43

MQ update

January 2003

In this issue

MQ Update

Published by

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

North American office

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

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.

Editor

Madeleine Hudson
E-mail: MadeleineH@xephon.com

Subscriptions and back-issues

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

Contributions

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

MQ Update on-line

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

Printed in England.

Creating self-executing, self-documenting scripts

Scripts are excellent tools for MQSeries administration but managing large numbers of scripts presents its own set of challenges. The output of configuration scripts is a valuable tool for forensic analysis and diagnostics; however, the collection and management of script output is frequently overlooked. It is, therefore, often difficult to know when a script was last run, whether it was successful, when it was last changed, or what the changes were. These issues are usually addressed through procedures that specify how the technician is to run the script, where to capture the output, what to name the output, and so forth. Even this leaves much to chance as it depends on the technician to construct the command-line arguments correctly every time a script is executed. A more reliable approach is to embed these policies within the script so that nothing is left to chance. With a few lines of code, **runmqsc** and **setmqaut** scripts can be self-executing and self-documenting.

SAMPLE MQSC SCRIPT

The script in Listing One uses **runmqsc** to secure the SYSTEM.DEF.SVRCONN channel by altering the MCAUSER attribute to contain the value 'nobody'. In addition to the commands the script contains the usual comment lines and change log.

LISTING ONE: QMGR.MQSC

```
* qmgr.mqsc
* Setup QMgr per shop standards
* Ø2-1Ø-Ø2 T.Rob - New Script
* ------------------------------
ALTER CHL(SYSTEM.DEF.SVRCONN) +
     Chltype(svrconn) +
     MCAUSER('nobody')
```

OUTPUT FROM LISTING ONE: QMGR.MQSC.OUT

```
Ø783889, 5765-B75 (C) Copyright IBM Corp. 1994, 2ØØØ. ALL RIGHTS
RESERVED.
```

```
Starting MQSeries Commands.
       : * qmgr.mqsc
       : * Setup QMgr per shop standards
       : * Ø2-1Ø-Ø2 T.Rob - New Script
     1 : ALTER CHL(SYSTEM.DEF.SVRCONN) +
       :      chltype(svrconn) +
       :      MCAUSER('nobody')
AMQ8Ø16: MQSeries channel changed.
One MQSC command read.
No commands have a syntax error.
All valid MQSC commands were processed.
```

The script is named 'qmgr.mqsc' and is executed with the command **runmqsc <qmgr.mqsc >qmgr.mqsc.05-30-02.out.**

There are several problems with this script. When run, it is possible that the output will overwrite some previous output or be lost entirely, depending on what is entered on the command line. This could be a problem if the previous output files provide the only history of changes to the queue manager. Additionally, nothing in the output file indicates which queue manager the script was run on. Finally, there is nothing to prevent mistakes, such as accidentally running this script through **setmqaut** rather than **runmqsc**. In general, too many factors are left to chance. Although the risk is small it is multiplied each time the script is executed. Given enough executions of the script that small risk becomes a certainty.

The approach we will take in addressing these issues will be to encapsulate our manual procedures into the script itself. This will allow the configuration to be implemented up-front, subjected to peer review, and then executed reliably and accurately any number of times.

DISPLAYING RUN-TIME INFORMATION

One of the easiest tweaks we can apply here is one of the most overlooked. Simply add the command **dis qmgr qmname** to the beginning of all **runmqsc** scripts.

If you rely on comments in the script, the name of the script, or the name of the output file to determine where it was run, you are never really certain. When reviewing script output the only sure way to tell where it was run is to have the script display the queue

manager name.

Similarly, when reviewing output it is often difficult to know whether the entire file has been captured. To provide assurance that the output has not been truncated add a final comment – *** End of script** – to the end of the script.

MAKING THE SCRIPT SELF-EXECUTING

In order to make the script generate time-stamped output files it is first necessary to set it up to be self-executing. The objective here is that the command line used to execute the script is nothing more than the script name. This is accomplished by making the script executable (**chmod +x qmgr.mqsc**) and then placing the following two lines of code at the top of the script:

```
#!/usr/bin/ksh
/opt/mqm/bin/runmqsc <<EOF
```

The first line invokes the Korn shell. The second line captures lines three and onward and pipes them into **runmqsc** for processing.

MAKING THE SCRIPT SELF-DOCUMENTING

A self-documenting script is one which ensures any script comments are displayed in the output and that the output is captured into time-stamped multi-generation files. The file name will contain both the script name and a time-stamp so that a simple directory listing will reveal which scripts were executed and when. The change history may be derived by comparing output from successive runs. We previously set up line two of the sample script to capture STDIN. We can modify it to manage STDOUT and STDERR as well. The new line looks like this:

```
/opt/mqm/bin/runmqsc >$0.`date "+%y%m%d-%H%M"`.out 2>&1 <<EOF
```

You will probably want to adapt this to your shop standards so let's break it down. Recall that the first fragment executes **runmqsc** for us. The second fragment creates a unique file name to capture redirection from STDOUT. The file name is composed of '$0', which substitutes the name of the script, a time-stamp, and the constant '.out'.

The example saves the output in the same directory as the script; however, it is just as easy to save it somewhere else. Simply add the path name ahead of $0. It is wise to keep scripts and logs out of */opt/mqm* and */var/mqm* so that they are not lost if MQ ever needs to be reinstalled.

Note that the time-stamp is in YYMMDD order so that the files sort out chronologically. The time-stamp can be unique by day, hour, minute, or second, as appropriate. In the example above, the time-stamp is unique to the minute. Removing the '-%H%M'will create a time-stamp that is unique to the day, but be aware that, if the script is run more than once on a given calendar day, later executions will overwrite the earlier ones.

The constant '.out' at the end of the file allows for wildcard matching of output files. Since the files tend to accumulate over time the suffix is useful when writing scripts to prune the directory. Typically, several scripts reside in the same directory and their output logs will all have differing names. The suffix at the end will be the only way to match all the logs as a group. If your output logs are accessible via the Web you might want to use *.txt* or *.htm* as the suffix.

The fragment '2>&1' captures STDERR and redirects it to STDOUT. It is important that this fragment comes after the redirection of STDOUT because the shell evaluates the file handles in left-to-right order on the command line.

The final fragment '<<EOF' has already been covered. The new version of the script is provided as Listing Two.

LISTING TWO: REVISED QMGR.MQSC

```
#!/usr/bin/ksh
/opt/mqm/bin/runmqsc >$Ø.`date "+%y%m%d-%H%M"`.out 2>&1 <<EOF
* qmgr.mqsc
* Setup QMgr per shop standards
* Ø2-1Ø-Ø2 T.Rob - New Script
* ------------------------------
dis qmgr qmname
ALTER CHL(SYSTEM.DEF.SVRCONN) +
     Chltype(svrconn) +
     MCAUSER('nobody')
```

```
* End Of Script
```

OUTPUT FROM LISTING TWO: QMGR.MQSC.020530-1936.OUT

```
Ø783889, 5765-B75 (C) Copyright IBM Corp. 1994, 2ØØØ. ALL RIGHTS
RESERVED.
Starting MQSeries Commands.
       : * qmgr.mqsc
       : * Setup QMgr per shop standards
       : * Ø2-1Ø-Ø2 T.Rob - New Script
     1 : DIS QMGR QMNAME
AMQ84Ø8: Display Queue Manager details.
   QMNAME(QM1)
     2 : ALTER CHL(SYSTEM.DEF.SVRCONN) +
       :      chltype(svrconn) +
       :      MCAUSER('nobody')
AMQ8Ø16: MQSeries channel changed.
       : * End Of Script
2 MQSC commands read.
No commands have a syntax error.
All valid MQSC commands were processed.
```

ADVANCED USAGE

If you run more than one queue manager per server in your shop you have probably realized that the examples used so far will run only on the default queue manager. Solving this problem requires a decision to either hard-code the queue manager as a parm to **runmqsc** or to make an exception to our original design goals and allow a command-line option. Hard-coding the name is rather easy: simply add it to line two where appropriate:

```
   /opt/mqm/bin/runmqsc QM1 >QM1.$Ø.`date "+%y%m%d-%H%M"`.out 2>&1 <<EOF
```

Adding the queue manager name as a command-line option is a little more involved. At the very least, we would like to do a reality check on the command line and make sure that the specified queue manager exists. If not, the script should print an error message and exit with a non-zero return code. An example of one way to do this is provided below. Note that the queue manager name is now embedded in the output file name.

LISTING THREE: QMGR.MQSC WITH COMMAND-LINE PARM

```
#!/usr/bin/ksh
```

```
if [[ -z $1 ]]
then
     Target=" "
     AbortMsg="default"
else
     Target=$1
     AbortMsg="$Target"
fi
# runmqsc will tell us whether the requested QMgr is alive
QMgr=`echo "dis qmgr qmname"       \
    | /opt/mqm/bin/runmqsc $Target \
    | tr ")" "\n"                  \
    | grep QMNAME                  \
    | tr "(" "\n"                  \
    | grep -v QMNAME`
if [[ $QMgr = "" ]] then
     echo "$Ø: Aborting! Could not find $AbortMsg QMgr"
     exit 255
fi
/opt/mqm/bin/runmqsc $QMgr >$Ø.$QMgr.`date "+%y%m%d-%H%M"`.out << EOF
* qmgr.mqsc
* Setup QMgr per shop standards
* Ø2-1Ø-Ø2 T.Rob - New Script
* ------------------------------
DIS QMGR QMNAME
ALTER CHL(SYSTEM.DEF.SVRCONN) +
     chltype(svrconn) +
     MCAUSER('nobody')
* End Of Script
```

EXTENDING THE CONCEPTS

Although the examples used **runmqsc** scripts the technique can be applied to **setmqaut** or any other utility for which input is normally prepared as a text file. Simply prepend any existing scripts with the ksh lines from Listing Two or Listing Three.

We touched briefly on making the log files available through a Web browser. In order for the Web server to recognize the files the suffix must be a known MIME type. Most Web servers will recognize *.txt* files with no modifications. If you create the files with an *.htm* extension the server will expect to find HTML-formatted documents. To make the output HTML compatible simply echo an HTML header before executing **runmqsc**:

```
#!/usr/bin/ksh
echo "<html><body><pre>"
```

```
/opt/mqm/bin/runmqsc >$0.`date "+%y%m%d-%H%M"`.out 2>&1 <<EOF
```

The <pre> tag must be included because it tells the browser to ignore HTML entities such as the < and > characters and to honour line breaks.

Although beyond the scope of this article, it is not difficult to extend the techniques presented here to include parameter substitution within the script text itself. This allows **setmqaut** scripts (in which the queue manager name is part of the command syntax) to be portable across queue managers. With **runmqsc**, parameter substitution allows for automatic generation of object names that are derived in part from the queue manager name. For example, many shops require channel names to contain the queue manager name.

T Robert Wyatt (USA)

Writing WMQI plug-in nodes in Java

INTRODUCTION

With WMQI 2.1 you can now write plug-in nodes in Java. In earlier versions of MQSI this support was restricted to C language only. Since Java is a platform-independent language your plug-in can be used across different platforms without modifying the source code. Custom plug-in code written in Java is compiled and packaged into jar files. The broker loads this jar file during start-up and makes the functionality available to the message flows. The *jplugin.jar* file contains important classes to write Java plug-ins, which are packaged into the *com.ibm.broker.plugin* package.

There are two main interfaces provided to write two different types of plug-in nodes in WMQI. They are:

- *MbInputNodeInterface* – use this interface if you are writing a custom plug-in input node.

- *MbNodeInterface* – use this interface if you are writing a custom plug-in node.

Note that you cannot write a plug-in parser in Java; it can be written only in C.

In this article I will try to explain how to write a simple Java plug-in node with an example. We will name our example plug-in node 'MsgFlowEnvInfo'.

MSGFLOWENVINFO PLUG-IN

There is no direct way of retrieving WMQI runtime information, such as broker name, queue manager name, message flow name etc, for a message flow. There are few plug-in utility functions available which will return this information at runtime, but it is very simple to write your own Java plug-in node to do this.

We will go through detailed steps on how to write and install a plug-in node and its components.

Files related to the plug-in node

- Required files are:
  - XML interface definition file
  - WDP file.
- Optional files are:
  - image files
  - help file
  - properties file
  - properties editor
  - a customizer.

You can use the *Smart Guide* in the WMQI Control Centre to generate some of these files. In this article we will create these files by hand; this will give us a chance to have a closer look at the

layout of different statements in these files.

Defining the plug-in node in the XML interface definition file

This file is in XML format and it is used to specify the configurable attributes, input/output terminals, and the image files (in GIF format) for the plug-in. The name of this file is 'MsgFlowEnvInfo' (without any extension). This file name is derived from our plug-in name, which is returned by the *getNodeName()* utility function call, without the suffix 'Node' text string.

Plug-in node definition

The following piece of code from the XML interface definition file describes the type of new node in conjunction with mandatory attributes and their values. There should be only one occurrence of <MessageProcessingNodeType> tag in the interface definition file.

The properties for these tags are shown in Table 1.

```
<MessageProcessingNodeType icon="images/MsgFlowEnvInfo.gif"
package="com.ingale.wmqi" creator="Kiran Ingale" version="1.0"
useDefaults="true" collectionPath="" scaleableIcon=""
versionTimestamp="" isPrimitive="true" longDescription=""
versionCreator="" creationTimestamp="" shortDescription=""
xmi.uuid="MsgFlowEnvInfo" xmi.id="MsgFlowEnvInfo"
xmi.label="MsgFlowEnvInfo">
```

Note that you could provide a key value for the shortDescription and longDescription attributes. The corresponding textual description will be looked up in the node's properties file while displaying them in the Control Centre.

Attributes

This <*Attribute*> tag defines attributes of the plug-in node. Since this plug-in is going to be very simple we will need only one attribute to store retrieved runtime information. The retrieved information will be stored in a Global Environment Tree under the name specified in the envTreeEleName attribute/property of the plug-in.

A plug-in node can have zero or more attributes. For each attribute we should define a single occurrence of the <Attribute> tag. We

Attribute/ tag name	Description
Icon	This field identifies the path of the icon file located in the images directory. The directory name is relative to *the <mqsi_root>\tool* directory. This icon is displayed in the Control Centre after you add this plug-in node to your workspace.
package	This identifies the location of the resources (property file, help file, customizer, property editor) for this plug-in node.
isPrimitive	Should be set to true always.
longDescription	This field provides textual description for the plug-in node. This text is displayed in the Long Description tab when you select properties for this plug-in node.
shortDescription	This text is displayed in the lower left corner when you click on the plug-in node in the Control Centre.
xmi.uuid	This field must contain the full name of the plug-in node without the suffix 'Node'. This identifier must be unique.

Table 1: Tag properties

could group these attributes using the <AttributeGroup> tag. If the attributes are not grouped they will be displayed in the default properties tab when the node properties are displayed.

The following piece of code determines what attribute our plug-in knows, a default value, and whether or not it is mandatory to specify a value. For our plug-in, if no value is specified for this attribute, it will create a Group Element with the name WMQIMsgFlowEnvInfo in the Environment Tree.

The properties for these tags are shown in Table 2.

```
<Attribute value="" xmi.uuid="envTreeEleName" encoded="true"
xmi.label="envTreeEleName" attributeOwner="" type="String"
valueMandatory="false"/>
```

Figure 1 illustrates a dialogue box for this plug-in, which is displayed when you select 'Properties' in the Control Centre.

Defining OutTerminal and InTerminal

Here I will explain how to define terminals for this plug-in node. We will need one input and two output terminals for this plug-in. <*OutTerminal*> and <*InTerminal*> tags define out and in terminals of the plug-in node. There should be an instance of *InTerminal* and *OutTerminal* tag for each input and output terminal defined in a plug-in node.

To define the input terminal we will create an instance of an <*InTerminal*> XML tag, for which the following attributes should be set:

- *xmi.uuid* – this value must be set to null ("").

Attribute/ Tag Name	**Description**
xmi.label	This field defines the label of an attribute to be displayed in the Control Centre. If a value is provided it extracts the long description for this value from the plug-in node's properties file. If the properties file does not have a matching value the value is displayed as it is.
type manuals.	This is the type of value that is stored in this attribute. For a list of possible values please refer to WMQI
value	This is the default value of the attribute.
xmi.uuid promoted.	This value is used internally when this attribute is
valueMandatory	This indicates whether the attribute is mandatory or optional.
encoded	A value of true or false indicates whether the value of this attribute needs to be encoded.

Table 2: Tag properties

- *xmi.label* – this field specifies the displayed name of the terminal. The value specified here should match the value specified in the code for a createInputTerminal() call. This field must be unique within this file.

```
<InTerminal icon="images/InTerminal.gif" creator="" orientation="9"
versionTimestamp="" longDescription="" y="" x="" versionCreator=""
creationTimestamp="" shortDescription="" xmi.label="in" xmi.uuid="">
<InTerminalTypeRef icon="images/InTerminal.gif" xml:link="simple"
xmi.label="InTerminalType" type="InTerminalType"
refType="InTerminalType" href="InTerminalType/InTerminalType"
title="InTerminal"/>
</InTerminal>
```

Similarly, to define two output terminals we will create an instance of <OutTerminal> XML tags. The attributes set for the output terminals are the same as those set for the input terminals.

```
<OutTerminal icon="images/OutTerminal.gif" creator="" orientation="9"
versionTimestamp="" longDescription="" y="" x="" versionCreator=""
creationTimestamp="" shortDescription="" xmi.label="out"
xmi.uuid="">
<OutTerminalTypeRef icon="images/OutTerminal.gif" xml:link="simple"
xmi.label="OutTerminalType" type="OutTerminalType"
refType="OutTerminalType" href="OutTerminalType/OutTerminalType"
title="OutTerminal"/>
</OutTerminal>

<OutTerminal icon="images/OutTerminal.gif" creator="" orientation="9"
versionTimestamp="" longDescription="" y="" x="" versionCreator=""
creationTimestamp="" shortDescription="" xmi.label="failure"
```

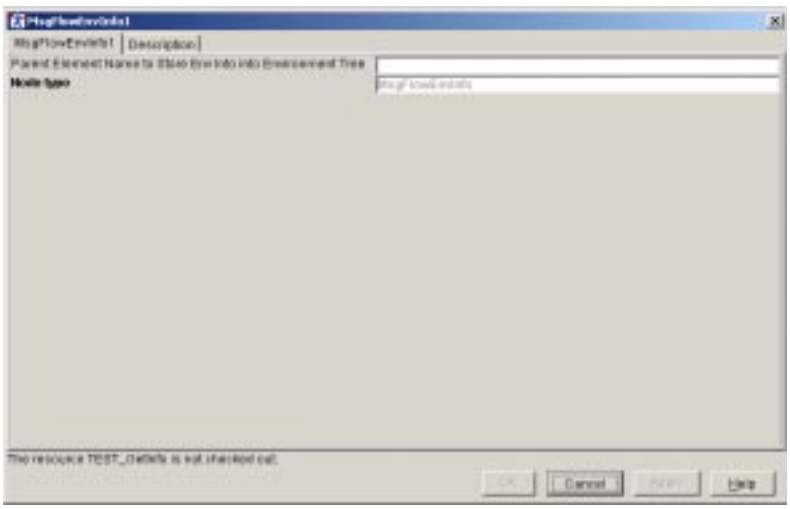

Figure 1: Dialogue box displayed when 'Properties' is selected

```
xmi.uuid="">
<OutTerminalTypeRef icon="images/OutTerminal.gif" xml:link="simple"
xmi.label="OutTerminalType" type="OutTerminalType"
refType="OutTerminalType" href="OutTerminalType/OutTerminalType"
title="OutTerminal"/>
</OutTerminal>
```

THE WEB DAV PROTOCOL (WDP) FILE

The WebDav protocol is used between the Control Centre and the Configuration Manager. This file provides information required by the protocol to send to the Configuration Manager. The file name should be the same as the XML interface definition file, with the extension *.wdp*.

The attributes shown in Table 3 must be set in the WDP file for this plug-in node.

```
<?xml version="1.0"?>
<properties xmlns:D="DAV:">
  <D:creationdate>2002-08-15T09:52:32-07:00</D:creationdate>
  <D:displayname>MsgFlowEnvInfo</D:displayname>
  <D:xmi.label>MsgFlowEnvInfo</D:xmi.label>
  <D:icon>images/MsgFlowEnvInfo.gif</D:icon>
  <D:versionStatus>new</D:versionStatus>
  <D:lockdiscovery xmlns:D="DAV:"/>
</properties>
```

ICONS FOR THE NODE

We should supply at least one icon to be used in the Control Centre tree view. We can also provide additional icon files to be displayed in Message Flow view when used while viewing the message flow in the Control Centre. If we don't supply these icon files the default icon will be used.

Optional Icon file names, with descriptions, with this plug-in node are:

- *MsgFlowEnvInfo.gif* – used when the plug-in is displayed in tree view.
- *MsgFlowEnvInfo30.gif* – used when the plug-in in the message flow is displayed in 25% zoom.

Attribute/ Tag Name	Description
displayname	This value is set to the name of the interface definition file.
xmi.label	This value will appear in the node tree in the Control Centre (Message Flows view) to identify the type of node. This value must be identical to the *xmi.label* property of the <*MessageProcessingNodeType*> tag in the XML interface definition file.
icon	This field will contain the path and name of the icon defined for this node relative to the <*mqsi_root*>*tool* directory. This value should identify the minimum size icon.

Table 3: Attributes that must be set in the WDP file

- *MsgFlowEnvInfo42.gif* – used when the plug-in in the message flow is displayed in 50% zoom.
- *MsgFlowEnvInfo58.gif* – used when the plug-in in the message flow is displayed in 75% zoom.
- *MsgFlowEnvInfo84.gif* – used when the plug-in in the message flow is displayed in 100% zoom.

DEFINING THE HELP TEXT FOR THE NODE

This step is optional. This file can provide help information for this plug-in node, which will be displayed when the user selects 'Help' from the plug-in properties dialogue box from the Control Centre. The format of this file is HTML. In this help file you can make use of the *bipnt.css* style sheet, which will make the appearance of the help text identical to standard plug-in help text.

The name of our help file must be *MessageProcessingNodeType_MsgFlowEnvInfo.htm*. As you can see here, the last part of the file name is identical to the *xmi.label* in the <*MessageProcessingNodeType*> tag in the XML interface definition file.

We will not go into the details of this file because it is very simple to understand the format, which is standard html.

EXPLORING JAVA SOURCE FOR THE PLUG-IN

Having decided attributes and terminals for this plug-in we will now start writing plug-in source code in Java. As per the Java language specifications the name of the source file should be the same as the class name. I will name this class 'MsgFlowEnvInfo' so our source file name will be 'MsgFlowEnvInfo.java'.

Since we are creating a new package for this plug-in we should declare the package name at the beginning of the source file:

```
package com.ingale.wmqi;
```

All plug-in related classes are packaged into a *com.ibm.broker.plugin.** package, so we need to import this package into our source code:

```
import com.ibm.broker.plugin.*;
```

To define a plug-in node we should sub-class the standard *MbNode* class and implement *MbNodeInterface*. Here, our plug-in node's class name is *MsgFlowEnvInfo* and it extends *MbNode* class. Since this is not an input node we will have to implement the *MbNodeInterface* interface.

The main class for this plug-in is declared as:

```
public class MsgFlowEnvInfo extends MbNode implements MbNodeInterface
```

We will now declare local variables to store plug-in attribute values, using the following code:

```
private String envTreeEleName;
```

The attributes label, usertTraceLevel, traeLevel, userTraceFilter, and traceFilter are already implemented as base configuration attributes. We should never implement these attributes in our code.

The *getNodeName()* method is called by the broker to retrieve the name of the plug-in node. The value returned by this method should be identical to the value specified in the *xmi.uuid* attribute in

<*MessageProcessingNodeType*> tag, without the 'Node' suffix.

```
public static String getNodeName() {
     return ("MsgFlowEnvInfoNode");
}
```

The constructor of the plug-in node class is called when the broker creates an instance of the plug-in node. This is when we should create output and input terminals using the *createOutputTerminal()* and *createInputTerminal()* methods respectively. The terminal names should match the values specified in the XML Interface definition file.

The following code extract shows how to do this:

```
public MsgFlowEnvInfo()
     throws MbException {
     createInputTerminal("in");
     createOutputTerminal("failure");
     createOutputTerminal("out");
}
```

For every attribute defined for this plug-in node we should have get/set methods defined. We must use standard JavaBean naming conventions for naming these methods. These attributes are received as a character string in XML messages, regardless of their data type.

```
public String getEnvTreeEleName() {
     return envTreeEleName;
}
public void setEnvTreeEleName(
           String elename) {
               this.envTreeEleName = elename;
}
```

Whenever a message is passed through a plug-in node its *evaluate()* method is called. This method is responsible for implementing the functionality of the plug-in node. The signature of this method is as follows:

```
public void evaluate(MbMessageAssembly messageAssembly,
MbInputTerminal inTerminal) throws MbException
```

messageAssembly represents the input message passed to this plug-in. An input message represents all four trees; you can call the appropriate utility method to retrieve a copy of these trees from the

input message assembly:

- *getMessage()* – returns the Message Tree.
- *getLocalEnvironment()* – returns the LocalEnvironment Tree.
- *getGlobalEnvironment()* – returns the Environment Tree.
- *getExceptionList()* – returns the ExceptionList Tree.

Since we are going to store data in the Environment Tree, using our plug-in, we should retrieve the Global Environment Tree from the message assembly, using the *getGlobalEnvironment()* method.

```
// Get Environment Tree from messageassembly
MbMessage envtree = messageAssembly.getGlobalEnvironment();
```

The *MbNode* class provides utility functions to retrieve MbBroker, MbExecutionGroup, and MbMessageFlow objects for this plug-in. We will call utility methods in our code to get these objects in order to extract runtime environment information.

```
// Get Broker object
MbBroker broker = getBroker();
// Get MessageFlow object
MbMessageFlow msgflow = getMessageFlow();
// Get ExecutionGroup object
MbExecutionGroup eg = getExecutionGroup();
```

Now we will call the utility methods of the above object to retrieve information (eg broker name, queue manager name, etc) and store it in local variables.

```
// Get Broker Data Source User-id
String  dsuid        = broker.getDataSourceUserId();
// Get Broker Name
String  bkname       = broker.getName();
// Get Broker's Queue Manager Name
String  qmgrname     = broker.getQueueManagerName();
// Get Execution Group Name
String  egname       = eg.getName();
// Get Message Flow Name
String  mfname       = msgflow.getName();
// Get Coodinated Transaction Property for this message Flow
boolean coordinated  = msgflow.isCoordinatedTransaction();
String tmode;
if ( coordinated )
     tmode = "True";
else
```

```
        tmode = "False";
```

After storing all the required information in local variables we will now add this information to the Environment Tree. To do this we need first to get the root element of the Environment Tree.

```
// Get Root Element for Environment Tree
MbElement EnvRootEle = envtree.getRootElement();
```

We will use the element name specified in the properties for this plug-in node. If no value is specified for this plug-in attribute we will create a group element with the name 'WMQIMsgFlowEnvInfo' in the Environment Tree and store the above retrieved information as its child elements.

```
// An element will be created by this name, if no value is specified
for this property
String envinfo = "WMQIMsgFlowEnvInfo";
// Create new element as Last Child into Environment Tree
MbElement envinfoele;
envinfoele =
EnvRootEle.createElementAsLastChild(MbElement.TYPE_NAME);
// Set Element name depending on the property value
if ( envTreeEleName != null) {
     envinfoele.setName(envTreeEleName);
} else {
     envinfoele.setName(envinfo);
}
// Create other elements as Child element to store Runtime
Environment Information
envinfoele.createElementAsLastChild(MbElement.TYPE_NAME_VALUE,
"BrokerName",bkname);
envinfoele.createElementAsLastChild(MbElement.TYPE_NAME_VALUE,
"QueueManagerName",qmgrname);
envinfoele.createElementAsLastChild(MbElement.TYPE_NAME_VALUE,
"ExecutionGroupName",egname);
envinfoele.createElementAsLastChild(MbElement.TYPE_NAME_VALUE,
"MessageFlowName",mfname);
envinfoele.createElementAsLastChild(MbElement.TYPE_NAME_VALUE,
"DataSourceUserId",dsuid);
envinfoele.createElementAsLastChild(MbElement.TYPE_NAME_VALUE,
"CoordinatedTransactionMode",tmode);
```

After we modify our Environment Tree we will propagate the updated *messageAssembly* through the out terminal.

```
// Propagate updated message to output terminal
MbOutputTerminal outTerminal = getOutputTerminal("out");
outTerminal.propagate(messageAssembly);
```

The following piece of code is optional. In this code we will catch *MbException* and log an error message in the event viewer. Before propagating an input message to the failure terminal we will first see if the terminal is connected or not. If the terminal is not connected this will throw a UserException, which will be caught by the framework.

If we don't provide a catch block in our code, the framework will catch the exception and perform a default action on it.

```
catch (MbException ex)
{
     // Log Error into Event Viewer
     MbService evtLog = new MbService();
     evtLog.logError(ex.getClassName(), ex.getMethodName(),
"WMQIv210","2951",
               ex.getTraceText(),ex.getInserts());
     // Get Failure Terminal
     MbOutputTerminal failureTerminal = getOutputTerminal("failure");
     // Propgate message to failure terminal if it is attached
     if ( failureTerminal.isAttached())
     {
          failureTerminal.propagate(messageAssembly);
     }
     // else throw userException
     else
     {
         // Throw User Exception
          throw new MbUserException(ex.getClassName(),
ex.getMethodName(),
"WMQIv210", "2951", ex.getTraceText(), ex.getInserts());
     }
}
```

This concludes our plug-in source code in Java. Next I will explain how to build and install this plug-in.

BUILDING A PLUG-IN

To build a plug-in jar file you should have JDK installed on your machine. You can download JDK from *http://java.sun.com*.

In this section I will explain how to build your plug-in on a Windows system. An equivalent command needs to be executed on other platforms.

Installing and configuring JDK

Please follow the installation instructions specified on the Web site to install and configure JDK on your workstation. You should also modify your CLASSPATH to include the *jplugin.jar* file. This file is located in:

- *<mqsi_root>\classes* directory on Windows.
- *<mqsi_root>/classes* directory on Unix.

The following command shows how to set up this path in a Windows environment:

```
C:\>SET CLASSPATH=%CLASSPATH%;<mqsi_root>\classes
```

Creating a directory structure and building a jar file

First you should create a directory structure similar to the package of the plug-in on your machine. Let's say you want to store your plug-in source in the *C:\plugin* directory. Now run the following commands to create the directory and build the plug-in jar file.

```
C:\>md plugin\com\ingale\wmqi
C:\>cd plugin\com\ingale\wmqi
C:\ plugin\com\ingale\wmqi>dir
 Volume in drive C is E866FD10
 Volume Serial Number is 182D-44E1
 Directory of C:\java\com\ingale\wmqi
08/26/2002  10:44a      <DIR>          .
08/26/2002  10:44a      <DIR>          ..
08/23/2002  10:09a               4,023 MsgFlowEnvInfo.java
               1 File(s)          4,023 bytes
               2 Dir(s)   2,772,616,192 bytes free
C:\plugin\com\ingale\wmqi>javac MsgFlowEnvInfo.java
C:\plugin>jar -cvf MsgFlowEnvInfo.jar
com\ingale\wmqi\MsgFlowEnvInfo.class
added manifest
adding: com/ingale/wmqi/MsgFlowEnvInfo.class(in = 3127) (out=
1460)(deflated 53%)
```

Installing a plug-in – Defining the node in the configuration repository

Only user-IDs who are members of an *mqbrdevt* group can define a plug-in node in the configuration repository. Alternatively, you could use super-user *IBMMQSI2*. You will have to copy the two files listed below into their respective locations. These steps need

to be performed only once.

- Copy *MsgFlowEnvInfo* to *<mqsi_root>\tool\repository\private\<hostname>\<queue_manager>\MessageProcessingNodeType.*
- Copy *MsgFlowEnvInfo.wdp* to *<mqsi_root>\tool\repository\private\<hostname>\<queue_manager>\MessageProcessingNodeType.*

<hostname> is the name of the server hosting the configuration manager or IP address of the server and *<queue_manager>* is your configuration manager's queue manager name.

The following files need to be copied onto each user's workstation:

- Copy *MsgFlowEnvInfo.properties* to *<mqsi_root>\tool\com\ingale\wmqi*.
- Copy *MsgFlowEnvInfo*.gif* to *<mqsi_root>\tool\images*.
- Copy *MessageProcessingNodeType_MsgFlowEnvInfo.htm* to *<mqsi_root>\tool\help\en_US\com\ingale\wmqi*.

Having copied the above files into their appropriate locations, the next step is to define the node in the configuration repository by executing the following steps:

- Step one: restart your Control Centre.
- Step two: under Message Flow view, right-click on IBM Primitives and select Add To Workspace->Message Flows.
- Step three: select MsgFlowEnvInfo from the displayed dialogue box and click on the Finish button. This will add the plug-in node to your tree view with a 'new' icon in front of it.
- Step four: right-click on the plug-in node and select check-in. This action will result in the removal of files stored in the *<mqsi_root>\tool\repository\private\<hostname>\<queue_manager>\~MessageProcessingNodeType* directory. These files are now stored in the configuration repository.
- Step five: restart your broker service.

Now the plug-in node is available for our use.

Sample output

The code listed immediately below details part of the Environment Tree that is created by the *MsgFlowEnvInfo* plug-in with runtime information for this message flow. Retrieved information is stored in the compound element *WMQIMsgFlowEnvInfo* in the Global Environment Tree. The structure of this information is as follows:

```
(0x1000000)WMQIMsgFlowEnvInfo = (
  (0x3000000)BrokerName                  = 'QABK01'
  (0x3000000)QueueManagerName            = 'QABK1QM'
  (0x3000000)ExecutionGroupName          = 'default'
  (0x3000000)MessageFlowName             = 'RQST_Xform'
  (0x3000000)DataSourceUserId            = 'db2i1'
  (0x3000000)CoordinatedTransactionMode = 'False'
)
```

CASE STUDY: USING MSGFLOWENVINFO PLUG-IN

With the help of an example I will try to explain how this plug-in can be used in a typical reqeust/reply scenario to set the *MQMD.ReplyToQMgr* field at runtime.

Business problem definition

Figure 2 illustrates a typical business scenario where a broker will carry out the transformation of asynchronous request/reply messages from XML to CWF and *vice versa*. Here, application A is a requester application and application B is acting as a responder.

There are channels defined between the NTQM^_BKQM^_MFQM queue managers. The requesting application puts a request message on a remote queue on the NTQM queue manager. Local definition of this request queue (*RQST.IN*) exists on BKQM. The request message flow will read the input message in XML format from this queue. The message will be then transformed into CWF format by *Request Msg Flow*. Since BKQM will also handle reply messages we need to specify this queue manager name in the *MQMD.ReplyToQMgr* field. Application B running on OS/390 will redirect reply messages to the reply queue on BKQM. Using *Reply Message Flow*, the broker will do the reverse transformation

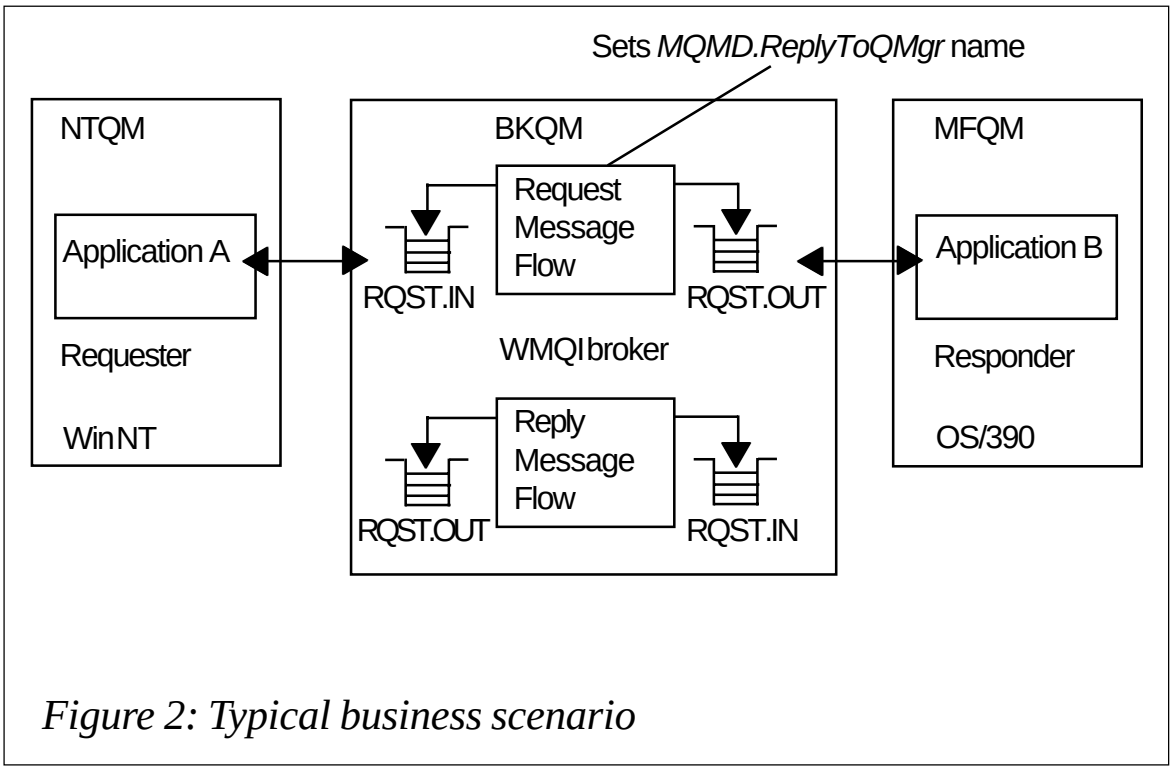

Figure 2: Typical business scenario

(CWF to XML) for reply messages before presenting reply messages to application A on the NTQM queue manager.

We will use the *MsgFlowEnvInfo* plug-in to retrieve the broker's queue manager's name at runtime and assign this value to the *MQMD.ReplyToQMgr* field.

Using the MsgFlowEnvInfo plug-in to solve the problem

The message flow illustrated in Figure 3 illustrates a simple transformation request message flow. By using the *MsgFlowEnvInfo* plug-in we are avoiding hard-coding of the broker's queue manager name. In Figure 3:

- The *RQST.IN* node reads the input message from the *RQST.IN* queue in XML format.
- The *MsgFlowEnvInfo1* node retrieves environment information for this message flow and stores it in the Environment Tree.
- *GetEnvFailed* is a throw node, which throws an exception in the case of errors.

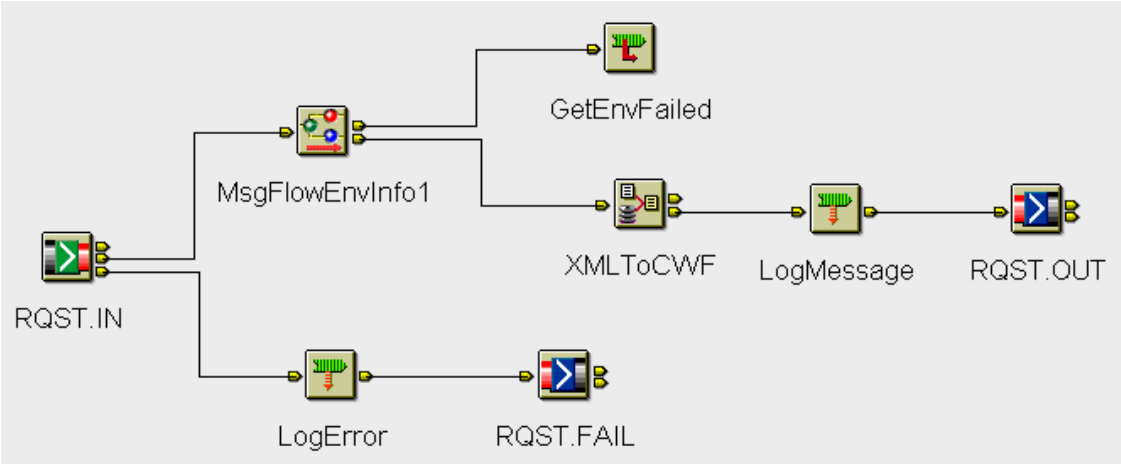

Figure 3: A simple transformation request message flow

- *XMLToCWF* is a compute node that carries out transformation of messages from XML to CWF format. This node will also set *MQMD.ReplyToQMgr* and *MQMD.ReplyToQ* names using the following ESQL:

```
/* Mapping ESQL code goes here */
….
/* Set Reply To Queue name */
SET OutputRoot.MQMD.ReplyToQ = 'RPLY.IN';
/* Set Reply To Queue Manager name */
SET OutputRoot.MQMD.ReplyToQMgr =
Environment.WMQIMsgFlowEnvInfo.QueueManagerName;
```

- *LogMessage* is a trace node to log request messages to the Environment Tree.

```
${Environment}
```

- *RQST.OUT* is an MQOutput node to write transformed request messages to the request queue on the MFQM queue manager.
- *LogError* is a trace node to log the ExceptionList.
- *RQST.FAIL* is an MQOutput node to write failed messages.

SOURCE CODE DOWNLOAD

The following files for this plug-in are available for download from www.xephon.com/extras/wmqi.txt.

- *MsgFlowEnvInfo.java* – Java source code.
- *MsgFlowEnvInfo.jar* – plug-in jar file.
- *MsgFlowEnvInfo.wdp* – WebDav file.
- *MsgFlowEnvInfo XML* – configuration file.
- *MsgFlowEnvInfo.gif* – image file.
- *MsgFlowEnvInfo.properties* – properties file.
- *MessageProcessingNodeType_MsgFlowEnvInfo.htm* – HTML help file.

REFERENCES

WMQI 2.1 Programming Guide.

Kiran Ingale, EAI Architect
Aviana Global Technologies (USA)

Another better MQSeries batch trigger monitor

INTRODUCTION

In the October 2002 issue of *MQ Update* (issue 40) there was an article by Bruce Borchardt (OS/390 Systems Coordinator at Kohls Department Stores, USA) about a better MQSeries batch trigger monitor.

At our site, too, we had been struggling with the sample batch trigger monitor that IBM supplies through SupportPac MA12 and we had decided to develop a new and, hopefully, better and more versatile replacement. However, we took quite a different approach from that in Bruce's organization, as this article will demonstrate.

OUR APPROACH

We identified the following requirements:

- For maximum flexibility we decided that we did not want to store any JCL directly into our process definitions. Instead, the process definitions should contain just a reference to the JCL, with the JCL stored elsewhere.
- We also decided that we wanted to implement a JCL generation process that would be as flexible as possible. For example, we wanted to provide a way to include JCL statements, depending on the project code assigned to the job.
- Finally, since we already had a number of batch jobs triggered through the sample batch trigger monitor we wanted to ensure that the original ('old-format') process definitions would still be supported. In this way, we could immediately start using our new batch trigger monitor and gradually replace the process definitions as we saw fit.

IMPLEMENTATION DECISIONS

The above considerations led us to the following decisions:

- The process definitions should contain just the name of a JCL INCLUDE member (which is taken from our standard JCL INCLUDE library).
- The ISPF file-tailoring services were perfectly suited to provide the flexibility that we needed for the JCL generation process.

The result of these decisions was the program MQSCBTMA, which can be found at www.xephon.com/extras/batchtrigger.txt.

MQSCBTMA

As any MQSeries batch trigger monitor will need to do, it will connect to a given queue manager, open a given batch initiation queue, and go into a GET wait loop on the queue. Whenever it is woken up by a trigger message it will generate the JCL for a batch

job, which it subsequently submits to the internal reader.

THE PARM STRING

The PARM string for the program should have the following format:

```
PARM='[qmgr],initq[,[skeleton][,[jobname][,[notify]]]]'
```

All parameter values except 'initq' are optional. The meanings of the parameter values are detailed below.

- *qmgr* – this is the name of the queue manager to which the program should connect. If this value is omitted the program will connect to its default queue manager (if available).
- *initq* – this is the name of the batch initiation queue on which the program will listen to trigger messages. This parameter is required.
- *skeleton* – this is the name of the ISPF skeleton member from which the program will generate the JCL for the batch jobs that it will submit. The default ISPF skeleton name is MQSCBTMA.
- *jobname* – this is the job name that the program will use for the batch jobs that it generates from 'old-format' process definitions.

  Just like the original sample program, the program will use only the first six positions of the job name specified here; the seventh and eighth positions will be replaced by a sequence number (from 00 through 99 and then back to 00). The default job name is MQSCBT00.

  Note that the program will use this parameter value only for 'old-format' process definitions.
- *notify* – this is the default user name that the program will issue on the NOTIFY parameter of the JOB cards that it generates.

  By default, the program will not issue a NOTIFY parameter except for process definitions that explicitly specify one.

  Note that this parameter value can be overridden only on 'new-format' process definitions.

THE PROCESS DEFINITIONS

The program supports two formats for its process definitions:

- The 'old' format is compatible with the format that is used by the sample batch trigger monitor:
  - the Application-ID contains the EXEC card
  - the user data contains up to four additional JCL cards, limited to 32 characters each
  - the environment data contains up to four final JCL cards, also limited to 32 characters each
  - any of these nine JCL cards may contain a '!' character, which will be replaced with the name of the queue that caused the trigger message to be generated (ie the MQTM-QNAME value).
- The 'new' format is much simpler:
  - the Application-ID has the format:

    ```
    jclincl[,[jobname][,[notify]]]
    ```

  - where *jclincl* is the name of the JCL INCLUDE member. This value is required.
  - *jobname* is the job name that the program will generate for the batch jobs that it generates from the process definition.

    By default, the job name will be equal to the JCL INCLUDE member name.
  - *notify* is the user name that the program will issue on the NOTIFY parameter of the JOB card. By default, the program will specify the user name that was passed to it through its PARM string; if the user name was omitted on the PARM string, too, then no NOTIFY parameter will be generated.
- The user data and the environment data will not be used by the program.

JCLINCL	JCL INCLUDE member name (new-format process definitions only).
JCLLINE1	EXEC card, taken from the Application-ID of the process definition and with the first occurrence of the '!' character replaced with the triggering queue name (for old-format process definitions only).
JCLLINE2	JCL card from positions 1 to 32 of the user data, after substitution of the '!' character, as above (for old-format process definitions only). For new-format process definitions the variable will contain the text from said positions without any substitution.
JCLLINE3	JCL card from positions 33 to 64 of the user data, after substitution of the '!' character as above.
JCLLINE4	JCL card from positions 65 to 96 of the user data, after substitution of the '!' character, as above.
JCLLINE5	JCL card from positions 97 to 128 of the user data, after substitution of the '!' character as above.
JCLLINE6	JCL card from positions 1 to 32 of the environment data, after substitution of the '!' character as above.
JCLLINE7	JCL card from positions 33 to 64 of the environment data after substitution of the '!' character as above.
JCLLINE8	JCL card from positions 65 to 96 of the environment data after substitution of the '!' character as above.
JCLLINE9	JCL card from positions 97 to 128 of the environment data after substitution of the '!' character as above.
JOBNAME	The job name that must be assigned to the batch job.
MQPROC	The name of the MQSeries process definition
NOTIFY	The user name that must be generated on the NOTIFY parameter of the JOB card. Blank if no NOTIFY parameter need be generated.
PROGRAM	The name of the program, ie MQSCBTMA.
PROJECT	A three-character project code taken from positions 1 to 3 of the INCLUDE member name (for new-format process definitions). For old-format process definitions the project code will be taken from positions 1 to 3 of the triggered queue name, provided that its fourth position is a period; otherwise, the project code will not be filled in.
QMGR	The local queue manager name.
QNAME	The triggered queue name.
SKELETON	The ISPF skeleton member name.

Table 1: ISPF dialogue variables

ISPF DIALOGUE VARIABLES

The program defines the ISPF dialogue variables given in Table 1.

AN EXAMPLE ISPF SKELETON

Following Table 1 is a simple ISPF skeleton that the program may use to generate the JCL for its batch jobs.

```
//&JOBNAME JOB  +++jobcard parameters+++<,NOTIFY=&NOTIFY|>
// JCLLIB ORDER=+++jcllib libraries+++
//* Job generated from skeleton &SKELETON by program &PROGRAM..
//* Queue manager  . . . . : &QMGR..
//* Process name . . . . . : &MQPROC..
//* Triggering queue . . . : &QNAME..
// SET   QMGR='&QMGR'
// SET MQPROC='&MQPROC'
// SET  QNAME='&QNAME'
)SEL &JCLINCL = &Z
&JCLLINE1
&JCLLINE2
&JCLLINE3
&JCLLINE4
&JCLLINE5
&JCLLINE6
&JCLLINE7
&JCLLINE8
&JCLLINE9
)ENDSEL
)SEL &JCLINCL = &Z
// INCLUDE MEMBER=&JCLINCL
)ENDSEL
```

STOPPING THE PROGRAM

To stop the batch trigger monitor you will have to send a 'REPORT' message with a feedback code of MQFB-QUIT to its initiation queue. You could use the CKTIEND sample program (from the MA12 SupportPac) or, alternatively, the MQSCPBTM program that I provided for this purpose.

A FEW NOTES ABOUT MQSCBTMA

- The program makes heavy use of nested COBOL subprograms. Without them I find it virtually impossible to keep some form of

structure in a COBOL program.

- As required as per our production standards the program will call the MQSeries API dynamically, instead of statically (which is how all IBM's sample programs seem to link to the MQSeries API).

  To support dynamic linking to MQSeries we provided a COBOL COPY member, MQIBATCH, which defines all dynamic MQSeries API entry points. (See the *MQSeries Application Programming Guide, Document Number SC33-0807-12, §3.7.2: Dynamically calling the MQSeries stub*, for information about this feature.)

- If you do not specify the queue manager on the PARM string that you pass to the program it will query the queue manager for its name (through an MQINQ API call).

A suggested feature that we did not implement

A suggestion was made that we implement an (optional) sequence number for the new-format process definitions (similar to the global sequence number that is used for the job names of old-format process definitions). This feature would work as follows:

- If the jobname parameter of a process definition was specified and if its seventh and eighth positions were numeric, the last two positions would be used for a sequence number.
- Whenever a batch job was submitted from such a process definition the sequence number would be incremented and the process definition (or at least its Application-ID) would be rewritten, with the new sequence number in the last two positions of the job name.

Note that there is no MQSeries API call available to modify a process definition. Instead, an **ALTER PROCESS** command must be issued to the system command input queue, after which the reply must be retrieved from a suitably defined reply queue.

Even though in the end we decided against implementing this feature I did write the 'CBLTMQAP' program as a proof of concept;

furthermore, the 'CBLTMQOQ' program can be used to create a suitable (dynamic) command reply queue based on the standard command input model queue. You can consult the comment blocks at the start of the program sources to find out how these programs work.

The included files, which can all be found at www.xephon.com/extras/batchtrigger.txt, are as follows:

- *MQSCBTMA.COB* – the main program source file.
- *MQSCPBTM.COB* – the source of a program to send the QUIT message.
- *MQIBATCH.CPY* – the COBOL COPY member that defines the dynamic batch MQSeries entry points.
- *MQSXCKTI.PROC* – a sample JCL procedure for the batch trigger monitor-started task.
- *MQSCBTMA.SKEL* – a very simple ISPF skeleton for the program to use.
- *CBLTMQAP.COB* – a sample program that will issue an **ALTER PROCESS** command and retrieve the reply.
- *CBLTMQOQ.COB* – a sample program that will open a queue and that can be used to create a queue based on a model. (I used this program to create a command reply queue for the *CBLTMQAP* program to use, based on the standard command reply model queue.)

Luc Van Rompaey, System Engineer
Telepolis Antwerpen (Belgium)

Clearing temporary files in WMQ for AS/400 V5.*x*

This Qshell script program will allow you to clear out temporary files that are created during the normal operation of WMQ for AS/400 versions 5.1 and later.

When an MQSeries channel runs, it creates an empty temporary file (actually a named pipe) in the */tmp* IFS directory. The pipe is used to allow MQ commands to interact with the channel job. The format of the file name is *MQSeries.nnnnnn,* where *nnnnnn* is the Process-ID of the channel job in question.

CLEARING THE FILES

In the normal course of events these files are only deleted when you issue the command which ends all queue managers and all jobs connected to those queue managers:

```
ENDMQM MQMNAME(*ALL) ENDCCTJOB(*YES)
```

At many MQ installations this command is run infrequently (if at all), which can lead to a build up of these temporary files. Because the files are empty they do not use significant disk space but they do waste inodes in the IFS and so should be deleted periodically as part of good system management.

The problem with deleting these files is that they must not be deleted if they are in use by MQSeries. To guard against this the provided script checks whether MQSeries is using the file before it is deleted. The script works by listing all of the *MQSeries.nnnnnn* files in */tmp* and then checking whether the associated channel job is still active by calling the iSeries specific **getjobid** QShell command. If the channel job is no longer in the system the file can be safely deleted.

To set up and run the script:

- Copy it into a file called ‘clrmqtmp.sh’ on your PC and ftp it to an IFS directory on the iSeries or create the file directly on iSeries using EDTF.
- Start the Qshell with the **QSH** command.
- Make the shell script executable with the command:

```
chmod 755 /yourpath/clrmqtmp.sh
```

- Run the shell script:

```
/yourpath/clrmqtmp.sh
```

You will see the following output:

```
   =  Clearing out unused MQ files in /tmp/    =
   Deleting file /tmp/MQSeries.2132
   Deleting file /tmp/MQSeries.2138
   Deleting file /tmp/MQSeries.3114
   Deleting file /tmp/MQSeries.3321
   Process identifier 5418 is 614792/M8PHILLI/RUNMQCHI -job alive -
file
   /tmp/MQSeries.5418 not deleted
   Process identifier 5613 is 615987/MJCAMP/RUNMQCHI -job alive - file
/
   tmp/MQSeries.5613 not deleted
   =     Unused files have been deleted        =
```

For an overview of the QShell please refer to *http://www.ibm.com/servers/eserver/iseries/whpapr/qshell_overview.html*.

The script follows. Depending on the CCSID of the job you are using when you view this file, the '$' character (as in $i) may be shown as '£'. This is normal.

CLRMQTMP.SH

```
#*  Program:    clrmqtmp.sh                                          *
#*  Author :    Mark Phillips                                        *
#*  Function:   Qshell script to clear out temporary files in        *
#*              /tmp.  The files are first checked to ensure that    *
#*              they are no longer needed by MQSeries                *
#*  Parameters: None                                                 *
echo "============================================="
echo "=  Clearing out unused MQ files in /tmp/    ="
echo "============================================="
 find /tmp -name "MQSeries.*" -print > /tmp/clrmq1.tmp 2>/dev/null
 cat /tmp/clrmq1.tmp | while read FILE ; do
   if  getjobid `echo $FILE |cut -d. -f 2` > /tmp/clrmq2.tmp 2>&1 ;
   then
     echo `cat /tmp/clrmq2.tmp` -job alive - file $FILE not deleted  ;
   else
     echo Deleting file $FILE; rm $FILE;
    fi
 done
rm clrmq1.tmp > /dev/null 2>&1
rm clrmq2.tmp > /dev/null 2>&1
echo "============================================="
echo "=     Unused files have been deleted        ="
echo "============================================="
#*                          End Program                           *
```


Authentication and authorization for JMS

INTRODUCTION

JMS is a programming interface defined by Sun as a vendor-neutral way for Java programs to access point-to-point and publish-subscribe messaging functions. While Sun has specified the API, the underlying implementation is left to a JMS provider and there are no requirements for interoperability between different providers. JMS in turn is part of the J2EE set of standards, the most recent level of which requires an application server to include a fully-functioning JMS component.

This article describes how some of the security aspects of the JMS programming interface are mapped to the services and APIs provided by WMQ. Many of the security-related functions of a queue manager are not influenced by the application program and I do not intend to repeat information available elsewhere on those. Here I will be writing about identification, authentication, and authorization, as these are areas which are not yet truly fixed in a Java standard but for which implementation decisions have been made, based on the use of WMQ facilities.

There are other JMS providers but I am going to be concerned here only with the WMQ-based service included in SupportPac MA88 (and shipped as part of WMQ V5.3), and with the J2EE environment implemented by the WebSphere Application Server (WAS).

BASIC STRUCTURE

WMQ's implementation of JMS is a layered approach, which builds upon other components: the Java classes (which are an OO form of the MQI), which in turn will use either a JNI interface to the C libraries, which connect applications to the queue manager, or a Java-only path for client connections.

Figure 1 shows the common routes for JMS applications connecting

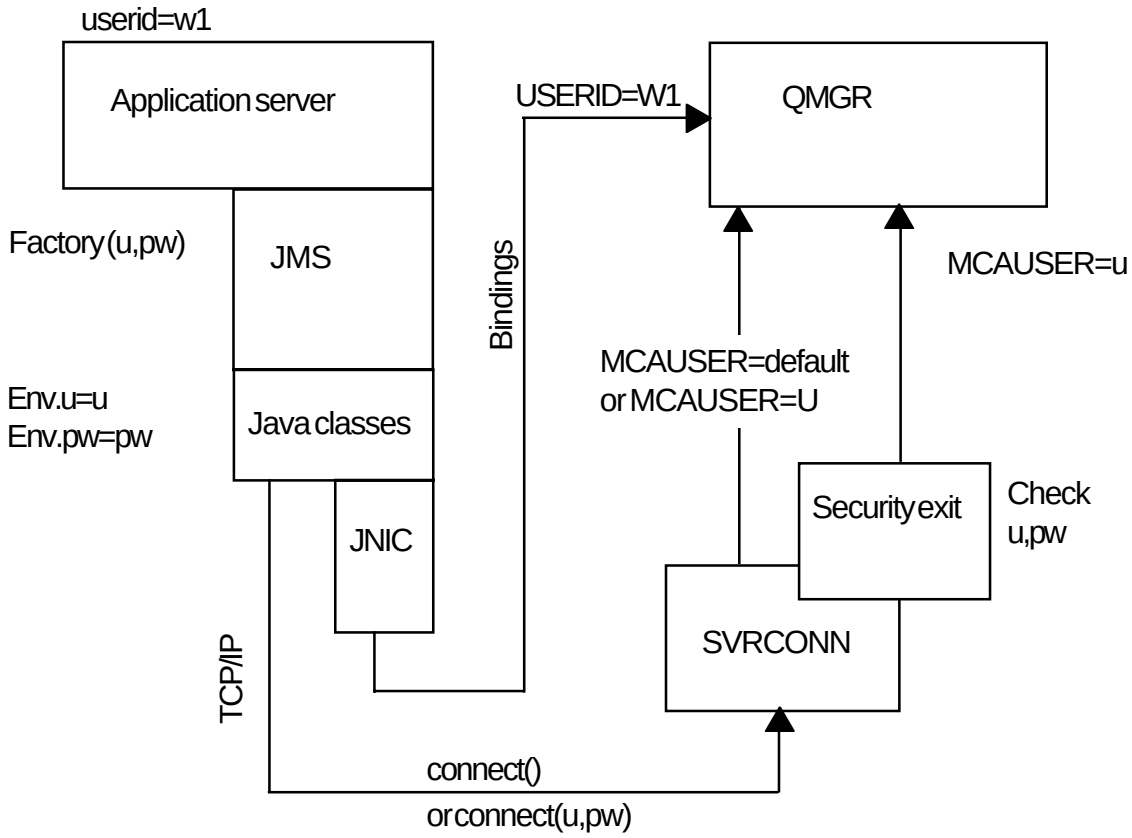

Figure 1: Common routes for JMS applications connecting to a queue manager

to a queue manager.

As Figure 1 illustrates, the JMS functions here are running inside a copy of the WebSphere Application Server program (or an alternative view might be the JVM that executes the WAS classes). As we need to look at identification I'm going to say that this copy of WAS is being executed by someone who has logged into the operating system with user-ID 'w1'. A JMS application does not have to be running within an application server, but many do.

THE CREATEQUEUECONNECTION METHOD

A JMS application must connect to the messaging service provider. Because each JMS provider might have its own format for defining

the route to the service an abstraction is used, where the configuration parameters are actually stored in a directory. A ConnectionFactory object is then populated with the provider-specific information and the application code then calls a method to drive the real connection.

There is one method for each of the point-to-point (createQueueConnection) and the publish/subscribe (createTopicConnection) subsets of JMS, with additional methods for the XA (global transaction) variants of these. As far as this article is concerned all four methods can be considered equivalent, so I'll abbreviate them to connect() for now.

JMS defines two forms of these connect() methods, one with no additional parameters and one where a user-ID and password can be passed. These variants will behave differently depending on whether bindings or client mode is used for the real connection to a queue manager.

CONNECTING IN BINDINGS MODE

An application can connect to WMQ in two different ways. The first of these is called bindings mode and is only available when the application and the queue manager are on the same machine. A local platform-specific inter-process communications mechanism is used for the application and queue manager to interact. C or C++ applications that are linked with *libmqm* or *mqm.dll* will be using bindings mode; for JMS applications the choice is made by setting the TransportType for the ConnectionFactory. This can be set programmatically to the value MQJMS_TP_BINDINGS_MQ or as the attribute TRANSPORT(BIND) in the JMSAdmin interface.

If the application issues the form of the connect() method with no parameters, the queue manager will automatically discover the application program's user-ID and use that for all future authorization checks.

If the application issues the form of the connect() method with user-ID and password parameters, the behaviour of the queue manager depends on the version of the Java classes being used.

Versions of the MA88 SupportPac prior to V5.2.1 will always throw an exception if the user-ID is specified in a bindings mode connection. In MA88 V5.2.1 and also in the code shipped with WMQ V5.3 this was relaxed so that it will reject the connection request only if the user-ID value is not the same as that under which the application program is running. The password parameter is always ignored and not passed through any additional processing.

The WMQ development organization is considering ways to allow the user-ID and password to be validated even in a bindings mode connection, but this is not currently available. Today's behaviour, with no real checking of user-IDs in bindings mode, is exactly the same as that available for C applications, where authentication of local users is assumed to have been already performed by the operating system. In a WAS environment such as that shown in Figure 1, the user-ID for a bindings connection will always, therefore, be w1.

If you have sufficient authority an application using the MQI or the base Java classes can override the default user-ID used for authorization and for insertion in the MQMD. However, the alternative user-ID and context manipulation facilities are not part of the JMS specification and, therefore, cannot be used; in other words, the JMSXUser-ID property cannot be set by an application.

If you want a message from a JMS application to have a different user-ID from the default (w1 in the example) when received on another queue manager, you should consider either writing a simple C 'server' application, which takes the inbound message, transforms it, and puts it to the real destination queue, or, alternatively, using a channel message exit, which can directly modify the contents of the MQMD.

CONNECTING AS A WMQ CLIENT

To connect as an WMQ client set the TransportType of the ConnectionFactory in your program to be *MQJMS_TP_CLIENT_MQ_TCPIP* or set the TRANSPORT(CLIENT) value in the JMSAdmin tool. This is equivalent to C or C++ applications linking with *libmqic* or *mqic32.dll*. Note that JMS applications can connect

only using the TCP/IP network protocol; access to LU6.2 and other protocols is available only to C or C++ client programs.

If the application uses the form of the connect() method with user-ID and password parameters, these are passed to the SVRCONN channel and can be tested in a security exit. The security exit can also set the MCAUSER attribute for the channel, which will be used for all future authorization operations made for that connection. If there is no security exit configured on the SVRCONN channel and the MCAUSER attribute is blank, the asserted user-ID will be used, but there is no authentication using the password.

If the application uses the form of the connect() method with no parameters then no identification information is passed to the SVRCONN channel. This is similar to the behaviour of the original (version 2.*x*) MQSeries C clients, which used environment variables for identification purposes.

It is important to realize that, unlike the current C clients, the Java client does not automatically pass the user-ID of the running application. The WMQ behaviour when there is no MCAUSER defined on the SVRCONN and when no user-ID is passed from a client application says that the authority given to the connection is that of the user-ID running the SVRCONN process itself – and this will normally be the mqm user-ID. This means that applications which do not specify user-IDs and queue managers without security exits will probably result in JMS programs having full WMQ authority.

Some people consider that the difference in behaviour between C and JMS clients, with one passing a local user-ID and the other requiring it to be configured or programmed, is a difference in the available security; some even consider the JMS behaviour to be a security loophole. However, my belief is that all of the mechanisms, whether configuration, program, environment variables, or automatic discovery of a local user-ID, have equivalent strength, as there is typically very little control over client machines and it is easy to create new users with whatever name is desired.

The only way to be certain of who is connecting to a queue manager, no matter what the language of the client program, is to

install security exits (or use the SSL facilities in V5.3) on the SVRCONN channel.

The security exit point is available to JMS client programs with an interface similar to the C definition. If an authentication check more complex than user-ID/password validation is required, you might need to write a client-side security exit in Java that can communicate with a C exit running in the SVRCONN channel. An example Java security exit is now available as SupportPac IC72.

One difference between the z/OS and distributed queue managers is that on z/OS there is very limited (coarse-grained) authorization checking for MQCONN calls. While you can restrict adapters, you cannot restrict individual users of an adapter – if one batch application can connect, all can. In contrast, on the distributed platforms every MQCONN goes through an authorization check for the user-ID.

This also shows up in SVRCONN processing. On z/OS, once a security exit has permitted the connection through to the queue manager, no further authorization check is made until the first MQOPEN. On the distributed platforms, the user-ID passed from the SVRCONN is subject to the same authorization check as if it were a bindings mode application. (The implementation of the MCA causes this pseudo-MQCONN, even though the MCA itself has already passed its own private MQCONN check.) If authorization events have been enabled for the queue manager, then from V5.3 onwards a failed connection from a client will generate an event message in the same way as bindings mode applications have always done.

SSL CLIENTS

With WMQ V5.3 we have added SSL capabilities to all channel types, including JMS clients. There are several properties associated with the ConnectionFactory that define how the SSL processing is to be carried out and which certificates are to be used. When the connection is made from the client to the queue manager the application program has the same connect() method

as previously and can choose either to send a user-ID and password or not. However, at a lower level, the SSL protocol also carries the Distinguished Name (DN) as part of the certificate. WMQ can make a simple authorization decision as to whether or not to allow that DN to establish the channel, but once that check has been passed all the same flows happen for security exits and other processing within the queue manager.

The DN cannot be used directly inside WMQ for any further authorization checks. But it might be appropriate to map the DN to a local user-ID instead of relying on an asserted identify coming from the connect() method. This mapping can be done automatically by RACF but all other environments will require a security exit to be written to override the MCAUSER.

It is important to realize that the use of SSL does not guarantee the validity of the user-ID passed from the client program in the connect() method; all the server can be sure of is that the client program has access to a suitable certificate. Other controls may need to be in place to ascertain who can use that certificate.

AUTHORIZATION FOR CONNECTED APPLICATIONS

Once an application has connected, whether through bindings or client mode, whether from JMS, the base Java classes, or any other language, whether from a hardcoded MCAUSER or set dynamically, the authorization steps are the same. The user-ID associated with the hConn or any construct built on top of it, such as a JMS session, is used for all calls to the authorization component, which might be the WMQ OAM or z/OS SAF. How authorization works on each platform is documented in the WMQ books.

INTEGRATION WITH WAS SECURITY

Normally the user-ID will have to be an entity known to the operating system because that is how both SAF and the OAM work. The user-ID must also match WMQ's naming rules (ie no more than 12 characters, except in certain circumstances on

Windows). However, these rules may not be appropriate inside an application server such as WAS. If you wish to have a more tightly-integrated authorization mechanism you might like to look at the version of WMQ that has been integrated with WAS V5.

For this combination, some additional cooperating exits have been written. One is installed as a SVRCONN security exit; given a user-ID and password from the JMS connect(), it calls a WAS security module for authentication; this user-ID need not conform to WMQ rules as the exit then sets a 'dummy' MCAUSER on the channel, which does conform to the rules. A second module is configured in place of the normal OAM to provide authorization services. When given the dummy user-ID it knows the real user-ID from its cooperation with the channel exit and can then pass the appropriate authorization request to WAS security.

While any messages generated by JMS in this environment will include the dummy user-ID in the MQMD, another channel exit could have been written to map this to something acceptable on other machines as the message is transferred.

SUMMARY

There are some important differences between C and Java application environments for how authentication and authorization are configured and administered. The actual capabilities are identical, however, with the same strength of protection. If you are going to have Java or JMS applications you need to understand these differences to maintain control over who can connect to your queue manager and to restrict what they can do once connected.

Mark Taylor
IBM Hursley (UK)

Contributing to *MQ Update*

In addition to *MQ Update*, the Xephon family of *Update* publications now includes *CICS Update*, *MVS Update*, *TCP/SNA Update*, *DB2 Update*, *AIX Update*, *and RACF Update*. Although the articles published are of a very high standard, the vast majority are not written by professional writers, and we rely heavily on our readers themselves taking the time and trouble to share their experiences with others. Many have discovered that writing an article is not the daunting task that it might appear to be at first glance.

They have found that the effort needed to pass on valuable information to others is more than offset by our generous terms and conditions and the recognition they gain from their fellow professionals. Often, just a few hundred words are sufficient to describe a problem and the steps taken to solve it.

If you have ever experienced any difficulties with MQ, or made an interesting discovery, you could receive a cash payment, a free subscription to any of our *Updates*, or a credit against any of Xephon's wide range of products and services, simply by telling us all about it. For a copy of our *Notes for Contributors*, which explains the terms and conditions under which we publish articles, please point your browser at www.xephon.com/nfc.

MQ news

Reconda has recently launched the latest version of its WebSphere MQ systems management support product QN-AppWatch. The company claims it provides controlled centralized access to application data and communications layers with industrial-strength security, configuration, change, and audit management functionality.

QN-AppWatch apparently combines a common architecture with flexible customization capabilities, standard browser, and single server installation to provide recordkeeping and detailing of an enterprise's MQ environment, promoting the creation and enforcement of enterprise-wide standards from one central environment.

Reconda states that V2.5 incorporates advanced online message editing capability as well as the ability to manipulate and convert message headers.

For more information contact:
Reconda, 15 East Putnam Avenue, Suite 306, Greenwich, CT 06830, USA.
Tel: (203) 299 4000.
Fax: (203) 299 4095.
Web: www.reconda.com

* * *

Nastel Technologies has unveiled a JMX (Java Management Extensions) API for WebSphere MQ Everyplace Mobile Messaging software. The new API, which Nastel created in conjunction with IBM, will ship with WMQ Everyplace.

Nastel claims its solution gives users a single point for controlling, monitoring, and automating WMQ Everyplace functions across platforms, in both connected and wireless networks. The new API component is being designed in Java and utilizes JMX to optimize efficiency.

For more information contact:
Nastel Technologies, 48 South Service Road, Melville, New York 11747, USA.
Tel: (631) 761 9100.
Fax: (631) 761 9101.
Web: www.nastel.com

Nastel Technologies (UK), 3 Tannery House Tannery Lane, Send, Surrey, GU23 7EF UK.
Tel: + 44 207 872 5412.
Fax: + 44 207 753 2848.

* * *