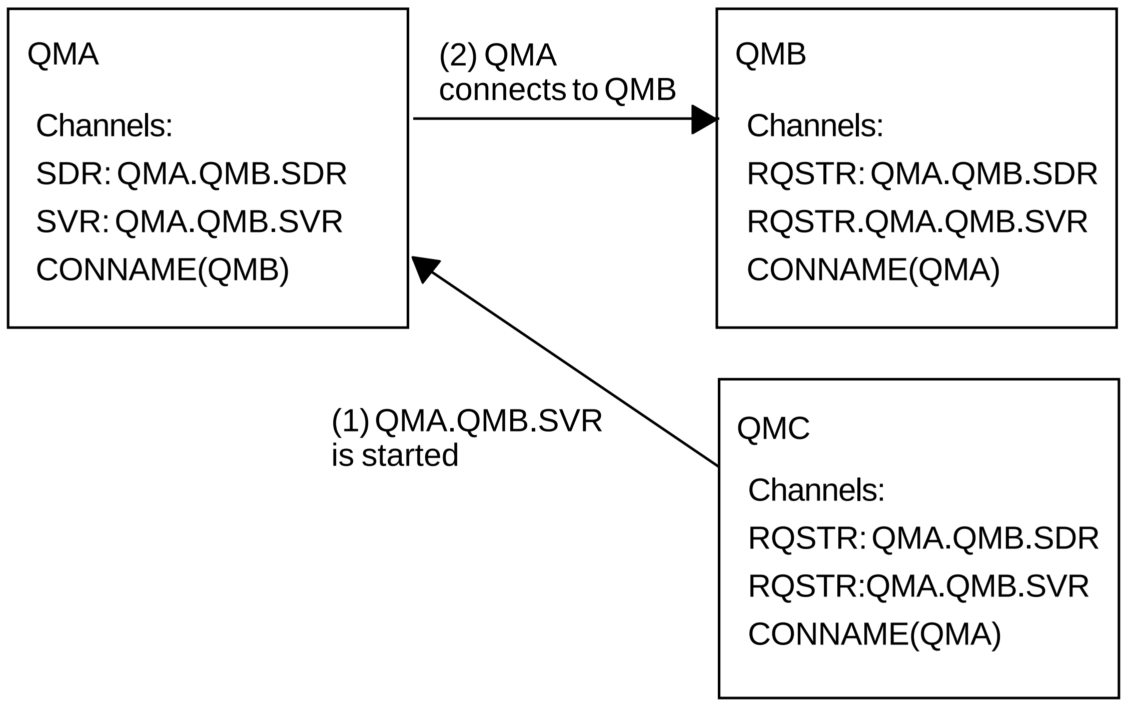

Figure 2: Starting the sender channel from QMC

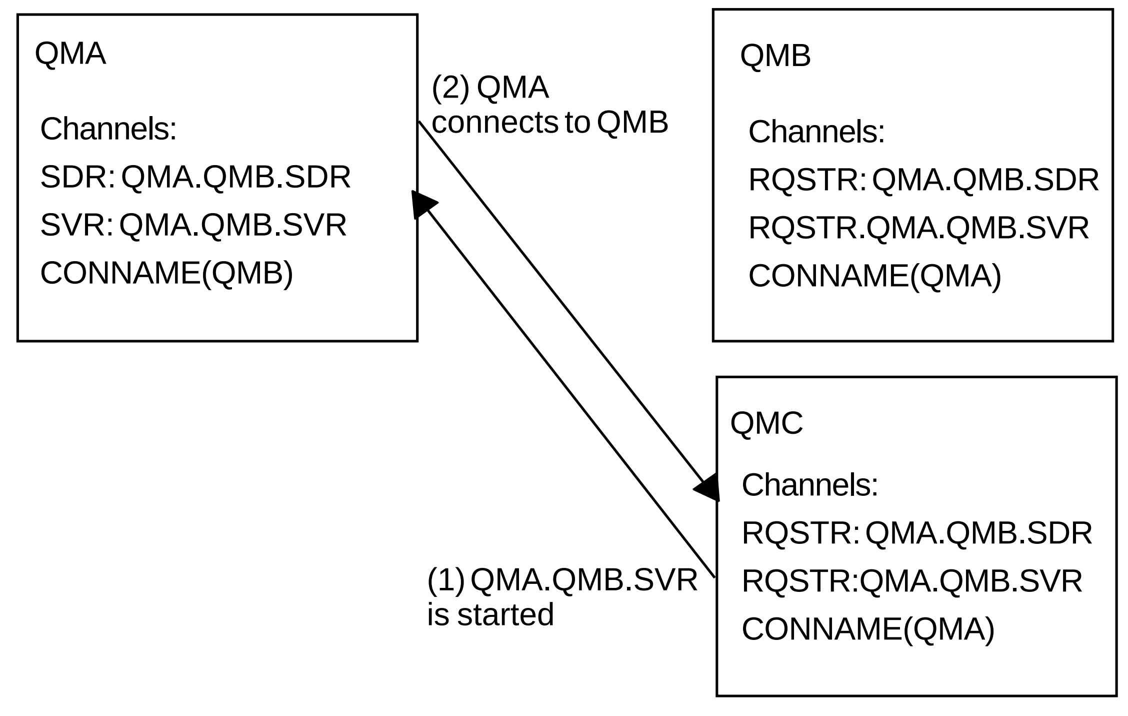

Figure 3: Starting the server channel from QMC

For this reason it is important to have a clear understanding of how server channels work and to use them only in the specific situations where they provide an advantage. Consider using SSL or a channel exit to restrict server channels to authorized partner nodes.

T Robert Wyatt (USA)

WBI Message Broker V5.0 Toolkit: an introduction

This article is aimed at users who have some working experience and knowledge of WMQ Integrator V2.1.

The tooling for this Message Broker is based on the Eclipse framework. Most entities created are stored as files and must belong to a project of some kind. This leads to a major revamp in the process of designing, creating, assigning, and deploying message flows and message sets. Message Broker V5.0 also sees the introduction of the Domain Connection to a Configuration Manager, BAR File Editor, and Perspectives. Access to repositories with version control is also another major addition to the toolkit's functionality.

This article is intended to provide a brief step-by-step guide to creating and deploying a message flow. This will provide an insight into the new tooling and some of its features.

REQUIREMENTS

The following setup is required before proceeding:

- Creation of a Configuration Manager with its queue manager and listener running.
- Creation of a Broker with its queue manager and listener running. (A simpler setup is required if the Configuration Manager and Broker share the same queue manager and listener, which is what we shall use in this example.)

PERSPECTIVES

All tasks that can be performed in Message Broker V5.0 are grouped into views known as perspectives. For example, in order to create and work with a Broker the user has to work in the Broker Administration perspective (Broker Topology, Assignments, Topics, Subscriptions, Operations, and Log Tabs for WMQI V2.1 users). The perspectives categorically separate various related tasks that can be performed in the toolkit, thereby providing an organized structure. Some activities and tasks can overlap by being accessible from more than one perspective. We go on to discuss which perspectives to use for various tasks.

(Note: the following steps show only one way of performing a task in the toolkit. There are a number of different methods available that will perform the same task.)

CREATE A DOMAIN CONNECTION

Message Broker V5.0 requires the user to create a connection to a domain, which is basically a connection to a Configuration Manager running on the system or on a remote system. The domain stores the values of the name of the Configuration Manager's queue manager, the hostname, port number, and also any security exits, if needed. The Domain Connection created has to be part of a Server Project and can be seen in the Broker Administration perspective.

Once connected the Domain Connection is shown as a separate entity in the Broker Administration Navigator pane in the Broker Administration perspective, under Domain Connections, which can be deleted and modified. In other words it can be seen as a wire connecting the Message Broker V5.0 Toolkit and the Configuration Manager.

(Note: the Domain Connection can also be seen and modified in the Broker Application Development perspective within the Resource Navigator pane.)

The following steps explain how to create a Domain Connection.

1. Start the Message Broker V5.0 Toolkit (known as the Control Centre in WMQI V2.1).
2. Select Window, Open Perspective, Broker Administration. (If Broker Administration is not in the list then select Other… and select Broker Administration from the Select Perspective window list that appears.)
3. Select File, New, Domain.
4. Enter the queue manager name, host name, and the port number details used by the Configuration Manager on the system. Click Next.
5. Enter a server project name, which the connection will belong to, and then a name for the connection to the Configuration Manager. Click Finish. (An existing server project can also be selected from the drop-down list.)

6 The Domain Connection name will appear in the Broker Administration Navigator pane under Domain Connections. It will be a member of the server project created in step 5 with *.configmgr* as the extension.

7 In the Domains pane the details of the Domain Connection appear with the various editors.

The properties of the Domain Connection can be seen in the Properties pane in the Broker Administration perspective.

CREATE A BROKER

The concept of a Broker is the same as with WMQI V2.1 although the steps involved in creating a Broker in the toolkit differ. A Broker can be created and seen in the Domains pane of the Broker Administration perspective. Although a Broker can be created using the File menu from any perspective it can only be modified in the Broker Administration perspective. A Broker will be created with one default Execution Group. The Broker will need at least one Execution Group at all times. A user will not be allowed to delete an Execution Group if it is the only one belonging to a Broker.

(Note: a Broker created in the toolkit is merely a reference to a Broker existing physically on the system. A Broker reference can be created without the Broker existing physically on the system. The actual Broker can be created from the command line using the **mqsicreateBroker** command.)

The following steps explain how to create a Broker.

1 Select File, New, Broker.

2 Select the domain created in the previous section and enter values for the Broker and Queue Manager Name fields. Click Finish.

3 The Broker created will appear in the Domains pane under Broker Topology with a default Execution Group.

The properties of the Broker created can be viewed in the Properties pane and its status in the Alerts pane.

(Note: the Alerts pane will show the Broker and Execution Group with a status of 'not running' if the Broker and Execution Group have not been deployed.)

These panes can be opened using the Window, Show View, Alerts and the Window, Show View, Properties menus.

CREATE A MESSAGE FLOW

A message flow can be created in the Broker Application Development perspective. The message flow has to belong to a message flow project and should be within a schema. The concept of a schema has been introduced to provide a structural grouping for message flows. For example, in a scenario where there are many message flows, flows based on a similar type of logic can be grouped into one schema.

(Note: a message flow should not be created in a server or message set project as it will not belong to a schema and hence cannot be deployed.)

The following steps explain how to create a message flow.

1 Select Window, Open Perspective, Broker Application Development. (If Broker Application Development is not in the list then select Other… and select Broker Application Development from the Select Perspective window list that appears.)

2 Select File, New, Message Flow Project. Enter a name for your project. Check the Use default box if it is not checked. Click the Finish button. The project created will appear in the Resource Navigator pane.

3 Select File, New, Message Flow. Enter the name of the message flow project created in step 2 or look for it using the Browse... button. Enter a name for the message flow. Leave the schema field blank as the default schema will be used.

Click Finish. The message flow created will appear in the Resource Navigator pane as a member of a default schema under the Message Flow Project name created in step 2 above.

4 The Message Flow Editor opens once the message flow is created. Design a simple message flow consisting of an MQInput and MQOutput node connected together. Make sure the values for the queue names are correct and save the message flow from the File menu.

Although this feature is similar to the V2.1 Message Flow designer there are some changes. For example, in V5.0 the user has to use the Selection option to drag and drop to put nodes on the flow editor canvas as opposed to the drag and drop feature in V2.1. Also, to connect nodes on the canvas the user has to select the Connection option.

The properties of the message flow can be seen in the Properties pane of the Broker Administration perspective.

(Note: a message flow cannot be deleted while in the Broker Administration Perspective. The user has to switch to the Broker Application Development Perspective to be able to delete the message flow.)

PERFORM A UNIT DEPLOY (RUN ON SERVER)

A unit deploy is a quick method of deploying a unit of a project. This method allows the user to deploy just one unit without having to deploy the entire project or other related projects with it. This method will dynamically create a server specified by the user to run/deploy the unit to. This is also very useful in a scenario where multiple developers are creating flows to be deployed to one Broker. Each developer can then deploy their flows and message sets separately. A deployable project can have multiple message flows and message sets to deploy, which can be called units.

In the following steps a message flow will be deployed to illustrate the working. This can be done from either the Broker Administration perspective or the Broker Application Development perspective.

(Note: a complete project can also be considered as a single unit if desired, and deployed to by the Run On Server option.)

The following steps explain how to perform a unit deploy.

1 Right-click on the message flow created previously and select Run on Server… from the list.

2 Click the Create a new Server radio button and highlight Broker Unit Test Execution Group in the Server Type section. Click Next.

3 Click on the Use an Existing Configuration Manager Connection File radio button. The drop-down box will contain the name of the connection to the Configuration Manager. Click Next.

4 Select the target Execution Group. Click Next. (A new target Execution Group can also be created.)

5 Enter a name for the Execution Group to be deployed to if it is blank. Click Next.

6 Click on the check box of the flow project created in the previous section. To see the flow selected click on the flow project. The message flow selected will be displayed in the right hand pane. Click Finish.

7 A publishing dialogue appears, giving information about publishing to a default server. Once the publishing is finished. Click OK.

Check for the deploy outcome and then check the following to see if the deploy was successful:

1 The message flow should appear under the Execution Group it was deployed to under Broker Topology in the Domains pane of the Broker Administration perspective.

2 Once deployed, the message flow is started so the alerts pane should have no messages about the Execution Group not running.

3 In the Domains pane open the Event Log editor by double-

clicking on it. There should be two successful deploy messages indicating successful deploy; BIP4040I and BIP2056I.

4 Check the system event viewer for errors; fix them if found and deploy again.

The above step-by-step routine will enable a user to become familiar with the new tooling and the way it works. As mentioned before, the tooling provides more than one way of performing a task. One such example is a BAR (Broker Archive File) deploy, which is another method of deploying to a Broker/Execution Group in the toolkit (but is outside the scope of this article). The above mentioned steps are an easier and quicker way to get started with the Message Broker Toolkit for WebSphere Studio.

Rohit Basin
IBM Hursley (UK)

WMQ clusters and shared queues in z/OS

This article examines how application architecture can be expanded to exploit WMQ clusters or shared queues.

Figure 1 is a simplified diagram of a WMQ and CICS configuration, where:

- RQST.CICS represents a queue that receives messages to be processed by CICS.
- A Queue Owning Region (QOR) monitors the queue depth.
- When messages arrive to the queue the QOR uses the CICSPlex System Manager (CPSM) to schedule transactions on the Application Owning Regions (AORs). CPSM will load the AORs evenly.

The problem with this configuration is that the LPAR1 (Logically Partitioned mode) constitutes a single point of failure. By using

WMQ clusters it is possible to have a configuration that:

- Does not have a single point of failure (the application remains available even if an LPAR is lost).
- Balances the workload across LPARs.

WMQ CLUSTERS

WMQ clusters, available with MQSeries for OS/390 V2.1, provide two main functions:

- Automatic propagation of queue definitions.
- Workload balancing.

Automatic propagation of queue definitions

Referring to Figure 2, let's assume that the z/OS queue managers are in a cluster. Users can define a CICS bridge queue (RQST.CICS) in each of queue managers QMGR1, QMGR2, and QMGR3 (with a different CICS region accessing each queue), and make the queue 'shared' in the cluster. In cluster terms, each queue manager owns an instance of the queue.

Suppose that there is another queue manager, QMGR0. This queue manager belongs to the cluster but does not own an instance of the queue. Applications connect to this queue manager and put messages to the CICS bridge queue.

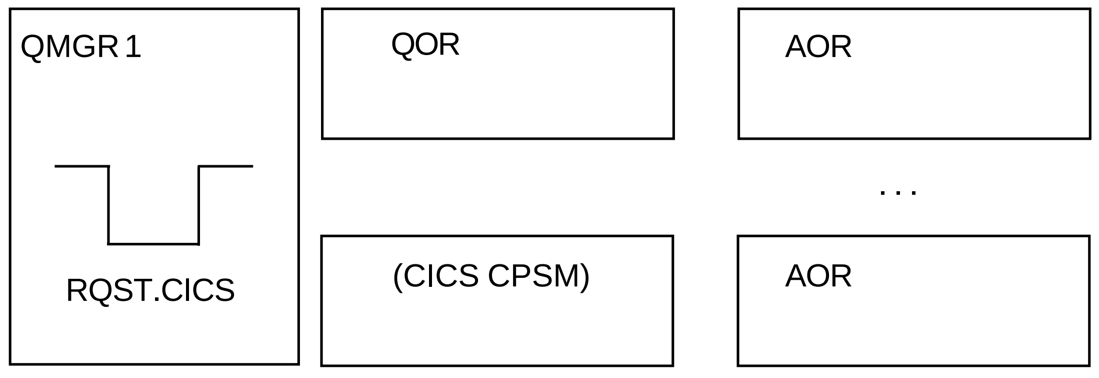

Figure 1: A simplified WMQ and CICS configuration

When an application puts a message to the bridge queue for the first time, the cluster automatically finds which queue managers own instances of the queue. Users do not need to create remote queue definitions in QMGR0. This illustrates the automatic propagation of queue definitions.

Workload balancing in clusters

Because QMGR0 does not own an instance of the destination queue the workload balancing mechanism distributes the load evenly across the other three queue managers QMGR1, QMGR2, and QMGR3. If QMGR0 owned an instance of the queue the message would go to that local instance. This is the default

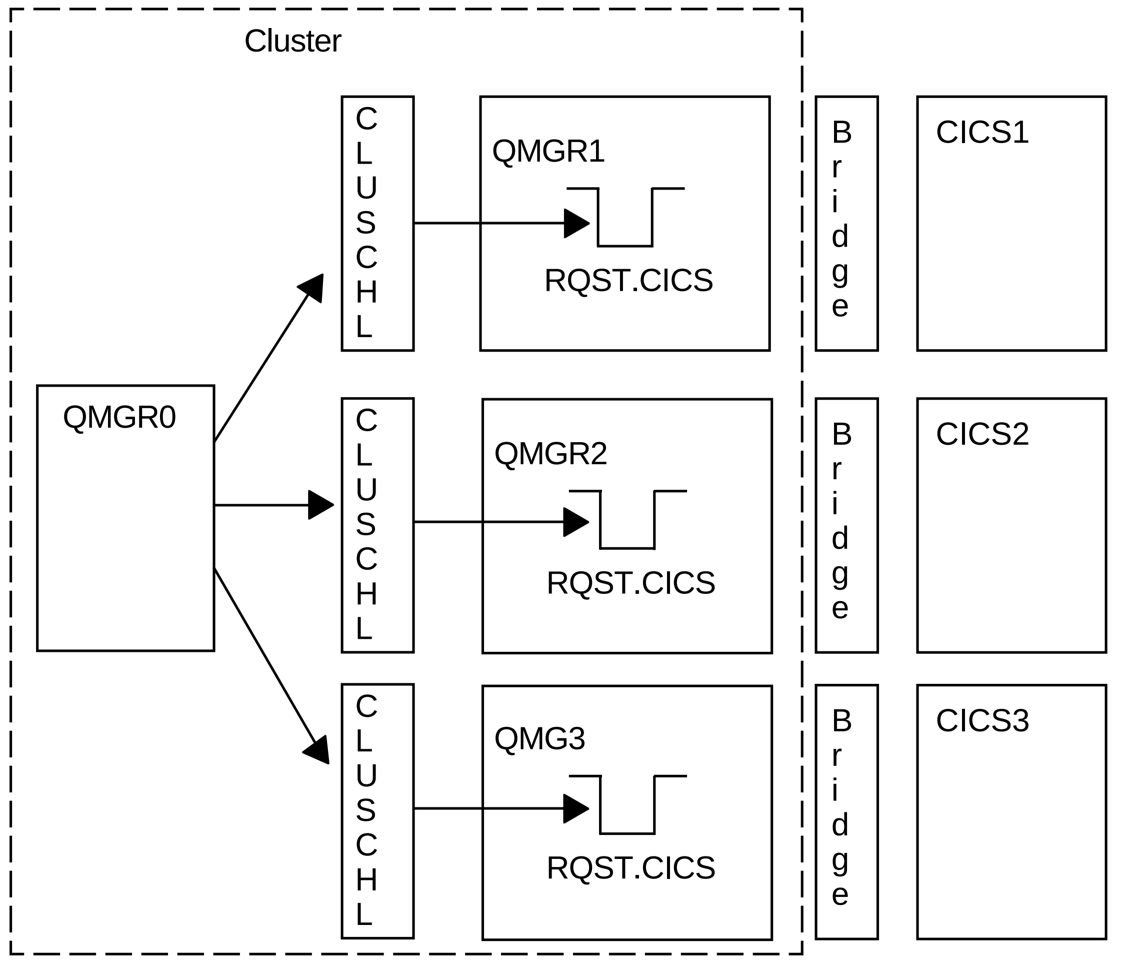

Figure 2: Automatic propagation of queue definitions

behaviour of the cluster and does not require the coding of exits.

It is possible to alter the default behaviour of the workload balancing mechanism by coding a Cluster Workload Exit. This would eliminate the need for ‘extra’ queue managers that do not own instances of target queues.

Cluster components

Repository

A cluster consists of at least one, preferably two, queue managers that have a full repository containing information about the queue managers, queues, and the channels that belong to that cluster. They also hold additional update information on requests from the other queue managers.

The other queue managers have a partial repository and hold information for the subset of queues and queue managers with which they need to communicate. A partial repository is created by the queue manager making inquiries when it first needs to access another queue or queue manager and subsequently requesting that it be notified of any new information concerning that queue or queue manager.

A full repository is updated when the queue manager that owns it receives new information from one of the queue managers in the cluster. This new information is also sent to the other repository to reduce the single point of failure if a repository queue manager breaks down.

The repository information is stored by each queue manager in messages on the SYSTEM.CLUSTER.REPOSITORY.QUEUE and the repository information is exchanged in messages on a queue called SYSTEM.CLUSTER.COMMAND.QUEUE.

Channels

Each queue manager that joins the cluster only needs to define a cluster sender channel (CLUSSDR) to one of the repositories. Once it does this it immediately learns which other queue managers in the cluster hold full repositories.

CLUSTER BENEFITS

Figures 3, 4, and 5 show examples of different WMQ clusters.

Remote queue managers

Figure 3 shows a WMQ cluster, where:

- The configuration achieves workload balancing and does not have a single point of failure.
- Any remote queue managers that are part of the cluster will put messages to the CICS queues in a ‘round-robin’ fashion, spreading the workload evenly across all LPARs.
- Applications connected to the queue managers in the LPARs (either locally or as clients) will put messages to the local queues. That is, an application connected to QMGR1 will put messages on the local queue instance that belongs to QMGR1.

The points above describe the default behaviour in a cluster. It is possible to enforce a round-robin algorithm (even if an instance of a local queue exists) by using a Workload Management Exit.

From a performance point of view it is better practice to allow the default behaviour and achieve workload balancing by evenly distributing the applications that connect to the LPAR queue managers.

Clients

Figure 4 illustrates how WMQ client applications fit within a WMQ cluster. There are two specific points to note regarding this configuration:

- It is necessary to have logic in the applications that will select a queue manager for connection in such a way that connections are evenly distributed across all queue managers (the double arrows represent connections).
- If a queue manager becomes unavailable the applications must reconnect to one of the surviving queue managers.

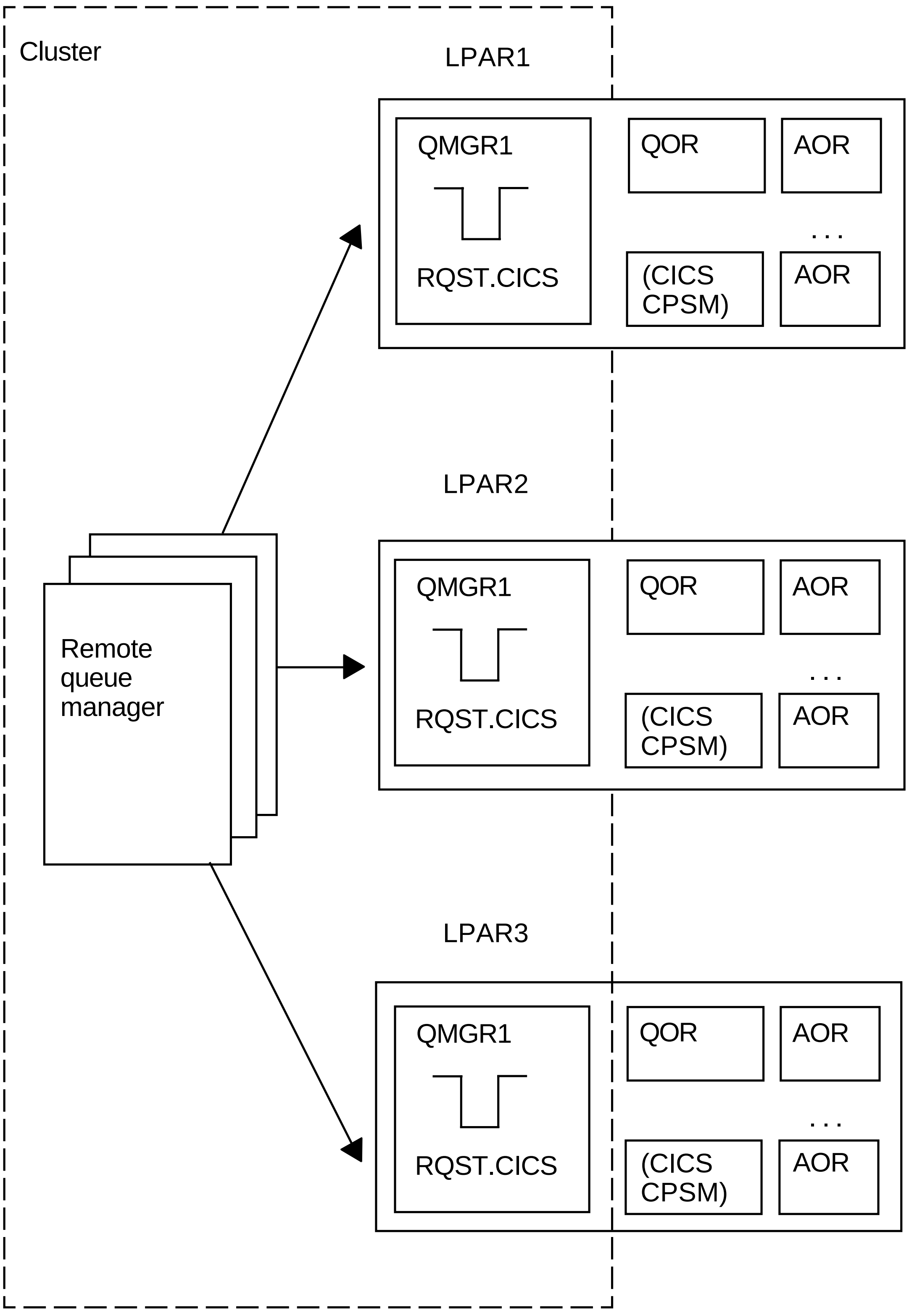

Figure 3: Remote queue managers

The configuration shown in Figure 4 achieves workload balancing and does not have a single point of failure but it does have two disadvantages:

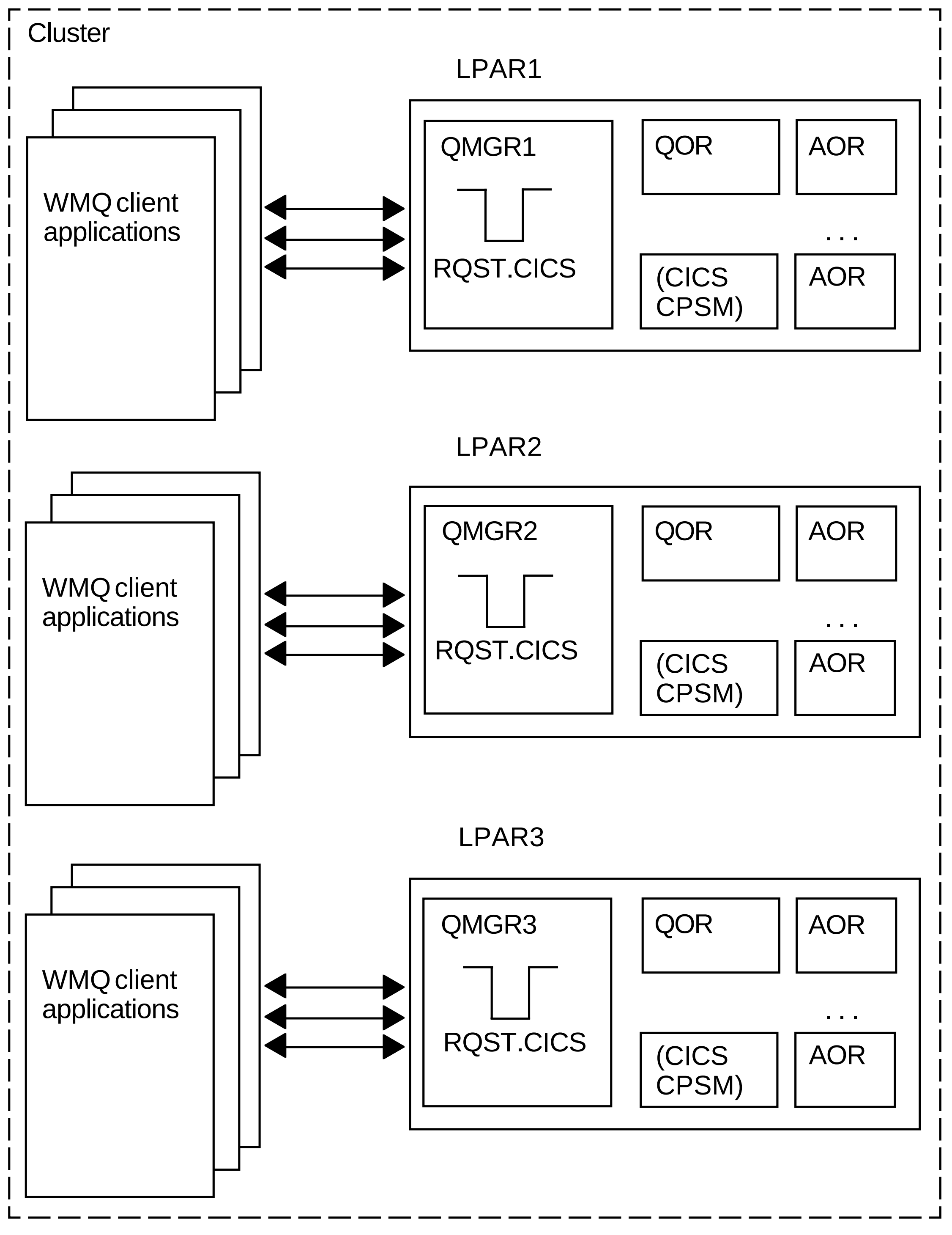

Figure 4: WMQ client applications within a WMQ cluster

- If a queue manager becomes unavailable any messages on its queues will not be processed until it restarts.
- The applications need extra logic to connect (and reconnect) to queue managers.

Two functions in WMQ V5.3 address these problems:

- Shared queues.
- Shared channels.

Queue-sharing groups

- Queue managers that can access the same set of shared queues are known as a queue-sharing group (QSG).
- QSG names can be up to four characters in length. The name must be unique, and different from any queue manager names.
- You put a message onto a shared queue on one queue manager and get the same message from the queue from a different queue manager.
- This facilitates a quick technique of interaction within a QSG that eliminates the need for active channels between queue managers.
- An application can connect to any of the queue managers within the QSG.
- Because all queue managers in the QSG can access all shared queues, an application does not depend on the availability of a specific queue manager.
- Any queue manager in the QSG can interact with the queue.

Shared Queues

- WMQ for z/OS V5.3 supports Shared Queues.
- WMQ for z/OS V5.3 Shared Queues supports persistent messages.

- A shared queue is a type of local queue whereby messages on that queue can be accessed by one or more queue managers that are in a Parallel Sysplex.
- Shared queues reside on a Coupling Facility (CF). All queue managers in a QSG can put and get messages from shared queues.
- The use of Shared Queues ensures better availability because all other queue managers in the QSG can continue processing the queue if one of the queue managers fails.
- The shared queue definition is stored in a DB2 shared database called the shared repository and because of this the queue only has to be defined once. It follows, therefore, that there are fewer definitions to generate.
- This is unlike the definition of a non-shared queue, which is stored on page set zero of the queue manager that owns the queue
- A shared queue cannot be defined if a queue with that name has already been defined on the page sets of the defining queue manager. Similarly, a local version of a queue cannot be defined on the queue manager page set zero if a shared queue with the same name already exists.

An example of how to use shared queues is illustrated in Figure 5.

This spreads the load among CICS members and provides transparent recovery in the event of a queue manager or CICS member outage.

Shared channels

WMQ for z/OS exploits Dynamic Domain Name Server (DDNS) for channel connections.

In Figure 5 clients can dynamically connect to any available queue manager in the QSG. The z/OS Workload Manager (WLM) spreads the connections evenly across queue managers.

This feature greatly simplifies the recovery logic in client applications

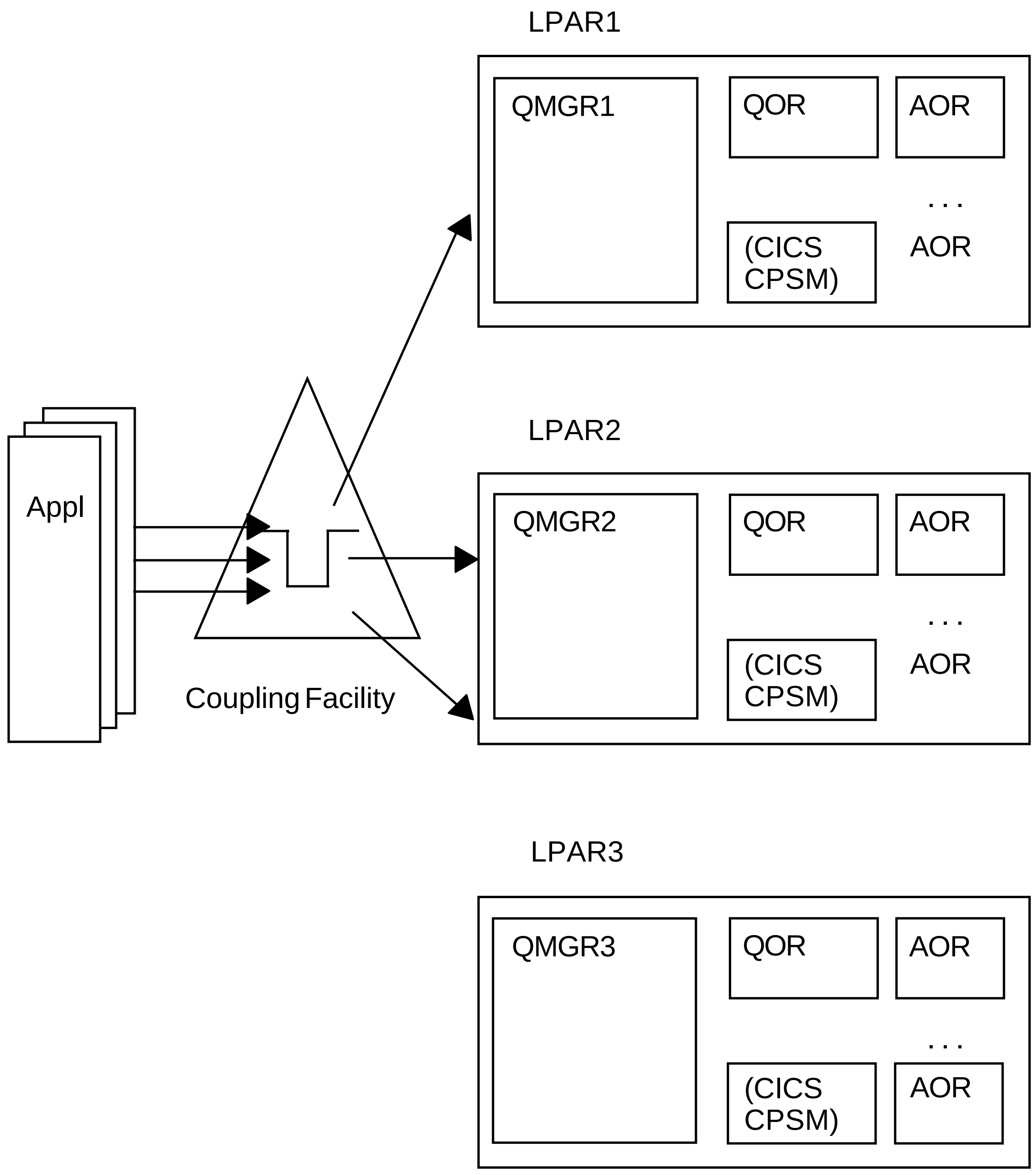

Figure 5: How to use shared queues

(because recovery consists simply of reconnecting to the same QSG).

Saida Davies
IBM (UK)

Using the XML Transform node in WMQI

XSLT (XML Stylesheet Language Transformation) is an XML language that transforms XML documents from one format to another. SupportPac IA0G provides a node that lets you use XSLT instead of compute node ESQL for simple reformatting.

Why would you want another reformatting tool when you have compute nodes? Here are some considerations:

- XSLT is a standardized and well-documented language. You can find support for it both inside and outside of the integrator community. ESQL on the other hand is a specialized language written for a single product. It's very difficult finding a book on ESQL at even the biggest and best bookstores.
- XSLT changes do not require a redeploy, just a file copy.
- You can reuse the same XSLT document inside and outside of WMQI.
- ESQL is the better language to use if you need to update database tables, fetch additional information through table look-ups, or do any kind of extensive computation.
- XSLT is best suited to taking data from one XML format and putting it into another. It can do some simple string functions but it is not well-suited to mathematical operations such as adding, rounding, or incrementing.

The XMLTransform node is available as support pack IA0G, and needs to be installed on both the WMQI server and on the developer workstations.

WMQI uses the Xerces XML parser from Apache and the XSL Transform node uses Apache's Xalan XSLT processor. If you are used to another parser, such as Microsoft's, be careful not to use proprietary parser extensions. Also, do not consider your XSLT to be fully tested until you have tested it against Xalan. A Web search should help you locate testing tools that use Xalan.

PITFALLS TO WATCH FOR

- *XML versus Blob*. While it may seem counter-intuitive, the XSL Transform node requires the message domain to be Blob, not XML. You can use the ResetContentDescriptor node to change message formats if you need to treat it as XML before or after the transformation.

- *XML encoding*. In your XSLT stylesheet suppress the XML declaration's ENCODING attribute. If you do not, you risk getting encoding conflicts when the message reaches your application server or workstation client. To draw you a picture, an attribute value such as *encoding="utf-8"* may prevent your XML from parsing if the message is sent to another server platform.

```
<?xml version="1.Ø" standalone="no" encoding="utf-8"  ?>
```

- *XML Declaration*. The XMLTransform node expects your XML to start with an XML declaration. If you do not include it, the XML will still transform but your event log will fill up with messages such as:

```
 (BrokerName.ExecutionGroupName) Java plug-in node error:
[com.ibm.xsl.mqsi.XMLTransformResources:
Error_Transformation_Engine]
 Error: The following error was received from the transformation
 engine: XML_PI_PARSING_FAILED.
 Error message generated by user Java plug-in node.
 Contact the node provider for further details.
```

- *Wire the Failure Terminal*. If you don't, it won't throw exceptions when the transform fails.

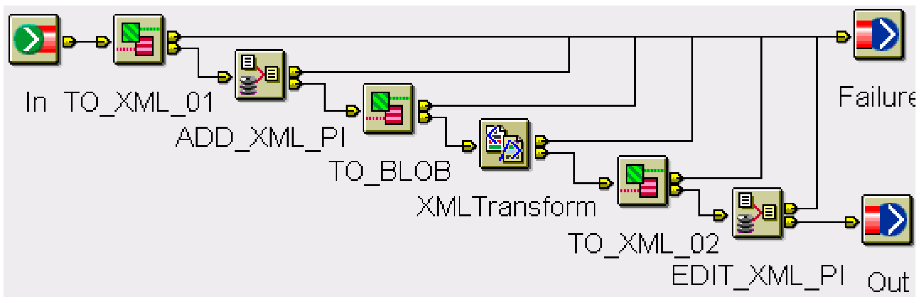

Figure 1: XSLT subflow

Node name	Node type
IN	**Input Terminal**
TO_XML_01	**ResetContentDescriptor**
	Ensures that the incoming format is XML so that compute node ADD_XML_PI will work. Message Domain = XML Reset Message Domain = True
ADD_XML_PI	**Compute**

	Adds an XML declaration to the message to avoid unnecessary error messages in the event log.
	`SET OutputRoot = InputRoot;`
	`SET "OutputRoot"."XML".(XML.XmlDecl).(XML.Version)='1.0';`

TO_BLOB	**ResetContentDescriptor**
	Changes the message format to blob so the XMLTransform node can use it. Message Domain = Blob Reset Message Domain = True
XMLTransform	**XMLTransform**
	This node points to an XSLT stylesheet file on the Integrator server. Promote the following properties so you can set them from the subflow level: Stylesheet Name Stylesheet Directory
TO_XML_02	**ResetContentDescriptor**
	Resets the message format back to XML. Message Domain = XML Reset Message Domain = True
EDIT_XML_PI	**Compute**

	Removes the encoding attribute from the message's XML declaration.
	`SET OutputRoot = InputRoot;`
	`SET "OutputRoot"."XML".(XML.XmlDecl).(XML.Version)='1.0';`

Failure	**Output terminal**
Out	**Output terminal**

Table 1: Subflow settings

SUGGESTED IMPLEMENTATION FOR XSLTRANSFORM

You can maximize the benefits and minimize the liabilities of the XMLTransform node by putting it into a subflow containing additional functionality. Figure 1 shows one way to do it. Configure the nodes according to Table 1.

CONCLUSION

XSLT may, in some cases, make more sense as a message transformation solution than ESQL. While the XMLTransform node may not be perfect, you can overcome the glitches by wrapping it in a subflow.

Mills Perry
ZyQuest (USA)

MQ news

Candle Corporation has announced the expansion of its offerings for Linux environments by providing new Linux support for WebSphere MQ management, as well as 64-bit platform support, across its Linux management solution portfolio.

Candle claims that its solution for WebSphere MQ, PathWAI Monitor for WebSphere MQ, delivers a high level of control and availability across the WebSphere MQ application infrastructure.

PathWAI Monitor for WebSphere MQ for Linux will be available in October 2003.

For more information contact:
Candle, 100 N Sepulveda Blvd, El Segundo, CA, 90245, USA.
Tel: +1 310 535 3600.
Fax: +1 310 727 4287.
Web: http://www.candle.com

Candle, 1 Archipelago, Lyon Way, Frimley, Camberley, Surrey, GU16 7ER, UK.
Tel: +44 1276 414 700.
Fax: +44 1276 414 777.

* * *

PolarLake recently announced the launch of its PolarLake Messaging Integrator product, which is claimed to enable XML documents to be received, transformed, and sent over tried and trusted messaging systems, without sacrificing the scalability, performance, and reliability required for enterprise deployment.

The company claims that the Integrator is designed to be an attractive option in terms of implementation cost and development time for any application integration project; particularly those using WebSphere MQ, TIBCO ActiveEnterprise, or any JMS-based messaging infrastructure.

Other features include support for content-based and publish/subscribe-based routing, message transformation and enrichment, validation, exception handling, transactionality, and activity monitoring.

For more information contact:
PolarLake USA, 39th Floor, 245 Park Avenue, New York, NY 10167, USA.
Tel: +1 212 672 1773.
Fax: +1 212 792 4001.
Web: http://www.polarlake.com

PolarLake (UK), 1 Liverpool Street, London EC2M 7QD, UK.
Tel: +44 20 7956 2090.
Fax: +44 20 7956 2001.

* * *