66

MQ update

December 2004

In this issue

A itc PUBLICATION

Published by

Xephon Inc
PO Box 550547
Dallas, Texas 75355
USA

Phone: 214-340-5690
Fax: 214-341-7081

Editor

Trevor Eddolls
E-mail: trevore@xephon.com

Publisher

Bob Thomas
E-mail: info@xephon.com

Disclaimer

Readers are cautioned that, although the information in this journal is presented in good faith, neither Xephon nor the organizations or individuals that supplied information in this journal give any warranty or make any representations as to the accuracy of the material it contains. Neither Xephon nor the contributing organizations or individuals accept any liability of any kind howsoever arising out of the use of such material. Readers should satisfy themselves as to the correctness and relevance to their circumstances of all advice, information, code, JCL, scripts, and other contents of this journal before making any use of it.

Subscriptions and back-issues

A year's subscription to *MQ Update*, comprising twelve monthly issues, costs $380.00 in the USA and Canada; £255.00 in the UK; £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 2000 issue, are available separately to subscribers for $33.75 (£22.50) each including postage.

Contributions

When Xephon is given copyright, articles published in *MQ Update* are paid for at the rate of $160 (£100 outside North America) per 1000 words and $80 (£50) per 100 lines of code for the first 200 lines of original material. The remaining code is paid for at the rate of $32 (£20) 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.

Printed in England.

WMQI Version 2 and WBIMB differences

This article deals mainly with the cosmetic differences between the two versions of IBM's strategic message broker system. They have become known as WMQI Version 2 and WMQI Version 5, although Version 5 should now be referred to as the WebSphere Business Integration Message Broker (WBIMB).

To the experienced developer and administrator of WMQI Version 2, the main differences can be seen in the development and administration interface. The old control centre has been replaced by a completely new open source Eclipse-based toolkit. For anyone who has already become familiar with Eclipse, such as Java developers, the transition may not be too radical; however, the introduction of Eclipse as the development interface introduces a whole new set of working practices. Here, I have highlighted some of the main differences between the two versions rather than given a comprehensive overview of the product. The reader must be familiar with WMQI Version 2 in order to appreciate the content, and anyone wishing either to learn WBIMB or to migrate their skill set should attend an appropriate course such as IBM's MQ66.

ECLIPSE

The use of Eclipse as the toolkit lays the foundation for a superior method of development and a more comprehensive means of source management. All resources under development are now held in the local file system. Eclipse manages resources around a project structure, and most resources, such as message flows, must belong to a project. This can lead to problems not experienced in WMQI Version 2 as will be seen later.

Eclipse organizes resource control under different perspectives. For example there is a development perspective for message flow creation and ESQL editing; an administration perspective for message flow deployment; and a debug

perspective for problem diagnosis. Within each perspective there are different windows and tab settings so that all functions available under the WMQI Version 2 tabs can be found somewhere within the appropriate perspective. Finding them can, at first, seem a little tricky, so perseverance and practice are essential.

SOURCE MANAGEMENT

One of the main criticisms of WMQI Version 2 is the lack of any comprehensive source control system. Eclipse allows strategic source control facilities such as Rational Clear Case and Microsoft CSV to be plugged in. The workspace refers to source files in the same way as WMQI Version 2, but far more of the WBIMB resources are now held as source; amongst them ESQL and deployable BAR files (discussed later).

There is now no inherent source control in WBIMB, so without a plugged-in resource control system there is no necessity to perform checkout prior to resource modification and check-in prior to deployment. These functions now form part of the plugged-in resource control facility.

CONFIGURATION MANAGER CONNECTION

One significant difference with WBIMB is the ability to carry out development without the need to connect to a configuration manager. The WMQI Version 2 control centre insists that a configuration manager is available before it will start. Nominating a different configuration manager requires the cumbersome practice of starting the control centre, connecting (or failing to connect) to the configuration manager last used by the control centre, and then connecting to the intended configuration manager if it differs from the previous one. WBIMB requires no such prior connection since all resources are held in the file system. The configuration manager is needed only in the administration perspective to configure broker execution groups and to perform deployments. Any number of configuration managers may be included in the

administration perspective and they may all be connected at the same time provided the appropriate WebSphere MQ connectivity is established.

COMPREHENSIVE ESQL EDITOR

Eclipse contains all those desirable features that should be included in a comprehensive editor such as colour schemes, search and replace, and content assist (context-based prediction). The WMQI Version 2 control centre contains none of these facilities and anyone working with large amounts of ESQL will be all too familiar with the frustration caused by its lack of finesse.

ESQL FILES

The ESQL relating to compute nodes, filter nodes, or any of the database nodes is no longer held as a property of the particular node, but is now held collectively as an ESQL file relating to the entire message flow. No matter how many compute nodes, filter nodes, or database nodes, there is only one ESQL file for the flow. The ESQL is separated by module delimiters, and the module name is referred to in the node properties. For large ESQL files, module selection is facilitated either by direct access from the node properties or by selecting the module from a list held in one of the development perspective windows.

WBIMB now allows the facility to share common ESQL. The project properties allow any other project to be nominated as a potential library of sharable modules. This allows for a more robust organization of source and a more conventional approach to application development than the linear method necessitated by the primitive nature of the WMQI Version 2 ESQL editor.

BAR FILES

In WMQI Version 2, message flows are dragged into an

appropriate execution group as an administration task under the assignments tabs. A deploy operation is then performed to ensure that these message flows are made available to the run-time broker. It is possible to nominate non-deployable message flows, which will cause the deploy operation to fail.

In WBIMB, a file known as a broker archive or BAR file is used to hold a selection of compiled deployable message flows, chosen from a list of acceptable candidates. Non-deployable flows such as sub-flows will not be included in this list and hence cannot be selected.

In WBIMB there is a downside caused by the use of Eclipse. It is not possible to deploy a message flow if that flow belongs to a project having errors, even though the errors are unrelated to the flow in question. This can be frustrating and is because of Eclipse's organization of resources. If a source management facility is in place, project rebuild and refresh may be needed to remove another developer's errors from the workspace, although even this can be unreliable and may prevent testing until the unrelated error is resolved.

Once a BAR file is created and is contained within the local or managed file system, it can be propagated to any other file system and hence be deployed to an appropriate broker.

THE MESSAGE REPOSITORY MANAGER (MRM)

There is no longer any requirement for a message repository database. Like all other resources, MRM definitions are held in the local file system.

Perhaps the most significant difference between the two versions, and one that affects the run-time broker, is the nomination of message sets. In WMQI Version 2, under the assignments tab, the message set is dragged into the broker outside of any execution group and hence belongs to the broker itself. In WBIMB the message set is included in the new BAR file, along with any message flows. The entire BAR file is then deployed to an execution group. This means that the

message set now forms part of an execution group rather than the broker, therefore it is possible to have different versions of the message set being used by different message flows if those message flows are in different execution groups. Care must be taken to redeploy to every execution group that requires the new message set and this can lead to an administration overhead.

Support for XML schema import and generation has been included.

RAPID APPLICATION DEVELOPMENT (RAD)

In WMQI Version 2, anyone wishing to perform a deploy operation needs appropriate authorization. This presents a potential security exposure in that developers who needed authority to deploy to a test broker may automatically have the right to deploy to a production broker. This exposure can be prevented in Version 2 either by having separate broker domains, with all the appropriate facilities to migrate from one broker to another, or by restricting the authority of the developer to disallow deploy operations. Some other means of requesting a deploy must then be invoked, such as an operator request by telephone, which inhibits a smooth development process.

In WBIMB, the developer is given the opportunity to select message flows that may be transferred and run on the broker without the need to access the administration perspective and perform a specific deploy. Developer authority is sufficient to allow this activity and gives a restricted administration capability for test purposes. It makes modification and re-testing of message flows much more streamlined. For the purpose of rapid application development, control files are created the first time the facility is invoked. These control files may be modified, as the developer's needs change.

REMOVAL OF THE COMMAND ASSISTANCE

WMQI Version 2.1 provides assistance from the start/programs menu with the creation of components such as the configuration

manager, broker, or username server in a Windows environment. This assistance has been removed from the start panel, and component creation must be carried out using the command line functions.

DEBUGGING

The WMQI Version 2 control centre debugger is both cumbersome and unreliable, and can lead to potential broker problems. WBIMB has a vastly superior debugging capability using the agent controller and includes the ability to single step through ESQL source. If message flows experience problems, it is possible simply to disconnect from the debugger in order to 'clean up'. Die-hard developers who rely on the WMQI Version 2 usertrace still have the option available under WBIMB, but the superior debugging capability may make this a necessity only in the most extenuating circumstances.

SECURITY

WBIMB has introduced many new security features.

ADDITIONAL MESSAGE PROTOCOLS

Support now exists within the broker to start a flow instance from input received via HTTP and WebSphere MQ Everyplace.

CONCLUSION

There are many other differences between the WMQI Version 2 and WBIMB as well as those that have been highlighted here. Only practice with the new control centre and re-learning will give a sufficient insight into all of these new features and differences. The experienced WMQI Version 2 developer will find the new WBIMB toolkit extremely versatile.

Ken Marshall
Middleware Consultant
MQSolutions (UK) Limited

WMQ and MDBs

It all started with the Java Message Service (JMS). This is the Java API that enables loosely-coupled Java clients to make asynchronous interactions with messaging systems such as WebSphere MQ.

JMS is basically a low-level API enabling applications to connect to messaging systems. To make coding for message consumption simpler, J2EE 1.3 introduced a new kind of enterprise bean, based on the JMS API, called the Message-Driven Bean (MDB). MDBs are part of the EJB 2.0 specification, and they act as JMS message listeners in the application server container.

WebSphere Studio Application Developer 5.0 supports the J2EE 1.3 specification, which includes EJB 2.0. With EJB 2.0, users can create MDBs, which let EJB containers receive messages asynchronously. Client components can send messages to an EJB container without having to wait for a reply. The ability to send messages to an EJB container rather than a Java object makes the application more efficient. An EJB container can send out distributed transactions when it receives a message. WebSphere MQ can then push updates into an EJB system and participate in a distributed transaction with other resources, for example a database. Without MDBs or EJBs it was necessary to code logic to browse a queue and then start a transaction.

Let's first look at JMS – WebSphere MQ provides messaging services based on the JMS API.

Implementing JMS messaging requires a JMS provider (ie WMQ) and JMS clients that use the messaging system. The clients can be message producers or message receivers. The producers identify the receivers by a destination object. So, the message producer uses a JMS provider to create a message containing a destination address, and the message

is sent to a message receiver having that particular destination address – simple!

To make things slightly more complex, there are two ways for JMS to send a message to a receiver. They are:

- Publish-and-subscribe – a single message producer sends a message to many receivers. This uses a channel called 'topic' to publish the message, which is broadcast to all receivers subscribed to the topic.
- Point-To-Point (PTP) messaging allows clients to send and receive messages synchronously and asynchronously through channels called queues. Although multiple receivers could be attached to a queue, the message is delivered to only the one receiver it was intended for.

JMS supports five message types:

- BytesMessage – allows users to send a message as a stream of bytes. This can be used for wrapping existing message formats.
- MapMessage – allows users to store data as name/value pairs. Users can manipulate the MapMessage via msg.setString(key, value) and msg.getString(key).
- ObjectMessage – allows serializable Java objects to be stored as part of a message. Users manipulate the ObjectMessage via msg.setObject(Object o) and msg.getObject().
- StreamMessage – allows users to send a message as a stream of primitives.
- TextMessage – holds a simple string message. Users can manipulate the TextMessage via msg.setText("foo") and msg.getText().

Now let's see where MDBs fit in. Message driven beans are enterprise beans that act as message receivers. They are different from other JMS clients that act as message receivers in that they can look after security, concurrency, transaction,

etc. The standard message receiver is a Java client program that receives and processes the message. It waits for a message to be delivered by the message producer. The length of time it has to wait can tie up system resources. To overcome this problem, what's needed is a client that will poll frequently for a message and if no message is received, timeout the request. This is what MDBs do.

MDBs are different from session beans and the entity beans.

Like stateless session beans:

- MDBs do not maintain conversational state between requests. They may have instance variables throughout their life cycle, but because of the pooling of bean instances by the EJB container, the bean instances that consume the messages may be different between requests. That is why the conversational state may not be stored properly.
- The EJB container maintains many bean instances of the same type in a pool. This enables concurrent message consumption and processing when several messages are delivered at the same time. This allows MDBs to deliver better performance and scalability.

Unlike session and entity beans:

- MDBs do not have remote or home interfaces, nor do they have a component interface. An MDB is a listener – it is not a remote process call component.
- MDBs do not expose any business methods that can be invoked by clients, such as a servlet, EJB, or Java application.

Like session and entity beans, MDBs must use the JMS API manually to send messages when they need to act as a message producer. However, sending messages from MDBs is not recommended – MDBs should delegate this task to the business logic layer.

MDBs have the flexibility of a JMS client and the enterprise

features of a stateless session bean. MDBs implement the MessageDrivenBean interface. This extends the javax.ejb.Enterprisebean interface, ensuring that the MDB can be controlled by the ejb container – just like any other enterprise bean. MDBs also implement the javax.jms.MessageListener interface. This helps them to adhere to the JMS messaging API. It's worth noting that the EJB2.0 specification allows MDBs to support only JMS-based messaging systems. The EJB2.1 specification allows them to support other kinds of messaging system.

An MDB has two states:

- Does not exist
- Method-ready.

To begin with, the bean exists in the 'does not exist' state. When the container starts up, it may load a few instances into memory so that they can be in the 'ready' state to consume messages. When a bean moves from the 'does not exist' state to the 'method-ready' state, the container first invokes the Class.newInstance() method to instantiate (create a new instance of) the bean. It then invokes the setMessageDrivenContext() method to set the reference to the ejbContext. This provides a reference to the MessageDrivenContext Interface, which is in turn used to handle transaction and security issues in the MDB. Finally the ejbCreate() method is invoked by the container. MDBs have only one ejbCreate() method and they are called only once in the lifetime of an MDB.

When the bean is in the 'method-ready' state, it is ready to receive messages. The container allocates a bean to receive a message when it arrives in the container. When a bean is servicing one message it cannot service another, so subsequent messages are delegated to other instances that exist in the pool. Once the bean has serviced a message it is again returned to 'method-ready' and is ready to process a new message. If the container cannot find a bean in the pool

to service a message it converts another bean from the ‘does not exist’ state to the ‘method-ready’ state.

Bean instances leave the ‘method-ready’ pool for the ‘does not exist’ state when the server no longer needs them. This occurs when the server decides to reduce the total size of the ‘method-ready’ pool by removing one or more instances from memory. The process begins by invoking the ejbRemove() method on the instance. At this time, the bean instance should perform any clean-up operations, such as closing open resources. The ejbRemove() method is invoked only once in the life-cycle of an MDB instance – when it is about to convert to the ‘does not exist’ state. During the ejbRemove() method, the MessageDrivenContext and access to the JNDI ENC are still available to the bean instance. Following the execution of the ejbRemove() method, the bean is dereferenced and eventually goes when the garbage is collected.

WebSphere uses an extension called a listener port, which allows users to bind an MDB bean to a JMS queue or topic. The two most important attributes of a port are:

- The JNDI name of the JMS queue connection factory.
- The JNDI name of the JMS queue or topic.

A listener port is configured at the application server level. Listener ports are managed by the Message Listener Service of an application server. Ports can be created in WSAD (WebSphere Studio Application Developer) or using the Web-based administrative console. The listener port can be bound to an MDB using the EJB deployment descriptor editor. The binding information is stored in the ibm-ejb-jar-bnd.xmi file.

In order to set up a development environment, it’s necessary to install WebSphere Studio Application Developer without the embedded messaging support. Then install WebSphere MQ separately. WebSphere MQ requires a custom installation in order to select the Java Messaging component.

With this arrangement, it’s possible to run a distributed

transaction from an MDB between WebSphere MQ and DB2, for example.

MDBs reduce the amount of coding necessary to achieve asynchronous message processing that is fast and efficient. MDBs work with WebSphere MQ (and other messaging software) and can help integrate it with other mainframe-based applications as well as having many uses in distributed environments.

Nick Nourse
Independent Consultant (UK)

Removing a queue manager from a WMQ cluster

INTRODUCTION

WebSphere MQ (WMQ) has included a feature called clustering since Version 5.1 of the product. This feature simplifies the administration of WMQ because it allows queues and channels to be defined automatically. Other benefits are some form of fail-over and workload mechanism. It is quite easy to join a queue manager to a cluster, but a lot more difficult to remove it smoothly.

Since Version 5.2 it has been possible to remove a queue manager forcibly out of a cluster, but only if you have access to one of the full repositories. This article describes a script that removes a queue manager from a cluster without the need to have access to a full repository queue manager, or enlist the help of someone else who has access.

THE CLUSTER REMOVAL SCRIPT

Aim of the cluster removal script

The cluster removal script allows the owner of a queue

manager, which is not a full repository, to remove it from a cluster. Because several steps are necessary to remove queues, a channel, and the queue manager, it is better to run these commands in a script, in order to perform the steps in the right order without missing anything.

How the cluster removal script works

The script simply performs the steps described in the IBM document *Queue Manager Clusters*. The required command line options are the name of the queue manager and the cluster to remove from. The script checks for cluster, channel, and name lists, which contain the name of the specified WMQ cluster.

First the script suspends the queue manager from the cluster. Then it alters any queue that is a member of the specified WMQ cluster name. If cluster membership is set by using a name list – eg when a queue is a member of more than one cluster – the name list will be replaced by another one that does not contain the specified cluster name. The next step is to alter the cluster receiver channel in the same way. In fact, this step removes the queue manager finally from the WMQ cluster. The last step is to alter the cluster sender channel in the same way, to interrupt the connection to the full repositories, and to clean up the local (partial) repository.

Using the cluster removal script

If you want to recreate the queue manager and its cluster membership at some point, be sure you have a configuration file (eg created with saveqmgr, which is available as SupportPac MS03 on the home page of IBM). To remove the queue manager from a WMQ cluster, run the script as follows:

```
ClusterRM queuemanager clustername
```

The script now checks that the user is a member of the group *mqm* and that the queue manager is a member of the cluster. Last but not least, the script asks the user to start the cluster removal. If the user answers ‘Y’, the removal will start.

SAMPLE CONFIGURATION

Description of the sample configuration

To test the script and see how it works, I created a sample configuration. This configuration consists of two WMQ queue manager clusters, named CL_1 and CL_2. The queue managers CLREPO1 (for CL_1) and CLREPO2 (for CL_2) act as full repository queue managers for these clusters. Another queue manager, named CLQMGR, is set up as a member of both clusters. Several queues are defined on CLQMGR as members of one or both clusters. The repository queue managers are not connected directly.

Setting up the sample configuration

Create three queue managers with the names CLREPO1, CLREPO2, and CLQMGR. Configure them with the MQSC files below, using the following command (replace qmgr by the names above):

```
runmqsc qmgr < qmgr.mqsc
```

Now start the channel listeners by issuing the following commands (replace *port* in this sample by 1414, 1415, and 1416):

```
nohup runmqlsr -m qmgr -t tcp -p port &
```

The queues are defined as members of one cluster (queues TEST.CL*) or both (via name list, queues TEST.NL.*).

MQSC definition files for the sample configuration

CLREPO1:

```
ALTER QMGR +
       REPOS('CL_1') +
       FORCE

DEFINE CHANNEL ('TO.CLREPO1') CHLTYPE(CLUSRCVR) +
       DISCINT(60) +
       CLUSTER('CL_1') +
       CONNAME('127.0.0.1(1414)') +
       REPLACE
```

CLREPO2:

```
ALTER  QMGR +
       REPOS('CL_2') +
       FORCE

DEFINE CHANNEL ('TO.CLREPO2') CHLTYPE(CLUSRCVR) +
       DISCINT(60) +
       CLUSTER('CL_2') +
       CONNAME('127.0.0.1(1415)') +
       REPLACE
```

CLQMGR:

```
DEFINE QALIAS('TEST.CL1.QA') +
       TARGQ('TEST.1.QL') +
       CLUSTER('CL_1') +
       REPLACE

DEFINE QALIAS('TEST.CL2.QA') +
       TARGQ('TEST.2.QL') +
       CLUSTER('CL_2') +
       REPLACE

DEFINE QALIAS('TEST.NL.QA') +
       TARGQ('TEST.QL') +
       CLUSNL('CL_LIST_1') +
       REPLACE

DEFINE QLOCAL('TEST.1.QL') +
       REPLACE

DEFINE QLOCAL('TEST.2.QL') +
       REPLACE

DEFINE QLOCAL('TEST.QL') +
       REPLACE

DEFINE QLOCAL('TEST.CL1.QL') +
       CLUSTER('CL_1') +
       REPLACE

DEFINE QLOCAL('TEST.CL2.QL') +
       CLUSTER('CL_2') +
       REPLACE

DEFINE QLOCAL('TEST.NL.QL') +
       CLUSNL('CL_LIST_1') +
       REPLACE

DEFINE CHANNEL ('TO.CLQMGR') CHLTYPE(CLUSRCVR) +
```

```
        DISCINT(60) +
        CLUSNL('CL_LIST_1') +
        CONNAME('127.0.0.1(1423)') +
        REPLACE

DEFINE CHANNEL ('TO.CLREPO1') CHLTYPE(CLUSSDR) +
        DISCINT(60) +
        CLUSTER('CL_1') +
        CONNAME('127.0.0.1(1421)') +
        REPLACE

DEFINE CHANNEL ('TO.CLREPO2') CHLTYPE(CLUSSDR) +
        DISCINT(60) +
        CLUSTER('CL_2') +
        CONNAME('127.0.0.1(1422)') +
        REPLACE

DEFINE NAMELIST ('CL_LIST_1') REPLACE +
        NAMES('CL_1','CL_2')

DEFINE NAMELIST ('CL_LIST_2') REPLACE +
        NAMES('CL_1','CL_2','CL_3')

DEFINE NAMELIST ('CL_LIST_3') REPLACE +
        NAMES('CL_1')

DEFINE NAMELIST ('CL_LIST_4') REPLACE +
        NAMES('CL_2')
```

Removing the sample configuration

To remove the sample, stop all queue managers and delete them using the following commands:

```
endmqm -i qmgr

dltmqm qmgr
```

DESCRIPTION OF THE CODE

The code consists of two main parts.

Part 1

Part 1 contains some functions to prepare the cluster removal. Nothing will really happen in this part of the script. The functions first check whether the queue manager is running

and a member of the specified cluster (functions *check_qmgr_running* and *check_cluster_member*). The queue manager is a cluster member, when a cluster sender channel has been found, which contains either the cluster name in the attribute *CLUSTER* or a name list in the attribute *CLUSNL*, which itself contains the cluster name in the attribute *NAMES*. Name lists, which contain the cluster names, are found in the function *find_cluster_namelists*.

In general, a manually-defined cluster sender should not belong to more than one WMQ cluster because otherwise you would have several queue managers with the same name. It is recommended by IBM to have unique queue manager names. Nevertheless it is possible to define a cluster sender using a cluster name list, and the script ClusterRM takes care of it.

Part 2

In the second part, the cluster removal will be executed. The user must accept this removal by answering the question 'Start the cluster removal now'? with 'Y'. First the queue manager is suspended from the specified WMQ cluster to disable any queues defined on this queue manager as usable by the cluster. Now each name list that contains the cluster name will be copied to a temporary name list without this entry (function *create_temp_namelists*). Then the *remove_queues_from_cluster* function removes any queue from the cluster by altering the attribute *CLUSTER* (set it to a space character) or by replacing the original name list in the attribute *CLUSNL* by the temporary one created before.

The steps described above remove the queues from the WMQ cluster. Now it is time to remove the queue manager itself from the cluster. This goal is reached by altering the manually-defined cluster receiver in the same way as described for the queues (altering the attributes *CLUSTER* or *CLUSNL)*. Now the queue manager is finally removed from the cluster, although you still have a cluster sender channel to the full repository

defined (and running). This channel (there may be more than one, if you have multiple full repositories) must still be defined for the cluster because all other cluster modifications, described above, have to be transmitted to the full repositories using the cluster sender channel.

Now ClusterRM cleans up the queue manager by altering this cluster sender in a similar way to the previous steps, and finally drops the cluster information by issuing the command:

```
REFRESH CLUSTER(clustername) REPOS(YES)
```

LISTING OF THE CLUSTER REMOVAL SCRIPT

```
#!/bin/ksh
#################################################################
#   TITLE
#
#   Script to remove a queue manager from a
#   WebSphere MQ cluster.
#################################################################
#   DESCRIPTION
#
#   This script automates the removing of a queue
#   manager from a WebSphere MQ cluster. The script
#   follows the steps described in the IBM document
#   "Queue Manager Clusters".
#
#   Hubert Kleinmanns
#   Senior Consultant
#################################################################

#################################################################
# Part I:
# Prepare cluster removal.
#################################################################
# Initialize the script environment.
#
function initialise
{
   $DEBUG

   # Set the program name and path.
   PROG_NAME=$(basename $1)
   DIR_NAME=$(cd $(dirname $1); pwd)
   # This script must be executed as group mqm.
   g='groups | grep -w mqm'
```

```
   if [ "$g" = "" ]
   then
      echo "\n*** ERROR: You need to be a member of group \"mqm\" to run
this program!\n"

      return 1
   fi
   # Store the actual date and time.
   DATE='date "+%Y%m%d%H%M%S"'
   # Initialize a flag.
   QMGR_EXISTS="n"
   # Setting the operating system.
   OS='uname -s'
   # Define a separator line.
    SEP_LINE="----------------------------------------------------------
--------------"

   # Check the command line parameters.
   if [ $# -eq 3 ]
   then
      QMGR_NAME=$2
      CLUS_NAME=$3
   else
      echo "usage: $PROG_NAME queue_manager mq_cluster"
      return 1
   fi

   # Set retry parameters for channel reccle.
   WAIT_SEC=5
   MAX_RETRY=1Ø
   # Define the name of the log.
   LOG_FILE="ClusterRM.log"
   # Create a log file and write a time stamp.
   echo
"*****************************************************************"  >
$LOG_FILE
   echo "*" >> $LOG_FILE
   echo "*   Cluster removal started at `date`" >> $LOG_FILE
   echo "*" >> $LOG_FILE
   echo
"*****************************************************************"  >>
$LOG_FILE
   echo "" >> $LOG_FILE
   return Ø
}
#
# Remove queue manager from WebSphere MQ cluster
#
function check_qmgr_running
{
```

```
   $DEBUG
   echo "   Checking queue manager $QMGR_NAME."
   # Run a dummy command.
   chk='echo "dis qmgr" | runmqsc $QMGR_NAME | grep QMGR'
   # Output must be not-empty.
   if [ "X$chk" = "X" ]
   then
      echo "   ERROR: Queue manager $QMGR_NAME is not running, start it
first!"
      return 1
   fi
   echo "   Queue manager $QMGR_NAME is running."
   return Ø
}
#
# Find the name lists, which contain the cluster name,
# from which the queue manager shall be removed.
#
function find_cluster_namelists
{
   $DEBUG
   cnt=Ø
   CL_NAMELIST=""
   cl_nl=""
   cl_nl_tmp=""
   echo "   Look for name lists which contain the cluster name
$CLUS_NAME."

   # Display all existing name lists (except system name lists).
   namelist='echo "DIS NL(*)" | runmqsc $QMGR_NAME | grep NAMELIST |
grep -v SYSTEM | sed -e "s/^.*(\(.*\)).*$/\1/"'

   # Loop and extract name lists, which contain the cluster name.
   for nl in $namelist
   do
      # Display the value of the NAMES attribute and look for the
cluster name.
      chk='echo "DIS NL($nl) NAMES" | runmqsc -e $QMGR_NAME | grep -v
"^[a-zA-ZØ-9].*" | sed -e "s/.*NAMES(//" | sed -e "s/).*$//" | grep -w
$CLUS_NAME'

      # If name list contains cluster name...
      if [ "$chk" != "" ]
      then
         # Set up a name for a temporary name list.
         tmp_nl="TEMP_${DATE}_${cnt}"
         # Store the name of the name list.
         cl_nl="$cl_nl$nl "
         cl_nl_tmp="$cl_nl_tmp$tmp_nl "
         # Increase the counter.
```

```
         let cnt=$cnt+1
      fi
   done

   # Store the name lists in two arrays.
   set -A CL_NAMELIST $cl_nl
   set -A CL_NAMELIST_TEMP $cl_nl_tmp
   # Count the entries in the arrays.
   NL_COUNT='echo $cl_nl | wc -w'
}
#
# Check, whether a cluster sender channel exists (otherwise the
# queue manager cannot be a member of the cluster.
#
function check_cluster_member
{
   $DEBUG
   echo "   Check if \"$QMGR_NAME\" is a member of cluster
\"$CLUS_NAME\"."

   # Initialize the test value.
   tst_val=1
   # Find a cluster sender channel, which contains the cluster name...
   chk='echo "DIS CHL(*) CHLTYPE(CLUSSDR) CLUSTER" | runmqsc $QMGR_NAME
| grep -w $CLUS_NAME'

   # ...and set test value to OK if channel was found...
   [ "$chk" != "" ] && tst_val=Ø
   # ...otherwise look for name lists in the channel definition.
   idx=Ø
   while [ $idx -lt $NL_COUNT ]
   do
      # Get the name list name.
      nl=${CL_NAMELIST[$idx]}
      # Find cluster sender channel, which contain the name list...
      chk='echo "DIS CHL(*) CHLTYPE(CLUSSDR) CLUSNL" | runmqsc
$QMGR_NAME | grep -w $nl'

      # ...and set test value to OK if a channel was found.
      [ "$chk" != "" ] && tst_val=Ø
      # Increase the counter.
      let idx=$idx+1
   done
   if [ $tst_val -ne Ø ]
   then
      echo "   ERROR: Queue manager \"$QMGR_NAME\" is not a member of
cluster \"$CLUS_NAME\"!"

      return 1
   fi
```

```
   echo "   Queue manager \"$QMGR_NAME\" is a member of cluster
\"$CLUS_NAME\"."

   return Ø
}
#
# Prepare the cluster removal
#
function prepare_remove_cluster
{
   $DEBUG
   initialise $*
   ret=$?
   [ $ret -ne Ø ] && return $ret
   echo "$SEP_LINE"
   echo "Prepare the cluster removal."
   # Check whether the queue manager is active.
   check_qmgr_running
   ret=$?
   [ $ret -ne Ø ] && return $ret
   # Find name lists that contain the cluster name.
   find_cluster_namelists
   # Check whether the qmgr is a member of the cluster.
   check_cluster_member
   ret=$?
   [ $ret -ne Ø ] && return $ret
   return Ø
}
#####################################################################
# Part II:
#
# Execute cluster removal.
#####################################################################
# Function that checks whether all channels are down. If
# at least one channel is in state RUNNING or STOPPING,
# the return value is NOT OK. This function checks several
# times (defined by the first argument) and waits a while
# (defined by the second argument) before the next test.
#
function wait_channels_down
{
   $DEBUG
   # Store the test counter and the wait interval.
   max_cnt=$1
   wait_tm=$2
   shift 2
   cnt=1
   # Loop over the test counter.
   while [ $cnt -le $max_cnt ]
   do
```

```
      down="Y"
      # Test any channel.
      for chl in $*
      do
         chk='echo "DIS CHS($chl) STATUS" | runmqsc $QMGR_NAME | grep
STATUS | egrep 'RUNNING|STOPPING''

     # Set check value to NOT OK, if any channel is running or stopping.
         [ "$chk" != "" ] && down="N"
      done
      # Return OK when all channel are down.
      [ "$down" = "Y" ] && return Ø
      # Increase the counter.
      let cnt=$cnt+1
      echo "         Waiting until the channels are down."
      # Wait a while.
      sleep $wait_tm
   done
   # Return NOT OK when at least one channel is running or
   # stopping after the specified time-out.
   return 1
}
#
# This function stops a list of channels, waits a while, and checks
# whether all channels are stopped, and starts the channels again.
#
function recycle_channel
{
   $DEBUG
   chl_type=$1
   # Stop now the interval.
   for chl in $*
   do
      echo "STOP CHL($chl) MODE(FORCE)" | runmqsc $QMGR_NAME >>
$LOG_FILE 2>&1
   done

   # Wait until the channels are down.
   wait_channels_down $MAX_RETRY $WAIT_SEC $*
   ret=$?
   if [ $ret -ne Ø ]
   then
      echo "   ERROR: Unable to restart channel."
      return $ret
   fi
   # Start the channel again.
   for chl in $*
   do
      echo "START CHL($chl)" | runmqsc $QMGR_NAME >> $LOG_FILE 2>&1
   done
```

```
   return Ø
}
#
# Create temporary name lists without the cluster, where
# the queue manager has to be removed from.
#
function create_temp_namelists
{
   $DEBUG
   nl_empty=""
   echo "   Create temporary name lists without cluster $CLUS_NAME."

   # Loop over all name lists.
   idx=Ø
   while [ $idx -lt $NL_COUNT ]
   do
      sep=""
      nl=${CL_NAMELIST[$idx]}
      tmp_nl=${CL_NAMELIST_TEMP[$idx]}
      # Get the names of the name list. Ignore the cluster name,
      # where the queue manager shall be removed from.
      names='echo "DIS NL($nl) NAMES" | runmqsc -e $QMGR_NAME | grep -v
"^[a-zA-ZØ-9].*" | grep -v "^$" | sed -e "s/.*NAMES(//" | sed -e "s/
).*$//" | sed -e "s/ *,//" | grep -v -w $CLUS_NAME'

      new_names=""
      # Loop over the found names and store these in a new list.
      for name in $names
      do
         new_names="$new_names$sep$name"
         sep=","
      done
      # If the new list would be empty...
      if [ "$new_names" = "" ]
      then
         # ...mark the name list as 'empty', to be deleted later.
         empty="yes"
      else
         # ...otherwise define the temporary name list...
         echo "DEFINE NL($tmp_nl) NAMES($new_names)" | runmqsc
$QMGR_NAME > /dev/null 2>&1

         # ...and mark the name list as 'not empty'.
         empty="no"
      fi
      # Store the 'empty' flag.
      nl_empty="$nl_empty$empty "
      # Increase te counter.
      let idx=$idx+1
   done
```

```
   # Store the 'empty' flags in an array.
   set -A CL_NAMELIST_EMPTY $nl_empty
}
#
# Remove queues from WebSphere MQ cluster
#
function remove_queues_from_cluster
{
   $DEBUG
   echo "   Remove queues from cluster $CLUS_NAME."
   # Loop over all queue types.
   for type in QLOCAL QALIAS QREMOTE QMODEL
   do
      # Get the queues which have the cluster attribute set
      # to the cluster, to be removed from.
      q_list='echo "DIS $type(*) CLUSTER" | runmqsc $QMGR_NAME | grep -w
$CLUS_NAME | sed -e "s/^.*QUEUE(\(.*\)).*$/\1/"'

      # Alter now the queues by removing the cluster attribute.
      for queue in $q_list
      do
         echo "ALTER $type($queue) CLUSTER(' ')" | runmqsc $QMGR_NAME >>
$LOG_FILE 2>&1
      done
   done
   # Loop now over all name lists.
   idx=Ø
   while [ $idx -lt $NL_COUNT ]
   do
      nl=${CL_NAMELIST[$idx]}
      empty=${CL_NAMELIST_EMPTY[$idx]}
      # Loop over all queue types with this name list set.
      for type in QLOCAL QALIAS QREMOTE QMODEL
      do
         q_list='echo "DIS $type(*) CLUSNL" | runmqsc $QMGR_NAME | grep
-w $nl | sed -e "s/^.*QUEUE(\(.*\)).*$/\1/"'

         # For each queue...
         for queue in $q_list
         do
            # ...if the 'empty' flag is set...
            if [ "$empty" = "yes" ]
            then
               # ...remove the cluster name list attribute.
               echo "ALTER $type($queue) CLUSNL(' ')" | runmqsc
$QMGR_NAME >> $LOG_FILE 2>&1
            else
         # ...otherwise alter the attribute to the temporary name list.
               tmp_nl=${CL_NAMELIST_TEMP[$idx]}
```

```
               echo "ALTER $type($queue) CLUSNL($tmp_nl)" | runmqsc
$QMGR_NAME >> $LOG_FILE 2>&1
            fi
         done
      done
      # Increase the counter.
      let idx=$idx+1
   done
}
#
# Remove queue manager from WebSphere MQ cluster
#
function alter_cluster_receiver
{
   $DEBUG
   echo "   Alter cluster receiver channel."
   # Get a list of cluster receiver channel, which contain the cluster
name.
   rcvr_list='echo "DIS CHL(*) CHLTYPE(CLUSRCVR) CLUSTER" | runmqsc
$QMGR_NAME | grep -w $CLUS_NAME | awk '{print $1}' | sed -e "s/
^.*CHANNEL(\(.*\)).*$/\1/"'

   echo "      Alter the cluster attribute."

   # Remove now the cluster attribute from the channel.
   for rcvr in $rcvr_list
   do
      echo "ALTER CHL($rcvr) CHLTYPE(CLUSRCVR) CLUSTER(' ')" | runmqsc
$QMGR_NAME >> $LOG_FILE 2>&1
   done
   # Restart the channel.
   recycle_channel $rcvr_list
   ret=$?
   [ $ret -ne Ø ] && return $ret
   echo "      Alter the cluster name list attribute."
   # Loop over all name lists.
   idx=Ø
   while [ $idx -lt $NL_COUNT ]
   do
      nl=${CL_NAMELIST[$idx]}
      empty=${CL_NAMELIST_EMPTY[$idx]}
      # Get a list of cluster receiver channel, which contains
      # the cluster name list.
      rcvr_list='echo "DIS CHL(*) CHLTYPE(CLUSRCVR) CLUSNL" | runmqsc
$QMGR_NAME | grep -w $nl | awk '{print $1}' | sed -e "s/
^.*CHANNEL(\(.*\)).*$/\1/"'

      # Loop over all cluster receiver channel.
      for rcvr in $rcvr_list
      do
```

```
         # ...if the 'empty' flag is set...
         if [ "$empty" = "yes" ]
         then
            # ...remove the cluster name list attribute.
            echo "ALTER CHL($rcvr) CHLTYPE(CLUSRCVR) CLUSNL(' ')" |
runmqsc $QMGR_NAME >> $LOG_FILE 2>&1
         else
            # ...otherwise alter the attribute to the
            #       temporary name list.
            tmp_nl=${CL_NAMELIST_TEMP[$idx]}
            echo "ALTER CHL($rcvr) CHLTYPE(CLUSRCVR) CLUSNL($tmp_nl)" |
runmqsc $QMGR_NAME >> $LOG_FILE 2>&1
         fi
      done
      # Restart the channel to activate the modifications above.
      recycle_channel $rcvr_list
      ret=$?
      [ $ret -ne Ø ] && return $ret
      # Increase the counter.
      let idx=$idx+1
   done
}
#
# Remove queue manager from WebSphere MQ cluster
#
function alter_cluster_sender
{
   $DEBUG
   echo "   Alter cluster sender channel."
   # Get a list of cluster sender channel, which contain the cluster
name.
   sdr_list='echo "DIS CHL(*) CHLTYPE(CLUSSDR) CLUSTER" | runmqsc
$QMGR_NAME | grep -w $CLUS_NAME | awk '{print $1}' | sed -e "s/
^.*CHANNEL(\(.*\)).*$/\1/"'

   # Extend the list by displaying the cluster queue managers - in order
   # to get automatically defined cluster sender channel.
   sdr_list="$sdr_list 'echo \"DIS CLUSQMGR(*) CLUSTER($CLUS_NAME)\" |
runmqsc $QMGR_NAME | grep -w CHANNEL | sed -e 's/^.*CHANNEL(\(.*\)).*$/
\1/''"

   echo "      Alter the cluster attribute."
   # Remove the cluster attribute from the channel.
   for sdr in $sdr_list
   do
      echo "ALTER CHL($sdr) CHLTYPE(CLUSSDR) CLUSTER(' ')" | runmqsc
$QMGR_NAME >> $LOG_FILE 2>&1
   done
   # Restart the channel to activate the modifications above.
   recycle_channel $sdr_list
```

```
   ret=$?
   [ $ret -ne Ø ] && return $ret
   echo "      Alter the cluster name list attribute."
   # Loop over all name lists.
   idx=Ø
   while [ $idx -lt $NL_COUNT ]
   do
      nl=${CL_NAMELIST[$idx]}
      empty=${CL_NAMELIST_EMPTY[$idx]}

      # Get a list of cluster sender channel, which contain
      # the cluster name list.
      sdr_list='echo "DIS CHL(*) CHLTYPE(CLUSSDR) CLUSNL" | runmqsc
$QMGR_NAME | grep -w $nl | awk '{print $1}' | sed -e "s/
^.*CHANNEL(\(.*\)).*$/\1/"'

      for sdr in $sdr_list
      do
         # ...if the 'empty' flag is set...
         if [ "$empty" = "yes" ]
         then
            # ...remove the cluster name list attribute.
            echo "ALTER CHL($sdr) CHLTYPE(CLUSSDR) CLUSTER(' ')" |
runmqsc $QMGR_NAME >> $LOG_FILE 2>&1
         else
            # ...otherwise alter the attribute to the
            #      temporary name list.
            tmp_nl=${CL_NAMELIST_TEMP[$idx]}
            echo "ALTER CHL($sdr) CHLTYPE(CLUSSDR) CLUSNL($tmp_nl)" |
runmqsc $QMGR_NAME >> $LOG_FILE 2>&1
         fi
      done
      # Restart the channel to activate the modifications above.
      recycle_channel $sdr_list
      ret=$?
      [ $ret -ne Ø ] && return $ret
      # Increase the counter.
      let idx=$idx+1
   done
}
#
# Function to replace the temporary name lists by the - modified -
# original name lists.
#
function replace_modified_namelists
{
   $DEBUG
   # Loop over all name lists.
   idx=Ø
   while [ $idx -lt $NL_COUNT ]
```

```
   do
      empty=${CL_NAMELIST_EMPTY[$idx]}
      nl=${CL_NAMELIST[$idx]}
      tmp_nl=${CL_NAMELIST_TEMP[$idx]}
      # If name list is marked as empty...
      if [ "$empty" = "yes" ]
      then
         # ...Delete the name list.
         echo "DELETE NL($nl)" | runmqsc $QMGR_NAME > /dev/null 2>&1
      else
         # ...Otherwise redefine original name list.
         echo "DELETE NL($nl)" | runmqsc $QMGR_NAME > /dev/null 2>&1
         echo "DEFINE NL($nl) LIKE($tmp_nl)" | runmqsc $QMGR_NAME > /
dev/null 2>&1

         # Replace temporary name list in all queues by the - modified -
         # original name list.
         for type in QLOCAL QALIAS QREMOTE QMODEL
         do
            q_list='echo "DIS $type(*) CLUSNL" | runmqsc $QMGR_NAME |
grep -w $tmp_nl | sed -e "s/^.*QUEUE(\(.*\)).*$/\1/"'

            for queue in $q_list
            do
               echo "ALTER $type($queue) CLUSNL($nl)" | runmqsc
$QMGR_NAME >> $LOG_FILE 2>&1
            done
         done
        # Replace temporary name list in all channel by the - modified -
         # original name list.
         for type in CLUSRCVR CLUSSDR
         do
            chl_list='echo "DIS CHL(*) CHLTYPE($type) CLUSNL" | runmqsc
$QMGR_NAME | grep -w $tmp_nl | awk '{print $1}' | sed -e "s/
^.*CHANNEL(\(.*\)).*$/\1/"'

            for chl in $chl_list
            do
               echo "STOP CHL($chl) MODE(FORCE) STATUS(INACTIVE)" |
runmqsc $QMGR_NAME >> $LOG_FILE 2>&1
               echo "ALTER CHL($chl) CHLTYPE($type) CLUSNL($nl)" |
runmqsc $QMGR_NAME >> $LOG_FILE 2>&1
          echo "START CHL($chl)" | runmqsc $QMGR_NAME >> $LOG_FILE 2>&1
            done
         done
         # Delete temporary name list.
         echo "DELETE NL($tmp_nl)" | runmqsc $QMGR_NAME > /dev/null 2>&1
      fi
      # Increase the counter.
      let idx=$idx+1
```

```
   done
}
#
# Remove the queue manager from the cluster
#
function execute_remove_cluster
{
   $DEBUG
   echo "$SEP_LINE"
   echo "Start the cluster removal."
   # Ask, if the cluster removal shall be started.
   ans=""
   echo "   Start the cluster removal now (y/n) ? \c"
   read ans
   if [ "$ans" != "y" -a "$ans" != "Y" ]
   then
      echo "INFO: Cluster removal cancelled by user!"
      return 1
   fi
   # Suppress some escape charaters (CTRL-C, CTRL-D ...).
   trap '' HUP INT ERR TERM
   # First now suspend the queue manager from the cluster.
   echo "   Suspend queue manager."
   echo "SUSPEND QMGR CLUSTER($CLUS_NAME)" | runmqsc $QMGR_NAME >>
$LOG_FILE 2>&1

   # Then create new name lists without the cluster name, from
   # which the queue manager shall be removed, for later use.
   create_temp_namelists
   # Remove now the queues from the cluster by altering the
   # cluster attributes.
   remove_queues_from_cluster
   # Remove now the queue manager from the cluster by altering the
   # cluster receiver channel.
   alter_cluster_receiver
   ret=$?
   [ $ret -ne Ø ] && return $ret
   # Interrupt now the connection between the queue manager and the
   # full repositories by altering the cluster sender channel.
   alter_cluster_sender
   ret=$?
   [ $ret -ne Ø ] && return $ret
   # Replace the temporary name lists by the -
   #    modified - original lists.
   echo "   Replace modified name lists."
   replace_modified_namelists
   # Last but not least drop the data of the own partial repository.
   echo "   Dropping cluster data."
   echo "REFRESH CLUSTER($CLUS_NAME) REPOS(YES)" | runmqsc $QMGR_NAME >>
$LOG_FILE 2>&1
```

```
   return 0
}
#################################################################
#   MAIN program
#################################################################
#DEBUG="set -x"
# Initialize the script environment
# Prepare the deinstallation
prepare_remove_cluster $0 $*
ret=$?
# Execute the deinstallation
if [ $ret -eq 0 ]
then
   execute_remove_cluster
   ret=$?
fi
# Finish the script logging and exit the program.
echo "$SEP_LINE"
exit $ret
```

Hubert Kleinmanns
Senior Consultant
N-Tuition Business Solutions AG (Germany)

Identifying any MQ in-doubt units of work

The MQUOWS REXX EXEC is designed to identify any MQ in-doubt units of work at CICS shutdown. This program is meant to be added to the end of the CICS PROC as a step after DFHSIP. The program reads the CICS SYSIN and scrapes out the INITPARM for MQ and the APPLID to format a valid MQ command to display all in-doubt units of work for that particular CICS region. If any in-doubts are found, the messages are written to the console so automation can pick them up. This can help avoid a CICS COLD or INIT start if any in-doubts are found. A CICS INIT start while in-doubt units of work exist can cause pageset corruption. Sample JCL is included in the program comments.

MQUOWS REXX EXEC

```
/*********************************************************************/
/*                                REXX                               */
/*********************************************************************/
/* Purpose: Check for MQ indoubt units of work at CICS shutdown      */
/*-------------------------------------------------------------------*/
/* Syntax:  mquows                                                   */
/*-------------------------------------------------------------------*/
/* Parms: N/A        - N/A                                           */
/*                                                                   */
/* Notes: Add as a step at the end of the CICS PROC                  */
/*        Depends on finding the CICS INITPARM SN= value in the CICS */
/*        SYSIN parms.  The CICSPARM DD must point to the correct    */
/*        CICS SYSIN.                                                */
/*                                                                   */
/*        Sample execution JCL:                                      */
/*                                                                   */
/*        //MQUOWS   EXEC PGM=IKJEFT01,PARM='MQUOWS'                */
/*        //STEPLIB  DD   DSN=mq.csqauth,DISP=SHR                    */
/*        //         DD   DSN=mq.csqanle,DISP=SHR                    */
/*        //SYSEXEC  DD   DSN=rexx.exec.pds,DISP=SHR                 */
/*        //CICSPARM DD   DSN=cics.sysin.pds,DISP=SHR                */
/*        //SYSTSPRT DD   SYSOUT=*                                   */
/*        //SYSTSIN  DD   DUMMY                                      */
/*                                                                   */
/*********************************************************************/
/*                      Change Log                                   */
/*                                                                   */
/*  Author       Date      Reason                                    */
/*  --------    --------  ------------------------------------------ */
/*********************************************************************/
/* Ensure all required DDs are present                               */
/*********************************************************************/
EXITRC = listdsi('STEPLIB' 'FILE')
if EXITRC <> 0 then
    do
     say 'STEPLIB is missing RC='EXITRC
     signal shutdown
    end
EXITRC = listdsi('SYSEXEC' 'FILE')
if EXITRC <> 0 then
    do
     say 'SYSEXEC is missing RC='EXITRC
     signal shutdown
    end
EXITRC = listdsi('CICSPARM' 'FILE')
if EXITRC <> 0 then
    do
     say 'CICSPARM is missing RC='EXITRC
```

```
     signal shutdown
    end
/*********************************************************************/
/* Read the CICS SYSIN dataset                                       */
/*********************************************************************/
"EXECIO * DISKR CICSPARM (STEM CICSPARM. FINIS"
EXITRC = RC
if EXITRC <> 0 then
    do
     say 'EXECIO error on CICSPARM RC='EXITRC
     signal shutdown
    end
/*********************************************************************/
/* Parse the CICS SYSIN looking for the QMGR                         */
/*********************************************************************/
do i=1 to cicsparm.0
    select
      when pos("'SN=",cicsparm.i) <> 0 then
           parse var cicsparm.i . "'SN=" qmgr "," .
      when pos("APPLID=",cicsparm.i) <> 0 then
           parse var cicsparm.i "APPLID=" applid .
      otherwise nop
    end
end
/*********************************************************************/
/* Allocate required datasets to VIO (SYSPRINT SYSIN CMDIN)          */
/*********************************************************************/
"ALLOC F(SYSPRINT) UNIT(VIO) SPACE(1 5) CYLINDERS NEW"
EXITRC = RC
if EXITRC <> 0 then
    do
     say 'ALLOC error on SYSPRINT RC='EXITRC
     signal shutdown
    end
"ALLOC F(SYSIN) UNIT(VIO) SPACE(1 5) CYLINDERS NEW LRECL(80)"
EXITRC = RC
if EXITRC <> 0 then
    do
     say 'ALLOC error on SYSIN RC='EXITRC
     signal shutdown
    end
"ALLOC F(CMDIN) UNIT(VIO) SPACE(1 5) CYLINDERS NEW LRECL(80)"
EXITRC = RC
if EXITRC <> 0 then
    do
     say 'ALLOC error on CMDIN RC='EXITRC
     signal shutdown
    end
/*********************************************************************/
/* Load records into SYSIN and CMDIN using EXECIO                    */
```

```
/*********************************************************************/
sysin.1 = 'COMMAND DDNAME(CMDIN)'
"EXECIO * DISKW SYSIN (STEM SYSIN. FINIS"
EXITRC = RC
if EXITRC <> Ø then
    do
     say 'EXECIO error on SYSIN RC='EXITRC
     signal shutdown
    end
cmdin.1 = 'DISPLAY THREAD('applid') TYPE(INDOUBT)'
"EXECIO * DISKW CMDIN (STEM CMDIN. FINIS"
EXITRC = RC
if EXITRC <> Ø then
    do
     say 'EXECIO error on CMDIN RC='EXITRC
     signal shutdown
    end
/*********************************************************************/
/* Report what the EXEC will do                                      */
/*********************************************************************/
say 'Checking' qmgr 'for indoubts from' applid
say
say 'Issuing MQ command:' cmdin.1
say
/*********************************************************************/
/* Run CSQUTIL                                                       */
/*********************************************************************/
address ATTCHMVS 'CSQUTIL' 'QMGR'
EXITRC = RC
if EXITRC <> Ø then say 'CSQUTIL error RC='EXITRC
/*********************************************************************/
/* Read CSQUTIL output                                               */
/*********************************************************************/
"EXECIO * DISKR SYSPRINT (STEM SYSPRINT. FINIS"
EXITRC = RC
if EXITRC <> Ø then
    do
     say 'EXECIO error on SYSPRINT RC='EXITRC
     signal shutdown
    end
/*********************************************************************/
/* Parse CSQUTIL output                                              */
/*********************************************************************/
do o=1 to sysprint.Ø
    select
/*********************************************************************/
/* Exclude all extraneous lines                                      */
/*********************************************************************/
      when word(sysprint.o,2) = 'CSQUØØØI'  then iterate
      when word(sysprint.o,2) = 'CSQUØØ1I'  then iterate
```

```
      when word(sysprint.o,1) = 'COMMAND'   then iterate
      when word(sysprint.o,2) = 'CSQU127I'  then iterate
      when word(sysprint.o,2) = 'CSQU120I'  then iterate
      when word(sysprint.o,1) = 'CSQU121I'  then iterate
      when word(sysprint.o,2) = 'CSQU055I'  then iterate
      when word(sysprint.o,1) = 'DISPLAY'   then iterate
      when word(sysprint.o,1) = '0CSQN205I' then iterate
      when word(sysprint.o,1) = 'CSQV401I'  then iterate
      when word(sysprint.o,1) = 'CSQ9022I'  then iterate
      when word(sysprint.o,2) = 'CSQU057I'  then iterate
      when word(sysprint.o,2) = 'CSQU058I'  then iterate
      when word(sysprint.o,2) = 'CSQU143I'  then iterate
      when word(sysprint.o,2) = 'CSQU144I'  then iterate
      when word(sysprint.o,2) = 'CSQU148I'  then iterate
/*********************************************************************/
/* Print the expected lines                                          */
/*********************************************************************/
/* GOOD response RC=0                                                */
/*********************************************************************/
      when word(sysprint.o,1) = 'CSQV412I' then
           do
           "SEND" "'"strip(sysprint.o,'T')"'"
            say strip(sysprint.o,'T')
            EXITRC = 0
           end
/*********************************************************************/
/* "BAD" response RC=20                                              */
/*********************************************************************/
      when word(sysprint.o,1) = 'CSQV406I' then
           do
           "SEND" "'"strip(sysprint.o,'T')"'"
            say strip(sysprint.o,'T')
            EXITRC = 20
           end
/*********************************************************************/
/* Print anything unexpected                                         */
/*********************************************************************/
      otherwise say strip(sysprint.o,'T')
    end
end
/*********************************************************************/
/* Shutdown                                                          */
/*********************************************************************/
shutdown: nop
           call outtrap trash. 0
          "FREE F(SYSPRINT)"
          "FREE F(SYSIN)"
          "FREE F(CMDIN)"
           exit(EXITRC)
```

Integrating COBOL applications with Microsoft BizTalk Server 2004

INTRODUCTION

Recently we completed a project, Multiconn BizTalk Extensibility, that provides an easy approach to integrating legacy COBOL applications into the Microsoft BizTalk Server 2004. This article describes the approach used by us.

OVERVIEW

Sometime software developers face the challenge of integrating two or more different systems into one solution to provide a wider service to their clients, or to expose some functionality of one system to the other one. One such example is the integration of a COBOL application that works on a legacy system (eg mainframe OS/390) into Microsoft BizTalk Server 2004.

On the one hand, the current legacy systems provide external interfaces for communication. One such interface is CICS. This is an application-programming interface that enables a client program to call a server program running in a CICS region and to pass and receive data. To use these legacy applications, MQSeries is used as a transport mechanism. A client application (that can run on any platform) starts a CICS application by sending a structured message to the CICS Bridge through MQSeries. Any data required by the CICS application can be included in the request message. Similarly, the CICS application can send data back to the client application in a message that is sent to a reply queue.

On the other hand, the Microsoft BizTalk Server 2004 provides an easy way to create complex business processes without excessive effort. It enables users to seamlessly integrate diverse applications and then use services exposed by those applications in a homogeneous environment.

To retrieve and send messages from/to an external system, the BizTalk Server requires the existence of a corresponding adapter that supports a transport mechanism to that system. And, in order to take full advantage of BizTalk Server processing, the messages must be transformed from their native format into its XML representation. BizTalk Server pipelines perform this transformation of incoming and outgoing messages.

Hence, it is necessary to implement the corresponding adapter and pipeline components that can deliver and then convert COBOL records to/from XML documents in order to integrate legacy COBOL applications into Microsoft BizTalk Server 2004.

The Multiconn BizTalk Extensibility project provides an implementation of an MQ adapter and pipeline components that support a CICS DPL conversation.

ABOUT THE MULTICONN BIZTALK EXTENSIBILITY PROJECT

This project provides an easy way to integrate legacy COBOL applications that access Microsoft BizTalk Server 2004 orchestrations through a CICS DPL bridge. The project includes the following components:

- The MQ adapter, which supports both the MQSeries server and client connections. It supports receive, send, solicit-response, and request-response communication patterns.
- Pipeline components that do run-time data transformations and support the CICS DPL conversation protocol.
- Design-time tools to generate annotated XML schemas and C# source from COBOL copybooks. The assemblies generated from these sources are used by CICS DPL pipeline components at run-time to transform XML to and from COBOL raw data messages.

The project includes documentation, executable files, and

sources of the adapter, pipeline components, and examples that demonstrate their capabilities. The project is arranged as a Microsoft Visual Studio solution and can be downloaded from http://workspaces.gotdotnet.com/MulticonnBiztalkExtensibility.

PREREQUISITES

The project requires the following components to be installed before it can start:

- .NET Framework Version 1.1.
- Microsoft BizTalk Server 2004 (the evaluation version can be downloaded from www.microsoft.com/biztalk/evaluation/trial/default.asp).
- Either MQSeries Version 5.3 server or client.
- Support pack MA7P for MQSeries V5.3 (MQ classes for Microsoft .NET). You can download this support pack free of charge from www-1.ibm.com/support/docview.wss?rs=203&uid=swg24004732&loc=en_US&cs=utf-8&lang=en.

Optionally, to build the solution and samples, the Microsoft Visual Studio .NET 2003 is required.

The project works under Windows 2000/XP/2003.

INSTALLATION

The following procedure walks you through the installation and configuration of the project:

1 Right-click the Multiconn BizTalk Extensibility.msi file. Select the **Install** menu item.

2 Specify the folder in which to install run-time files, source, and samples.

3 Follow the wizard's instructions.

4 Click the **Start** button, select the **Programs** menu item, select the **Microsoft BizTalk Server 2004** menu item, and then select the **BizTalk Server Administration** application.

5 In the **BizTalk Administration Console**, double-click the **Microsoft BizTalk Server 2004 (local)** node, expand **Hosts**, and then select the **BizTalkServerApplication** in the left pane.

6 In the results pane, right-click the host instance (typically, the computer name), and then click **Stop**. The status of the host instance changes to **Stopped**.

7 In the results pane, right-click the host instance, and then click **Start**. The status of the host instance changes to **Start pending**. You must click **Refresh**, or right-click the host instance and then click **Refresh**, to change the status to **Running**.

Multiconn BizTalk Extensibility.msi installs and registers the following components:

- MQ adapter.
- CICS DPL pipeline components.
- Design-time generating tools.
- Registers the cobol-schema.xsd file, which allows the editing of annotated XML files in Microsoft Visual Studio .NET 2003.
- Sources of the Multiconn BizTalk Extensibility project.
- Samples for the adapter and pipeline components testing.

Note: it's strongly recommended that you read the documentation carefully before starting to work with this adapter and pipeline components.

HOW IT WORKS

The MQ adapter is the first and last BizTalk components that

touch a message within the BizTalk system and is used to exchange messages between applications and BizTalk Server 2004 through either the MQSeries server or client. It is a component written in Microsoft Visual C# .NET. The adapter is created and hosted within the BizTalk service process. This means that BizTalk creates and manages the lifetime of the adapter, initializes it, services the adapter requests, and terminates the adapter on service shutdown.

Actually the project provides two implementations of an MQ adapter – the receive adapter (receiver), and the send adapter (transmitter).

The receive adapter listens to the MQ queue for an incoming message. When a message is received, the receive adapter hands off the message to the Messaging Engine, which passes the message through a receive pipeline, and then persists it in the MessageBox database. When the receive adapter works in request-response mode, it receives a request message from the client, submits it to the BizTalk server, waits for a response, and then sends the response back to the client.

The send adapter gets a message from a send pipeline and puts it to the destination queue as is. The pipeline's responsibility is to serialize a BizTalk message to a corresponding MQ record. When the send adapter works in solicit-response mode, it puts a request message to the destination queue, waits for a response message, and then submits the response message back to the BizTalk server.

For a BizTalk Server 2004 application to execute a business process, it must be able to correctly process messages containing documents in the format of the application. A message pipeline performs this processing. Incoming messages are processed through a receive pipeline, while outgoing messages go through a send pipeline.

Internally BizTalk Server uses XML as the format for processing all documents. To handle the task of unifying different document

formats, the BizTalk Server 2004 provides pipeline components (receive and send) that developers can create to customize the conversion of XML messages to and from messages in the format of the application.

Our CICS DPL pipeline components (DplSerializer and DplDeserializer) allow XML messages to be transformed to a raw data format that corresponds to the format of communication of the CICS DPL bridge. The DplSerializer pipeline component is intended for use in the assemble stage of a receive pipeline, and the DplDeserializer pipeline component is used in the disassemble stage of a send pipeline.

The schema of their work is quite simple: when DplSerializer gets an XML message to convert, it transforms the XML it contains into an instance of a class (the type of class that corresponds to XML is stored in a property of the DplSerializer). The next operation is to convert the instance of the type into raw data for COBOL. For this the COBOL serializer component is used.

DplDeserializer works in quite the opposite way. It uses a COBOL serializer component to transform a raw data stream into an instance of a class, and then a standard component called XmlSerializer is used to convert the instance to XML.

But where did we get such nice types that can be serialized in to both XML and COBOL raw data? The answer is that our design-time tools can generate such types!

Please note that the correct contact address for Xephon Inc is PO Box 550547, Dallas, TX 75355, USA. The phone number is (214) 340 5690, the fax number is (214) 341 7081, and the e-mail address to use is info@xephon.com.

There are two of them – the COBOL copybook parser and the COBOL importer. The COBOL copybook parser generates annotated XML schemas from COBOL copybooks. A developer can edit these schemas using an appropriate XML schema editor (eg Microsoft Visual Studio) in order to tune outgoing types. The COBOL importer consumes these annotated XML schemas and generates C# source and optionally assemblies. Classes generated using these tools contain annotation attributes for XML serialization and attributes for COBOL serialization (that the COBOL serializer component performs).

As for the COBOL serializer component, we see it as a counterpart (in some sense) to XmlSerializer. It inspects class meta-data, and creates and caches plans for COBOL serialization and deserialization. The next time that COBOL serialization is invoked, the cached plan is used.

The following workflow describes the BizTalk Server behaviour at run-time:

1 A user-defined business process (orchestration) generates a request message for a COBOL application and transmits it to the send port.

2 Then the produced message goes into the send CICS DPL pipeline. The pipeline serializes the properly-filled MQCIH header and this message to an MQ record, and passes it to an MQ transmitter (send MQ adapter).

3 The transmitter establishes a connection to the specified endpoint and sends the request through MQ to the CICS DPL bridge on a legacy system.

4 Then BizTalk starts the MQ receiver, if it is not yet started. The receiver is implemented as an MQ listener that listens to the incoming MQ queue. The listener waits for a response from the COBOL application.

5 When the response is received by the MQ adapter, it hangs off to receive the CICS DPL pipeline.

6 This pipeline deserializes messages from the received

MQ record to the XML document, and transmits it to the orchestration through the receive port.

Note: the serialization/deserialization of MQCIH header and BizTalk messages are performed by assemblies that were generated at design-time.

Editor's note: this article will be concluded next month.

Arthur and Vladimir Nesterovsky (Israel)

MKS has announced that it has expanded the integration of MKS Integrity Manager with Eclipse, WebSphere, and Mercury Quality Center.

MKS Integrity Manager is a process and workflow solution. It allows an organization to define and implement a repeatable development process. Integration with Eclipse and WebSphere development environments brings the capabilities of the product directly to the developer's desktop. Using the MKS 'WorkTray', developers on distributed, mainframe, and iSeries platforms can receive work assignments (tasks), run queries, review, and update, without ever stepping outside their integrated development environment (IDE). All software changes are easily linked to approved development tasks, providing complete traceability of all development work.

For further information contact:
MKS, 410 Albert Street, Waterloo, ON N2L-3V3, Canada.
Tel: (519) 884 2251.
URL: www.mks.com/press/index.jsp?action=readarticle&article_id=8714.

* * *

Willow Technology has announced its Ectropyx integration platform for sensors. Ectropyx is designed to provide online, secure, and reliable integration between sensors and enterprise applications and databases, allowing centralized administration and control of sensors and controllers, as well as the processing of collected sensor data. For sensors such as RFID readers, this permits true online transaction processing.

Supported environments include WebSphere MQ, WebSphere Integration Broker, and JMS providers such as WebSphere, BEA WebLogic, and the open source JBoss Application Server. Supported Integration Server platforms are AIX, HP-UX, Linux, Solaris, and Windows.

For further information contact:
Willow Technology, 900 Lafayette Street, Suite 604, Santa Clara, CA 95050-4967, USA.
Tel: (408) 296 7400.
URL: www.willowtech.com.

* * *

Ember, a maker of ZigBee chips, and Arcom have created a network gateway that can sit between a firm's ZigBee-powered wireless sensor network and its WebSphere middleware. ZigBee is a wireless technology allowing electronic devices to connect to the Internet.

Basically, it provides an end-to-end telemetry communications gateway that pumps ZigBee network data from remote devices to WebSphere MQ Integrator. The gateway itself is a single piece of equipment, powered by a 400MHz Intel RISC processor. It runs an embedded version of Linux, as well as an embedded version of Java.

For further information contact:
Ember, 343 Congress St, 5th Floor, Boston, MA 02210, USA.
Tel: (617) 951 0200.
URL: www.ember.com/company/press/press-101204.html.

* * *