39

# MQ update

*September 2002*

## In this issue

3 Configuring a Web Client on Windows NT using IIS PWS Web Server: part two – testing

12 Heartbeat: channel monitoring tool for MQ clusters

17 WebSphere MQSeries Integrator cross-reference

27 MQSI exception processing: request/reply messages: part two – subflows

37 Maximizing message availability

48 MQ news

# MQ Update

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

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

**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.

## Configuring a Web Client on Windows NT using IIS PWS Web Server: part two – testing

Continuing last month's article on configuring a Web Client on Windows NT using PWS Web Server, we conclude here with a test example.

### TESTING WITH WEB CREDIT EXAMPLE

Please note that from here on, <*MQWFDir*> should be replaced with the actual MQSeries Workflow root directory (eg *C:\Program Files\MQSeries Workflow*).

If you are not using the default configuration (as in this case) you must modify the FDL (<*MQWFDir*>*\scenario\WebCredit\WebCredit.fdl*) to direct the user-defined Program Execution Server to the queue manager and queue that you selected, and update the *CustomerUPES.propertie*s file to reflect your environment.

Make sure your Web server (or the user ID that the Web server uses) has connect authorization to DB2, otherwise the JSPs cannot read data from DB2 (this includes adding the file *db2java.zip* to the Web server's CLASSPATH).

- Download the WebCredit example from *http://www-4.ibm.com/software/ts/mqseries/txppacs/wa82.html* to a temporary folder, eg TMP.
- Unzip the file *wa82.zip* and copy the contents under the directory *scenerio\WebCredit\...* to *C:\Program Files\MQSeries Workflow\scenario directory*.
- Create the customer database:
  - open the DB2 Command Centre
  - copy the contents of the file <*MQWFDir*>*\scenario\WebCredit\CreateDBx.sql* into the Command Centre (where *x* is '6' for DB2 6.1 and '7' for DB2 7.1)
  - press Ctrl+Enter in the Command Centre. This executes all the

commands in the SQL file and creates the database as well as the two tables in the database

– close the DB2 Command Centre.

- Copy the HTML/JSP files to the Web Client directories. From the scenario directory copy the contents of the starter directory to the Web Client's *webpages\starter* directory and the contents of the program's directory to the Web Client's *webpages\programs* directory.

  If the Web Client that comes with MQSeries Workflow 3.3 is used the destination root directory has to be changed to *<MQWFDir>\cfgs\fmc1\WebClient\webpages,* where *fmc1* is the configuration ID under which the Web Client has been installed.

- Modify the external process start pages. If you are using the Web Client that ships with MQSeries Workflow 3.3, the pages that start the process have to be modified. The Web Client in MQSeries Workflow 3.3 is usually configured to include the workflow configuration ID in its URL (eg */MQWFClient-FMC*). This means that all occurrences of */MQWFClient/* have to be replaced with */MQWFClient-<CFGID>/,* where *<CFGID>* is the ID under which the Web Client has been configured, eg *FMC1*.

  The following files in the starter directory have to be modified (look for the *<form>* tag):

  – *WebCreditRequest.html*

  – *WebCreditRequest_NewCustomer.html*

  – *WebCreditRequest_IDDoesNotExist.html*

  – *WebCreditRequest_ExistingCustomer.jsp*

  – all other JSPs in the folder.

- Set-up for the Java-based UPES.

  In the following steps *<FMCQM>* is used as the name of Workflow's queue manager. Please substitute your actual name – *FMCQM1* – whenever you see this placeholder.

  To create the queue for the user-defined PES:

  – Open the MQSeries Explorer.

  - Make sure your queue manager is running (default: <*FMCQM*>).
  - Select *Console Root\IBM MQSeries\Queue Managers\<FMCQM>\Queues.*
  - In the tree on the left, right-click Queues and select New/Local Queue.
  - For queue name enter WEBCREDIT (in uppercase) and click OK.
  - In the dialog box that appears click Share in Cluster.
  - In the dialog box that appears select FMCGRP in the Cluster combo box and click OK.
  - Optionally, setup triggering for the UPES (it starts the UPES automatically when a new message arrives – see *Triggering*).
  - Close the MQSeries Explorer.

- Update the batch files for the CustomerUPES.
  - Find the batch files in *<MQWFDir>\scenario\WebCredit\CustomerUPES.*
  - Update the file *env.bat* to reflect your environment:
    - *JAVA_HOME* – the top-level directory of the JDK

      ```
      REM **set JAVA_HOME=d:\Java\IBMJDK13
      set JAVA_HOME=c:\Jdk1.3.1
      ```

    - *XERCES_HOME* – the directory where *xerces.jar* (XML parser for Java) is stored

      ```
      REM **set XERCES_HOME=d:\fmcwinnt\smp\dpxml
      set XERCES_HOME=C:\Program Files\MQSeries
           Workflow\SMP\Dpxml
      ```

    - *DB2_HOME* – the top-level directory of the DB2 installation

      ```
      REM  **set DB2_HOME=d:\sqllib
      set DB2_HOME=C:\Sqllib
      ```

    - *MQ_HOME* – the top-level directory of the MQSeries installation

```
REM  **set MQ_HOME=d:\Programs\MQSeries
set MQ_HOME=C:\Progra~1\MQSeries
```

- *JAMES_HOME* – the top-level directory of the Java Apache Mail Enterprise Server (this is optional).

- Make sure that you have already imported the FDL file that contains the STARTER user ID (*<MQWFDir>\scenario\credit\starter\starter.fdl*), which is contained in the MQSeries Workflow Web Client.

```
fmcibie -uADMIN -ppassword -o -t -i
<MQWFDir>\scenario\credit\starter\starter.fdl  -y  FMC1
```

If you get any errors try the following:

```
cd <MQWFDir>\scenario\credit\starter\
fmcibie -uADMIN -ppassword -o -t -i starter.fdl  -y  FMC1
```

- Remember to mention the configuration ID – in this case *FMC1* – if it is not set as the default.

- Make sure that you have configured the Web Client *<MQWFDir>\cfgs\<CFGID>\WebClient\WebClient.properties* for the MQWF 3.3 version of the Web Client, to allow anonymous process starts using this STARTER user-ID.

Check the following corrections that need to be made to the *WebClient.Properties* file.

```
#Logfile=e:/fmcwinnt/cfgs/fmc/log/servlet.log
Logfile=C:/Program Files/MQSeries Workflow/cfgs/fmc1/log/servlet.log
Make JSP viewer as Default Viewer
# JSPViewer uses JSPs instead of HTML template files
#DefaultViewer=com.ibm.workflow.servlet.client.JSPViewer
DefaultViewer=com.ibm.workflow.servlet.client.JSPViewer
```

Correct:

```
#StarterUserID=STARTER
#StarterPassword=password
#StarterSystemGroup=FMCGRP
#StarterSystem=FMCSYS
StarterUserID=STARTER
StarterPassword=password
StarterSystemGroup=FMCGRP1
StarterSystem=FMCSYS1
```

Uncomment:

```
#EnableTemplatelists=true
#EnableInstancelists=true
#EnableWorklists=true
EnableTemplatelists=true
EnableInstancelists=true
EnableWorklists=true
```

Uncomment:

```
     #AutoRefresh=false
     AutoRefresh=false
     # MQWF runtime database name, user ID and password of the DB2
user.
     #Database   = FMCDB
     #DB2User    = db2admin
     #DB2Password= db2admin
     Database   = FMCDB1
     DB2User    = db2admin
     DB2Password= db2admin
```

- To modify *C:\jakarta-tomcat-3.2.3\bin\tomcat.bat* add the following line at the beginning of the file:

```
set JAVA_HOME=c:\jdk1.3.1
```

After:

```
set CLASSPATH=%CP%
echo Using CLASSPATH: %CP%
echo
```

add:

```
set
CLASSPATH=%CP%;c:\Progra~1\MQSeri~1\bin\Java33ØØ\fmcojagt.jar;c:\
Progra~1\MQSeri~1\bin\fmcohcli.jar
```

- To register the servlet with the servlet container for servlet API 2.2 and later, create a Web Application in your container with root *URI/ MQWFClient* and document root *ConfigurationRootDirectory/cfgs/ ConfigurationID/webpages*.
  - create an alias with *MQWFClient-FMC1* pointing to *C:\Program Files\MQSeries Workflow\cfgs\fmc1\WebClient\ webpage*s.
  - in *C:\jakarta-tomcat-3.2.3\bin\server.xml* add *<Context path=/ MQWFClient-FMC1* docBase=*C:/Program Files/MQSeries*

*Workflow/cfgs/fmc1/WebClient/Webpages* crossContext=*true*, debug=9, reloadable=false, trusted=*false* /> before </ ContextManager> </Server>. (Note that the leading forward slash (/) and long names containing blanks have to be used.)

– in *C:\Program Files\MQSeries Workflow\cfgs\fmc1\WebClient\webpages\web.xml:*

```
@ enable if you want to use a configuration file
    <init-param>
      <param-name>ConfigurationFile</param-name>
<param-value>c:/fmcwinnt/WebClient/WebClient.properties</
param-value>
    </init-param>
@
-->
<init-param>
<param-name>ConfigurationFile</param-name>
<param-value>C:/Program Files/MQSeries Workflow/cfgs/fmc1/
WebClient/WebClient.properties</param-value>
</init-param>
</servlet>
```

- Import the process model into MQSeries Workflow

```
fmcibie -uADMIN -ppassword -o -t -i
<MQWFDir>\scenario\WebCredit\WebCredit_UPES.fdl -y  FMC1
```

## Running the scenario

- Run the batch job **START_MQ_FMC1** to start the queue manager, MQSeries trigger monitor, and MQSeries command server for configuration FMC1. It will also start two DOS windows – do not close them.
- Start the MQ Workflow Server as an NT service:
  - go to Start>Settings>Control Panel>Services>; select MQ Series Workflow 3.3- FMC1; click start; you will see a prompt saying that the service has started.
- Start Tomcat. Go to *C:\jakarta-tomcat-3.2.3\bin* and click on *Startup.bat*. This will pop up another window. Do not close it.
- Start the user-defined PES. This will pop up another window. Do not close it. This step is only necessary if you use the Java-based UPES. Locate the file *CustomerUPES.bat* in the the directory

<*MQWFDir>\scenario\WebCredit\CustomerUPES*.

Start the UPES using the *CustomerUPES.bat* batch file. To demonstrate how MQSeries holds on to a message it's possible to defer starting the UPES until a message has arrived in the queue (eg after the first activity in the process has been executed).

If you want to shut down the Customer UPES you can use the shutdown script that is provided in *<MQWFDir>\scenario\WebCredit*. Since this version of the UPES is implemented with Synchpoint control it is also possible to kill just the UPES (it may take a short while for MQSeries to clean up its resources, however).

- Use the Web browser to start a new process:
  - Point your browser to *http://Localhost/MQWFClient-FMC1/starter/WebCreditRequest.html*.

    If you get a 'Page not found error HTTP 404' it could be possible that the Workflow server is not running. Please check and start the servers using the *fmcamain* utility.

    If everything is fine the browser opens a window requesting a customer ID or select New Customer. Click on New Customer.
  - You will be prompted to enter your details. The example WEBCREDIT is a very basic version so only enter valid information. For example, do not put commas or currency symbols in the amount field.
  - Once the form is submitted you will receive a tracking-ID and the system thanks you for your efforts.
  - Now open another window and point your browser to *http://Localhost/MQWFClient-FMC1/RTC.html*. This prompts for user ID and password. Enter user-ID 'ADMIN', password 'password', and system group 'FMCGRP1'.
  - You can see the application you have sent in the work list of ADMIN, but you cannot really start the work item.
  - Log off ( the button is located in the top right corner) and log on again as WEBBANK_CLERK. Enter user-ID

'WEBBANK_CLERK', password 'password', and system group 'FMCGRP1'.

- Now you can start the work item. Try using the various options available.
- Sometimes, if the risk involved is large, you have to log on as WEBBANK_LOANMANAGER to approve a loan. Enter user-ID 'WEBBANK_LOANMANAGER', password 'password', system group 'FMCGRP1'.

## REGISTERING THE UPES FOR AUTOMATIC START THROUGH MQSERIES (TRIGGERING)

- Open the MQSeries Explorer.
- Open the object tree to *Console Root\IBM MQSeries\Queue Managers\<QMNAME>\Advanced\Process Definitions* where *<QMNAME>* is the name of your queue manager (the default is *FMCQM*).
- Right-click Process Definitions and select New\Process Definition.
- Enter the following values (leave unspecified fields empty):
  - Process Definition Name: *WEBCREDIT.UPES*
  - description: UPES for the WebCredit Example Process
  - application type: Windows NT
  - application identifier: *<MQWFDir>\scenario\WebCredit\CustomerUPES\run_trigger.bat.*
- Press OK.
- Open the object tree to *Console Root\IBM MQSeries\Queue Managers\<QMNAME>\Queues.*
- Right-click the WEBCREDIT queue.
- Select the Triggering tab.
- Enter the following values (leave unspecified fields empty):

  - trigger control: on
  - trigger type: first
  - trigger depth: one
  - trigger message priority: 0
  - initiation queue name: FMCTRIGGER
  - process name: WEBCREDIT.UPES.
- Make sure the batch file *<MQWFDIR>\scenario\WebCredit\CustomerUPES\run_trigger.bat* reflects your environment (adapt the PATHs in the various variables).
- Next time a message arrives in the UPES queue the UPES will be started automatically.

The *Readme.html* located in the WebCredit folder provides useful information on how to configure the example.

## APPENDIX A

### How to add/create environment variables in Windows NT

- Start the control panel.
- Select system.
- Click the Environment tab.
- On the Environment tab in the list where you want to add the variable, click any existing variable name and type, appending the value you want to Add.
- If you want to create a new user variable, type the name of the new variable in the variable box and type the value in the value box.
- Click Set.
- You have to restart the computer to make the new settings effective.

### How to add virtual directory in IIS/PWS

- Click on the IIS/PWS icon.

- Click on Advanced Options.
- Click on Add.
- Type the full directory path, eg *c:\jakarta-tomcat-3.2.3\bin\win32\i386*.
- Type an alias, eg ‘jakarta’.

**How to edit the Registry**

- On Windows NT click Start > Run.
- Type ‘regedit’.

Be very careful when editing it!

*Chandra Upadhyayula*
*Programmer Analyst (USA)*

## Heartbeat: channel monitoring tool for MQ clusters

One of the most difficult aspects of monitoring an MQSeries cluster is the fact that channels between queue managers in the cluster are dynamically created. There is no static list of channels against which a monitoring tool can check to make sure throughput is being maintained. Furthermore, it would be inefficient for such a tool to test all possible pathways in a cluster. From this point on, the sender/receiver channel pair that allows a message to flow from one queue manager to another within the cluster will be referred to as a pathway. Since any queue manager in the cluster can communicate with any other queue manager also within the cluster, the potential number of sender/receiver channel pairs is *n^2-n*. Most of these potential pathways are never exploited by the cluster because the typical MQSeries application sends messages to only a subset of a cluster’s queue managers.

Heartbeat is a tool that will monitor the movement of messages throughout the cluster and report via e-mail whenever issues arise. To better understand how Heartbeat decides which pathways to test, some subtle details about clusters need to be explained. First, it is important to draw a distinction between dynamically created channels and static channels. In a traditional, non-clustered MQSeries environment messages are put to a remote queue definition, which is associated with a transmission queue. Likewise a sender channel on the queue manager is associated with this transmission queue. This is how the queue manager knows which channel to send the message across. This channel is considered static because it was manually defined at some point. This definition will always exist within the queue manager no matter what the state of the channel or how long it has remained unused.

Another important characteristic about static channels is that while a static channel may have no status its definition always exists. This can be illustrated with two commands issued within **runmqsc**. Issuing **display channel(*)** will list all static channels that have been defined for the queue manager, regardless of their status. **display chstatus(*)**, on the other hand, displays only those channels which have some status. These states cover varying conditions the channel may be in: initializing, binding, starting, running, stopping, stopped, retrying, or requesting. However, if the channel has no status whatsoever, it won't appear in the list. A good example of this is when a channel is first defined; because a **start channel(*channelname*)** has never been issued, the queue manager has not yet evaluated the condition of this channel.

Dynamic channels, as the name indicates, are created automatically by the queue manager only when they are needed to send a message along a pathway in the cluster for which no channel definition already exists. The information necessary to create this channel is gathered from one of the full repositories for the cluster. The queue managers at each end of the pathway create a cluster sender channel and cluster receiver channel respectively. While a channel definition is created it is important to note that dynamic channels never appear in the list of channels generated by the **display channel(*)** command. This is what makes it so difficult for a monitoring tool directly to test channels in a cluster; there is no way to obtain a list of all the channels a clustered queue manager has created dynamically. The **display chstatus(*)** command, while it does give a list of dynamic channels, only lists those which have a status. As it is possible for dynamic channels to exist that have no state, this too is an unreliable method for obtaining a list of dynamic channels.

Note that while cluster channels can only be created dynamically it is possible for both clustered and non-clustered channels to be static. The most obvious example of this is the cluster channels defined between any member of the cluster and one or more full repositories. When adding a new queue manager to a cluster, only a cluster sender and receiver channel to one of the full repositories needs to be created. At this point, the repository can provide the new queue manager with information about the structure of the cluster along with channel definitions to all the other full repositories. Whenever the new queue manager needs to communicate with other members of the cluster it dynamically creates the necessary channels. However, the cluster sender and cluster receiver channel between the new queue manager and the full repository are static. These channels will appear in the list from the **display channel(*)** command and their definitions will remain even if the queue manager is removed from the cluster (whereas all dynamic channel definitions will be erased).

Heartbeat will ensure that only useful pathways are tested by exploiting another fact about MQSeries clusters: the definition for any dynamically created cluster channel will be removed after 90 days of inactivity. Armed with this fact it is possible to test only those channels for which messages have travelled across in the last 90 days rather than all possible channels. But, as explained before, obtaining this list cannot be done through straightforward means. Instead, it will be done indirectly by making use of another **runmqsc** command: **display clusqmgr(*)**, which returns a list of all queue managers in the current queue manager’s repository. If this command is issued on a queue manager that is a full repository for the cluster, it will contain a list of all queue managers in the cluster. If it isn’t a full repository, this queue manager’s partial repository will only contain those queue managers with which it has communicated in the last 90 days. Accordingly, the pathways between this queue manager and any queue managers not on its list of cluster queue managers shouldn’t be tested, as these pathways will not have been used in the last 90 days. To determine the names of the channels associated with these queue manager names, a particular naming convention will be used throughout the cluster. To illustrate, imagine queue managers called TEST1 and TEST2. These are both full repositories for the cluster called CLUS1. The cluster channels we statically define between the two will be named as follows:

- TEST1 cluster sender channel: TO.TEST2.
- TEST1 cluster receiver channel: TO.TEST1.
- TEST2 cluster sender channel: TO.TEST1.
- TEST2 cluster receiver channel: TO.TEST2.

Those who are familiar with setting up clusters will notice that this convention actually follows the one described in the *MQSeries Queue Manager Clusters* guide. As the name indicates, Heartbeat will be sending messages across these channels to determine their condition. These messages will contain a timestamp which allows Heartbeat to determine a roundtrip time for the message. But these messages will need to be put into a queue and picked up again from a reply queue. Keeping with the above example, the following queues need to be defined:

- TEST1: queue called TO.TEST1, member of the CLUS1 cluster.
- TEST1: queue called TEST1_HB_REPLY, member of the CLUS1 cluster.
- TEST2: queue called TO.TEST2, member of the CLUS1 cluster
- TEST2: queue called TEST2_HB_REPLY, member of the CLUS1 cluster.

To generalize, the following objects need to be defined on all queue managers in the cluster for Heartbeat to function:

- Cluster sender channel(s) defined as TO.*remote_qmgrname*, where *remote_qmgrname* is the name of the queue manager that this sender channel is pointing at.
- Cluster receiver channel(s) defined as TO.*this_qmgrname*, where this_qmgrname is the name of the queue manager this channel is defined on.
- A clustered local queue with the same name as the cluster receiver channel(s), TO.*this_qmgrname*.
- A clustered local queue called *this_qmgrname*_HB_REPLY, where *this_qmgrname* is the name of the queue manager this queue is defined on.

The flow of the Heartbeat program is relatively straightforward. When started, Heartbeat queries the queue manager for the list of all clustered queue managers it has communicated with in the last 90 days by issuing the **display clusqmgr(*)** command. Because of the naming convention Heartbeat is able to infer the name of the channel that is used to communicate with each queue manager in the list. The naming convention also ensures that a clustered queue exists on each remote queue manager by the same name as the channel used to deliver the message to that queue. In other words, if queue manager TEST1 sees queue manager TEST2 we know the cluster sender channel that TEST1 uses to send messages to TEST2 is called TO.TEST2, and there exists a clustered queue on TEST2 also called TO.TEST2. By putting a message in the TO.TEST2 queue from queue manager TEST1 Heartbeat ensures that the message will travel over the TO.TEST2 channel. A second program, *HBreply.pl*, is triggered when the message lands on the TO.TEST2 queue. This program simply puts the message to the queue specified in its ReplyToQ parameter. In this case it would be TEST1_HB_REPLY, as the message originated from queue manager TEST1. Heartbeat, meanwhile, is waiting for the message to return, and when it does, the timestamp in the message is compared with the current system time. This yields an accurate measure of the roundtrip time that the message took to travel through the cluster. Heartbeat will only wait a predetermined amount of time for a given message to return, at which time Heartbeat will send out a detailed e-mail indicating that there is a problem with the cluster and which channels are affected.

To start Heartbeat from the command line the following parameters must be specified:

```
    perl heartbeat.pl qmgrname mailto timeout wait_between_tests
logfile
```

- *qmgrname* – the name of the queue manager that Heartbeat should connect to. Note, this parameter must be specified even if the queue manager to which Heartbeat should connect is the default queue manager.
- *mailto* – the e-mail address of the person who should receive all status messages from Heartbeat.
- *timeout* – the amount of time in seconds that Heartbeat should wait for a message it sends out to be returned. When this time expires, an e-mail message will be sent out.

- *wait_between_tests* – the amount of time in seconds that Heartbeat should wait before testing all the channels again. If this value is very small you may experience slowdown in the cluster's message throughput because of the load that Heartbeat will be putting on the channels.
- *Logfile* – the fully qualified path and filename that Heartbeat's log will be written to. This log is in addition to the e-mail notifications.

Ideally *hbreply.pl* should be triggered when a message arrives on each of the TO.*qmgrname* queues in the cluster. It only takes a single parameter: the name of the queue manager it should connect to – *perl hbreply.pl qmgrname*.

The code for Heartbeat and Hbreply can be found on the Web at www.xephon.extras/heartbeat.txt.

*Brandon Duncan*
*Founder, MQSeries.net (USA)*

## WebSphere MQSeries Integrator cross-reference

### THE PROBLEM

When I've visited clients who have a large number of WebSphere MQSeries Integrator (WMQSI) flows and nodes it's often been very difficult to see the wood for the trees. A simple request to find and change a flow can take – and indeed has taken – many hours.

### SUGGESTED SOLUTION

What was needed was some sort of cross-reference listing of all the flows with their respective nodes, and here is one way to do just that.

Unfortunately there is no official IBM documentation that tells us how and where this sort of data is kept. A look at the databases used by WMQSI did not reveal what was required.

Instead I looked at the Control Centre and the option that allows you to export your workspace (eg click File and select ‘export all in workspace’). This creates an XML file with the structure shown below.

## WMQSI-EXPORTED XML FILE

```
<?xml version="1.Ø" encoding="UTF-8"?>
<!DOCTYPE XMI SYSTEM "mqsi.dtd" >
<XMI xmi.version="1.Ø">
  <XMI.header>
     <XMI.documentation>
        <XMI.owner>ruud</XMI.owner>
     </XMI.documentation>
  </XMI.header>
  <XMI.content>
     <MessageProcessingNodeType xmi.label="FFFFFFFF">
        <MessageProcessingNode xmi.label="NNNNNNNN">
          <AttributeGroup[1]>
             <Attribute value="QQQQQQQQ"/>
          </AttributeGroup[1]>
          <AttributeGroup[2]>
             <Attribute value="DDDDDDDD"/>
          </AttributeGroup[2]>
          <Attribute>
            <XMI.extension>
               <ComputeMetadata>
                  <TableDescriptor dataSource="DSDSDSDS"
                                   tableName="TBTBTBTB"/>
               </ComputeMetadata>
            </XMI.extension>
          </Attribute>
        </MessageProcessingNode>
      </MessageProcessingNodeType>
   </XMI.content>
</XMI>
```

Where:

- FFFFFFFF represents the flow name.
- NNNNNNNN represents the node name.
- QQQQQQQQ represents the queue name.
- DDDDDDDD represents the domain name.
- DSDSDSDS represents the data source name.
- TBTBTBTB represents the table name.

This is obviously a subset of all the various tags and attributes that are available. The next task was to find an XML parser in order to analyse the export file. I decided to use WMQSI itself!

## MQ AND MQSI REQUIREMENTS

Define four local queues on the broker's queue manager.

- Error queue: MQSIXREF_ERR.
- Input queue: MQSIXREF_IN with backout queue MQSIXREF_ERR.
- Output queue: MQSIXREF_SIM, which is for a 'simple' xref.
- Output Queue: MQSIXREF_DTL, which is for a 'detailed' xref.

A simple message flow was created, as shown in Figure 1, with the following attributes:

- The MQInput node MQSIXREF_IN has DOMAIN=BLOB and queue MQSIXREF_IN.
- 'Strip' is a compute node that strips off the first two lines, which contain the XML version number and DTD details. This is required to enable the later compute nodes to work.
- 'Change to XML' is a ResetContentDescriptor node to change the domain back to XML.
- 'Do 2 things' is a FlowOrder node.
- 'SimpleXref' is a compute node which analyses the export data and

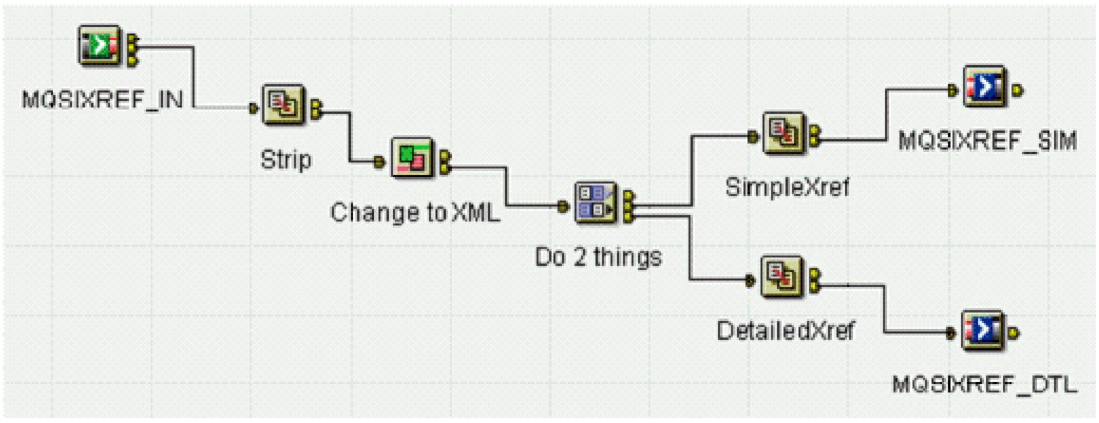


*Figure 1: A simple message flow*

produces a simple cross-reference of flows and nodes.

- ‘DetailedXref’ is like SimpleXref but also outputs details of the message domain, queue name, data source, and table name.

### Compute node Strip

This is simply a one-line statement to remove the two unwanted lines:

```
SET OutputRoot."BLOB"."BLOB" = SUBSTRING(InputRoot."BLOB"."BLOB"
FROM 74);
```

Ensure you tick the ‘Copy message headers’ radio button.

### Compute node SimpleXref

The compute node used to analyse the export file looks is detailed below.

Ensure that you tick the ‘Copy message headers’ radio button – this generates some ESQL (which has been omitted).

- Enter SQL below this line. SQL above this line might be regenerated, causing any modifications to be lost:

```
DECLARE MPN   INTEGER;
DECLARE NODES INTEGER;
DECLARE J     INTEGER;
DECLARE K     INTEGER;
```

- Calculate the total number of flows:

```
SET MPN =
CARDINALITY(InputBody."XMI"."XMI.content".
MessageProcessingNodeType[]);
SET OutputRoot."XML".XREF.Flows.TotalNumber = MPN;
```

- Loop round and pick up the flow name:

```
SET J = 1;
WHILE J <= MPN DO
  SET OutputRoot.XML.XREF.Flow[J].Name =
     InputBody."XMI"."XMI.content".MessageProcessingNodeType[J]
.(XML.attr)"xmi.label";
  SET K = 1;
```

- Calculate the total number of nodes in this flow:

```
SET NODES =
CARDINALITY(InputBody."XMI"."XMI.content".
MessageProcessingNodeType[J]
```

```
     .MessageProcessingNode[]);
```

- Loop round and pick up the node names:

```
     WHILE K <= NODES DO
       SET OutputRoot.XML.XREF.Flow[J].Node[K] = CAST(K AS CHARACTER)
||
       InputBody."XMI"."XMI.content".MessageProcessingNodeType[J].
     MessageProcessingNode[K].(XML.attr)"xmi.label";
       SET K = K + 1;
       END WHILE;
       SET J = J + 1;
     END WHILE;
```

### Compute node DetailedXref

If additional information is required the above compute node can easily be changed. For example, you might want to extract the domain name and the name of the input queue as well as the name of any databases and tables being accessed.

- Enter SQL below this line. SQL above this line might be regenerated, causing any modifications to be lost.

```
DECLARE MPN     INTEGER;
DECLARE NODES   INTEGER;
DECLARE J       INTEGER;
DECLARE K       INTEGER;
DECLARE QS      INTEGER;
DECLARE DM      INTEGER;
DECLARE DB      INTEGER;
DECLARE QNAME   CHARACTER;
DECLARE DOMAIN  CHARACTER;
DECLARE DSOURCE CHARACTER;
DECLARE TBNAME  CHARACTER;
```

- Calculate the total number of flows:

```
SET MPN =
CARDINALITY(InputBody."XMI"."XMI.content".
MessageProcessingNodeType[]);
SET OutputRoot."XML".XREF.Flows.TotalNumber = MPN;
```

- Loop round and pick up the flow name:

```
SET J = 1;
WHILE J <= MPN DO
  SET OutputRoot.XML.XREF.Flow[J].Name =
InputBody."XMI"."XMI.content".MessageProcessingNodeType[J].
(XML.attr)"xmi.label";
```

```
  SET K = 1;
```

- Calculate the total number of nodes in this flow:

```
SET NODES =
CARDINALITY(InputBody."XMI"."XMI.content".
MessageProcessingNodeType[J].
MessageProcessingNode[]);
```

- Loop round and pick up the node names:

```
WHILE K <= NODES DO
SET OutputRoot.XML.XREF.Flow[J].Node[K] = CAST(K AS CHARACTER) ||
InputBody."XMI"."XMI.content".MessageProcessingNodeType[J].
MessageProcessingNode[K].(XML.attr)"xmi.label";
```

- Attempt to pick up the queue name:

```
SET QS = CARDINALITY(InputBody."XMI"."XMI.content".
MessageProcessingNodeType[J].
MessageProcessingNode[K].AttributeGroup[1].Attribute[]);
    IF QS > Ø THEN
      SET QNAME = InputBody."XMI"."XMI.content".
MessageProcessingNodeType[J].
         MessageProcessingNode[K].AttributeGroup[1].
Attribute[1].(XML.attr)"value";
      SET OutputRoot.XML.XREF.Flow[J].Node[K].QName = QNAME;
```

- Attempt to pick up the domain name:

```
SET DM = CARDINALITY(InputBody."XMI"."XMI.content".
MessageProcessingNodeType[J].
MessageProcessingNode[K].AttributeGroup[2].Attribute[]);
      IF DM > Ø THEN
        SET DOMAIN = InputBody."XMI"."XMI.content".
MessageProcessingNodeType[J].
MessageProcessingNode[K].AttributeGroup[2].
Attribute[1].(XML.attr)"value";
        SET OutputRoot.XML.XREF.Flow[J].Node[K].Domain = DOMAIN;
      END IF;
    END IF;
```

- Attempt to pick up the datasource name:

```
SET DB = CARDINALITY(InputBody."XMI"."XMI.content".
MessageProcessingNodeType[J].
     MessageProcessingNode[K].Attribute."XMI.extension".
ComputeMetadata.TableDescriptor[]);
    IF DB > Ø THEN
      SET DSOURCE = InputBody."XMI"."XMI.content".
MessageProcessingNodeType[J].
MessageProcessingNode[K].Attribute."XMI.extension".
```

```
ComputeMetadata.TableDescriptor[1].(XML.attr)"dataSource";
     SET OutputRoot.XML.XREF.Flow[J].Node[K].DataSource = DSOURCE;
```

- Attempt to pick up the table name:

```
SET TBNAME = InputBody."XMI"."XMI.content".
MessageProcessingNodeType[J].
MessageProcessingNode[K].Attribute."XMI.extension".
ComputeMetadata.TableDescriptor[1].(XML.attr)"tableName";
      SET OutputRoot.XML.XREF.Flow[J].Node[K].TableName = TBNAME;
    END IF;
    SET K = K + 1;
  END WHILE;
  SET J = J + 1;
END WHILE;
```

**Result from compute node SimpleXref**

When the message has been placed on the output queue run a utility to read the message and output it to an XML file.

Here, the total number of flows is 11. The first four have been shown.

Note that ‘CatchTrace’ is a sub-flow within ‘MyException’ and a flow in its own right. ‘MQSIXREF’ is the flow used to create the export file.

Note also that each of the node names has been preceded with a number to help show the number of nodes within a flow, but (unexpectedly) it has been displayed in reverse order.

```
- <XREF>
- <Flows>
  <TotalNumber>11</TotalNumber>
  </Flows>
- <Flow Name="MyException">
  <Node>1CatchTrace1</Node>
  <Node>2Throw2</Node>
  <Node>3Filter1</Node>
  <Node>4MQOutput1</Node>
  <Node>5Compute1</Node>
  <Node>6MQInput1</Node>
  </Flow>
- <Flow Name="CatchTrace">
  <Node>1Throw1</Node>
  <Node>2Trace1</Node>
  <Node>3TryCatch1</Node>
  </Flow>
- <Flow Name="MQSIXREF">
  <Node>1MQSIXREF_DTL</Node>
  <Node>2MQSIXREF_SIM</Node>
  <Node>3DetailedXref</Node>
  <Node>4SimpleXref</Node>
  <Node>5Do 2 things</Node>
  <Node>6Change to XML</Node>
  <Node>7Strip</Node>
  <Node>8MQSIXREF_IN</Node>
  </Flow>
- <Flow Name="EX12">
  <Node>1AddOrder</Node>
  <Node>2AddCustomer</Node>
  <Node>3Throw1</Node>
  <Node>4Trace1</Node>
  <Node>5TryCatch1</Node>
  <Node>6E1Øout</Node>
  <Node>7E12in</Node>
  </Flow>
```

## Result from compute node DetailedXref

The same four flows have been shown here, but this time with more detail. Look out for the queue name, domain, DataSource, and TableName.

```
- <XREF>
- <Flows>
  <TotalNumber>11</TotalNumber>
  </Flows>
- <Flow Name="MyException">
  <Node>1CatchTrace1</Node>
  <Node>2Throw2</Node>
  <Node>3Filter1</Node>
- <Node>
  4MQOutput1
  <QName>E1Øout</QName>
  </Node>
  <Node>5Compute1</Node>
- <Node>
  6MQInput1
  <QName>TESTEXCEP</QName>
  <Domain>XML</Domain>
  </Node>
  </Flow>
- <Flow Name="CatchTrace">
  <Node>1Throw1</Node>
  <Node>2Trace1</Node>
  <Node>3TryCatch1</Node>
  </Flow>
- <Flow Name="MQSIXREF">
- <Node>
  1MQSIXREF_DTL
  <QName>MQSIXREF_DTL</QName>
  </Node>
- <Node>
  2MQSIXREF_SIM
  <QName>MQSIXREF_SIM</QName>
  </Node>
  <Node>3DetailedXref</Node>
  <Node>4SimpleXref</Node>
  <Node>5Do 2 things</Node>
  <Node>6Change to XML</Node>
  <Node>7Strip</Node>
- <Node>
  8MQSIXREF_IN
  <QName>MQSIXREF_IN</QName>
  <Domain>BLOB</Domain>
  </Node>
  </Flow>
- <Flow Name="EX12">
- <Node>
  1AddOrder
  <DataSource>TEST</DataSource>
  <TableName>ORDER</TableName>
  </Node>
- <Node>
  2AddCustomer
  <DataSource>TEST</DataSource>
  <TableName>CUSTOMER</TableName>
  </Node>
  <Node>3Throw1</Node>
  <Node>4Trace1</Node>
  <Node>5TryCatch1</Node>
- <Node>
  6E1Øout
  <QName>E1Øout</QName>
  </Node>
- <Node>
  7E12in
  <QName>E12in</QName>
  <Domain>XML</Domain>
  </Node>
  </Flow>
```

## CREATING A CROSS-REFERENCE: SUMMARY

All the steps required to produce the cross-reference are listed here:

- Start the Control Centre and create an empty workspace.
- Add all message flows to the workspace.

- Export all – this will export all flows that have been deployed as well as those in your workspace.
- Use the IH03 Supportpac utility MQPUT2 to place the XML export file on a queue, and feed the message through the WMQSI flow.
- Extract the message from the queues and save it as an XML file.

I used the **amqsget** sample as a model, and just added the write to a file.

- Click on the XML file to view the results in a browser window.

### Known issues

The XML export file's first two lines stop the compute node from working. To overcome this problem an extra compute node was added (called Strip) to strip them off.

WMQSI stores the code and comments of a compute node as XML attributes.

If you use the '>' or '<' characters in your comments the XML parser gets confused and produces an error. Ideally, WMQSI ought to store these as '&gt' and '&lt' strings.

## CONCLUSION

The method I've used has been tested on both WMQSI V2.0.2 and V2.1 and has worked without any problems. The largest export file used was 1.7Mb in size with about 120 message flows.

To find out where a node is used all you do is open up the XML file in your favourite browser and do a 'find'. This file can be stored on the network to allow everyone access to it or a local copy created by the developer. Ideally, this sort of function should be available within the base product itself.

If you found this article useful, or have suggestions for improving the solution, then please let me know via e-mail: *ruudvz@btclick.com*.

*Ruud van Zundert,*
*Independent Consultant (UK)*

# MQSI exception processing: request/reply messages: part two – subflows

Last month we looked at the options for dealing with request/reply messages. We continue here with specific reference to subflows.

## USING THE MQRFH2 HEADER

The SupportPacs work well but some shops aren't comfortable deploying them into production systems because they're not supported. Another way of storing the reply information is to make use of the <*usr*> folder in the MQRFH2 header. (Please refer to the MQ manual for a detailed description of the MQRFH2 header.)

The theory is the same – save the reply information during the request flow and retrieve the reply information during the reply flow, as shown in Figure 1. The ESQL of the compute node 'Save ReplyInfo' will save the reply information into the *usr* folder on the MQRFH2, as below. Just in case RFH2 header is not there:

```
SET OutputRoot.MQMD.MsgType = MQMT_REQUEST;
SET OutputRoot.MQMD.Format = MQFMT_RF_HEADER_2;
```

Set the RFH2 header:

```
SET OutputRoot.MQRFH2.(MQRFH2.Field)Format = 'MQSTR   ';
```

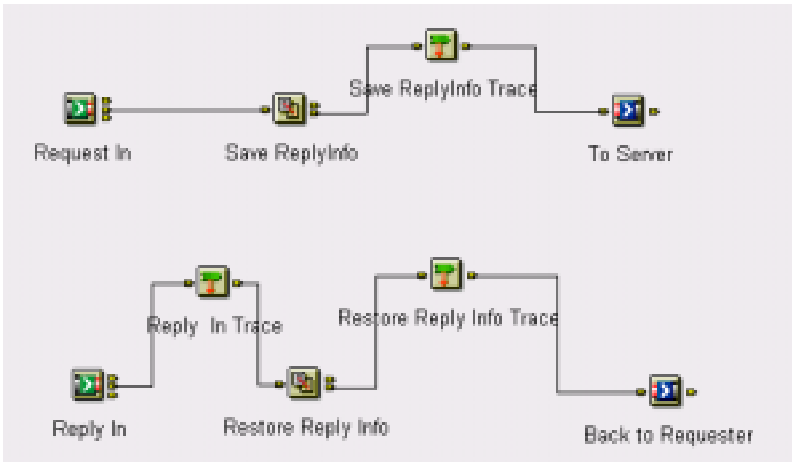


*Figure 1: Saving and retrieving information*

Save the reply information:

```
SET OutputRoot.MQRFH2.usr.ReplyToQ = InputRoot.MQMD.ReplyToQ;
SET OutputRoot.MQRFH2.usr.ReplyToQMgr = InputRoot.MQMD.ReplyToQMgr;
```

Note that there is a bug in setting the RFH2 header if the incoming message does not already have an RFH2 header appended. It does not work properly when you choose to 'Copy entire message'. In order for the above code to work you need to select 'Copy message headers', not 'Copy entire message'. Then, in the ESQL, add the following to copy the body of the incoming message:

```
SET OutputRoot.XML = InputRoot.XML;
```

On the reply flow, since the header is passed along with the message, the compute node 'Restore Reply Info' simply needs to move the fields back to the MQMD. Restore the reply information:

```
SET OutputRoot.MQMD.ReplyToQ = InputRoot.MQRFH2.usr.ReplyToQ;
SET OutputRoot.MQMD.ReplyToQMgr = InputRoot.MQRFH2.usr.ReplyToQMgr;
```

The trace file produced in this flow can be found at www.xephon.com/extras/save.txt.

The beauty of using the <*usr*> folder is that there is no need to add a SupportPac; additionally, the reply information always follows the message. The message flow is simpler than when utilizing a SupportPac.

One thing to remember when using this method is to make sure all the queue managers running on different tiers are at the version that supports this MQRFH2 header – especially those shops running MQ for OS/390.

## EXCEPTION HANDLING SUBFLOW

In this section we will begin to construct the exception handling subflow. This subflow is built according to the specifications laid out previously. Figure 2 illustrates the flow.

When an exception is caught it will follow this flow. The flow order node 'Write Error before throw' is there to make sure the error message and the request message (or reply message) with error information inserted is written out first. The MQOutput nodes will both write messages outside the syncpoint such that the messages will be committed even if the 'ReThrowError' node throws an error later and rolls the message back to

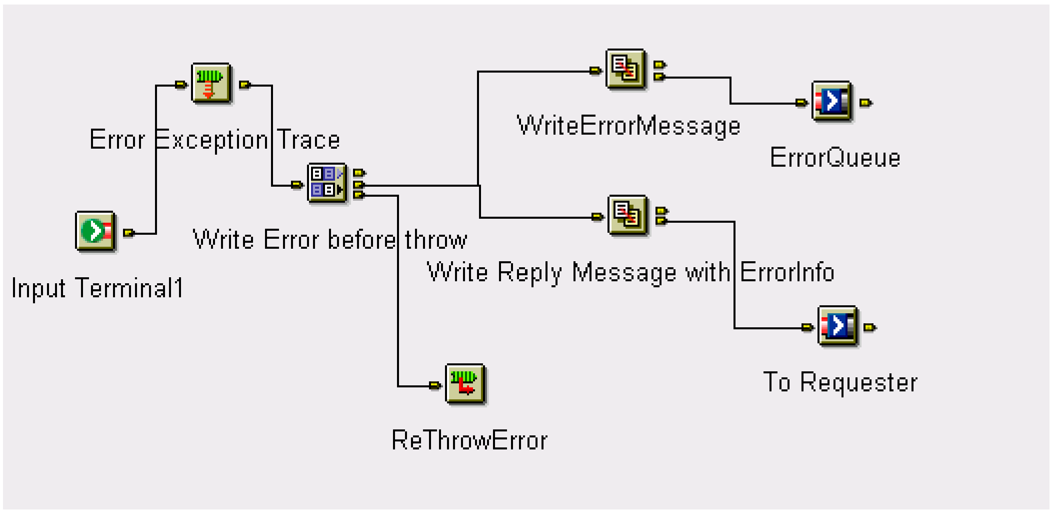


*Figure 2: Exception handling subflow*

the input queue, for which we set a backout threshold of 1 with a corresponding backout queue assigned. In this way the message will only go through the message flow once.

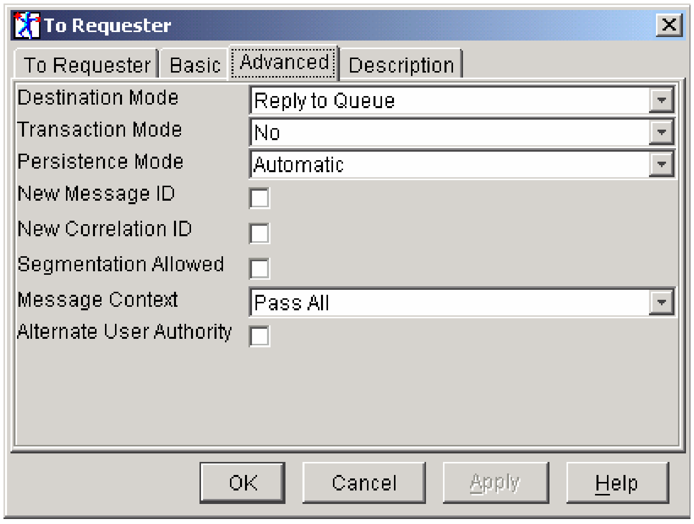
The ESQL of the 'WriteErrorMessage' compute node is the same as the one we used in the datagram message. The ESQL of the 'Write Reply Message with Error Info' compute node is very similar. The only difference is that the error information is inserted as part of the original message itself.

```
SET OutputRoot = InputRoot;
```

Enter SQL below this line. SQL above this line might be regenerated, causing any modifications to be lost.

```
/* Error number extracted from exception list */
DECLARE Error INTEGER;
/* Current path within the exception list */
DECLARE Path CHARACTER;
DECLARE ErrorType  CHARACTER;
DECLARE ErrorTrans CHARACTER;
DECLARE ErrorText  CHARACTER;
DECLARE SQLCode CHARACTER;
SET ErrorType  = '';
SET ErrorTrans = '';
SET ErrorText  = '';
/* First copy the MessageID to CorrelId for the Error Message if it is
a request */
IF InputRoot.MQMD.MsgType = MQMT_REQUEST  THEN
     SET OutputRoot.MQMD.CorrelId = InputRoot.MQMD.MsgId;
END IF;
/* Start at first child of exception list */
```

```
SET Path ='InputExceptionList.*[1]';
/* Loop until no more children */
WHILE EVAL( 'FIELDNAME(' || Path || ') IS NOT NULL' ) DO
  /* Check if error number is available */
  IF EVAL( 'FIELDNAME(' || Path || '.Number) IS NOT NULL' ) THEN
    /* Remember only the deepest error number */
    SET Error    = EVAL( Path || '.Number' );
    SET ErrorType  = EVAL('FIELDNAME('||Path||')');
     IF ErrorType = 'DatabaseException' THEN
      SET SQLCode = COALESCE(EVAL(Path || '.Insert[2].Text'),'');
     END IF;
    SET ErrorTrans = EVAL(Path || '.Label');
    SET ErrorText  = EVAL(Path || '.Text');
    IF ErrorType = 'UserException' THEN
     SET ErrorText = ErrorText || ' (' || (EVAL(Path ||
'.Insert.Text')) || ')';
    END IF;
  END IF;
  /* Step to last child of current element (usually a nested exception
list) */
  SET Path = Path || '.*[LAST]';
END WHILE; /* End loop */
SET OutputRoot.XML.Message.Exception.Type      = ErrorType;
IF ErrorType = 'DatabaseException' THEN
  SET OutputRoot.XML.Message.Exception.SQLCode     = SQLCode;
END IF;
SET OutputRoot.XML.Message.Exception.Number        = CAST(Error AS
CHAR);
SET OutputRoot.XML.Message.Exception.SuspenseTimestamp   =
CAST(CURRENT_TIMESTAMP AS CHAR);
SET OutputRoot.XML.Message.Exception.InboundQueue   =
```



| | |
|---|---|
| Destination Mode | Reply to Queue |
| Transaction Mode | No |
| Persistence Mode | Automatic |
| New Message ID | ☐ |
| New Correlation ID | ☐ |
| Segmentation Allowed | ☐ |
| Message Context | Pass All |
| Alternate User Authority | ☐ |

*Figure 3:MQOutput node 'To Requester'*

```
InputRoot.MQMD.SourceQueue;
SET OutputRoot.XML.Message.Exception.Transaction    = ErrorTrans;
SET OutputRoot.XML.Message.Exception.Reason         = ErrorText;
--SET OutputRoot.XML.Message.Exception.DetailException   =
InputExceptionList;
```

Since this message will be sent to the requester the detail of the exception may not be needed, so I commented out the last part to make the message shorter. But you can always copy it if you think it is required.

There is something sleek in the MQOutput node 'To Requester' shown in Figure 3. On the Advanced tab we select from the drop-down list of Destination Mode to Reply to Queue without specifying the queue manager and queue name in the Basic tab, which we usually do. The MQOutput node will then put the message to the queue specified in the MQMD. This is similar to using DestinationList, which can provide some dynamic values, depending on the MQMD of the message.

## USING THE EXCEPTION SUBFLOW

The main difference in exception handling between a datagram and the request/reply type of message is in where to attach the exception handling subflow. For request messages the exception subflow is attached to the catch terminal of the input node, which is the same for datagrams. Figure 4 provides an example.

For reply messages, as explained earlier, it is necessary for the reply message to retrieve the reply information on the reply message flow first,

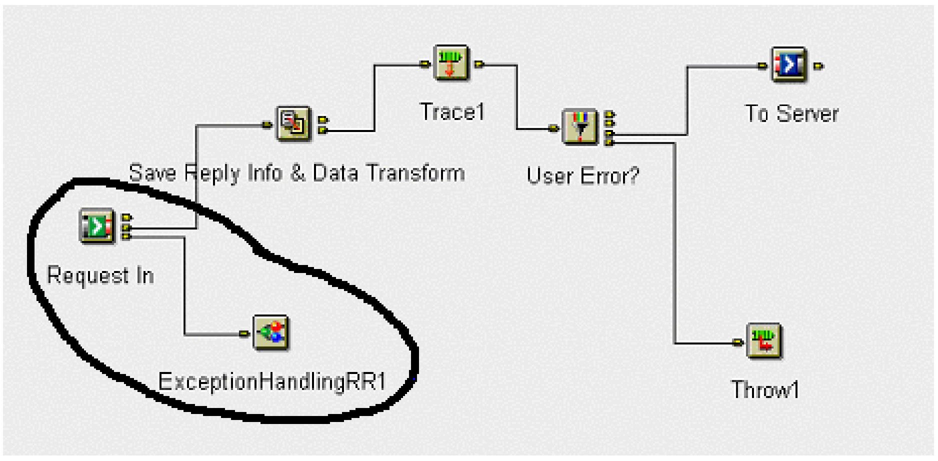


*Figure 4: Exception subflow*

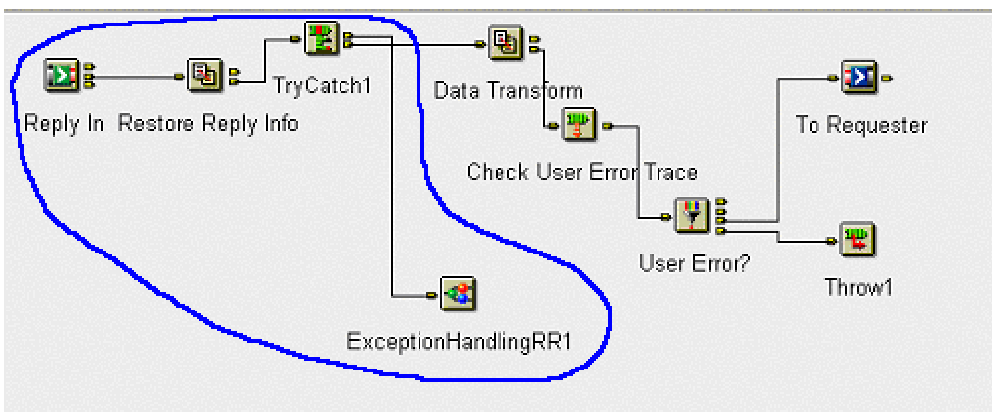


*Figure 5: 'Restore Reply Info' node*

before we pass on the message to the rest of the message flow. This will ensure that, when the exception is caught, the reply message will be forwarded with the error information populated, to the correct reply queue where the requester is waiting for it. So the exception handling subflow for the reply message will be appended not to the catch terminal of the input node but to the catch terminal of a try-catch node that we wired right after the compute node that restored the reply information on the reply message flow. Figure 5 illustrates the setup.

Here the 'Restore Reply Info' compute node will restore the reply information as described previously. In this example we are making use of the MQRFH2 header to store and restore the reply information. The ESQL consists of just two statements:

- Restore the reply information:

```
SET OutputRoot.MQMD.ReplyToQ = InputRoot.MQRFH2.usr.ReplyToQ;
SET OutputRoot.MQMD.ReplyToQMgr = InputRoot.MQRFH2.usr.ReplyToQMgr;
```

- Set the MsgType equal to reply just in case:

```
SET OutputRoot.MQMD.MsgType = MQMT_REPLY;
```

We also set the message type to reply (in case it is not set up properly by the server program that processed the request) because we will check this field in the exception logic. We only need to copy the Msg-ID into the Correl-ID if the message type is a request. For a reply message the server program should have already copied this field when it constructed the reply message.

A try catch node is connected right after this so that when an exception happens later in the flow the message will roll back to the try catch node; it will be caught by the catch terminal of this node and the reply information will be preserved.

In the exception subflow, when it has written out both the error message and the reply message populated with error information, it will do a throw and the message will then be rolled back to the input queue of the input node and subsequently to the backout queue.

The rest of the flow in this example will simulate a bogus user exception – the same as the one used in the datagram example. When a reply message encounters an error, messages are written out. The sample reply message is like this:

```
<Message>
<MsgBody>a sample reply message</MsgBody>
</Message>
```

Here is a trace output from the exception handling subflow.

```
*** Exception Trace generated at 2ØØ1-Ø9-2Ø 18:34:15.961
(
  (Øx1ØØØØØØ)UserException = (
    (Øx3ØØØØØØ)File     = 'F:/build/S2Ø2_P/src/DataFlowEngine/
BasicNodes/ImbThrowNode.cpp'
    (Øx3ØØØØØØ)Line     = 229
    (Øx3ØØØØØØ)Function = 'ImbThrowNode::evaluate'
    (Øx3ØØØØØØ)Type     = 'ComIbmThrowNode'
    (Øx3ØØØØØØ)Name     = 'eØd5f1f2-e8ØØ-ØØØØ-ØØ8Ø-a1Ø328c4362e'
    (Øx3ØØØØØØ)Label    = 'Alex Test2.Throw1'
    (Øx3ØØØØØØ)Text     = 'User exception thrown by throw node'
    (Øx3ØØØØØØ)Catalog  = 'MQSIV2Ø1'
    (Øx3ØØØØØØ)Severity = 1
    (Øx3ØØØØØØ)Number   = 3ØØ1
    (Øx1ØØØØØØ)Insert   = (
      (Øx3ØØØØØØ)Type = 5
      (Øx3ØØØØØØ)Text = 'The mandatory field of AddOrder.xml is
missing!!'
    )
  )
)
```

An error message is written out to the error queue as shown here:

```
<Exception>
<Type>UserException</Type>
<Number>3ØØ1</Number>
```

```
<SuspenseTimestamp>TIMESTAMP &apos;2001-09-20 18:34:15.991&apos;</
SuspenseTimestamp>
<InboundQueue>ALEX_REPLY_IN</InboundQueue>
<Transaction>Alex Test2.Throw1</Transaction>
<Reason>User exception thrown by throw node (The mandatory field of
AddOrder.xml is missing!!)</Reason>
<DetailException>
 <UserException>
 <File>F:/build/S202_P/src/DataFlowEngine/BasicNodes/ImbThrowNode.cpp</
File>
 <Line>229</Line><Function>ImbThrowNode::evaluate</Function>
 <Type>ComIbmThrowNode</Type><Name>e0d5f1f2-e800-0000-0080-
a10328c4362e</Name>
 <Label>Alex Test2.Throw1</Label><Text>User exception thrown by throw
node</Text>
 <Catalog>MQSIV201</Catalog><Severity>1</Severity><Number>3001</Number>
 <Insert><Type>5</Type>
 <Text>The mandatory field of AddOrder.xml is missing!!</Text>
 </Insert></UserException>
</DetailException>
</Exception>
```

The reply message, with the error information populated, is sent back to the queue specified in the MQMD:

```
<Message>
 <MsgBody>a sample reply message</MsgBody>
 <Exception>
  <Type>UserException</Type>
  <Number>3001</Number>
  <SuspenseTimestamp>TIMESTAMP &apos;2001-09-20 18:34:16.281999&apos;</
SuspenseTimestamp>
  <InboundQueue>ALEX_REPLY_IN</InboundQueue>
  <Transaction>Alex Test2.Throw1</Transaction>
  <DetailException>
   <UserException>
    <File>F:/build/S202_P/src/DataFlowEngine/BasicNodes/
ImbThrowNode.cpp</File>
    <Line>229</Line>
    <Function>ImbThrowNode::evaluate</Function>
    <Type>ComIbmThrowNode</Type>
    <Name>e0d5f1f2-e800-0000-0080-a10328c4362e</Name>
    <Label>Alex Test2.Throw1</Label>
    <Text>User exception thrown by throw node</Text>
    <Catalog>MQSIV201</Catalog>
    <Severity>1</Severity>
    <Number>3001</Number>
    <Insert>
     <Type>5</Type>
     <Text>The mandatory field of AddOrder.xml is missing!!</Text>
    </Insert>
   </UserException>
  </DetailException>
 </Exception>
 <xception>
  <Reason>User exception thrown by throw node (The mandatory field of
AddOrder.xml is missing!!)</Reason>
 </xception>
</Message>
```

The original message is rolled back to the backout queue.

## COMBINED EXCEPTION HANDLING SUBFLOW

Combining the two subflows we created for the datagram message and request/reply message we have a general purpose exception handling subflow, which is shown in Figure 6.

After testing the above flow a problem arose with the MQOutput node 'To Requester'. It seems that, when Reply To Queue was selected from the Advanced tab Destination Mode drop-down list, the MQOutput node tried to set the context information, even though we selected 'Pass All' on the Advanced tab Message Context.

Worst of all, it altered the MsgType field on the MQMD. It altered the request message to a reply message, and the reply message to a datagram. If this is acceptable within your organization you can leave it as it is. But this is not really desirable since our strategy of exception handling depends

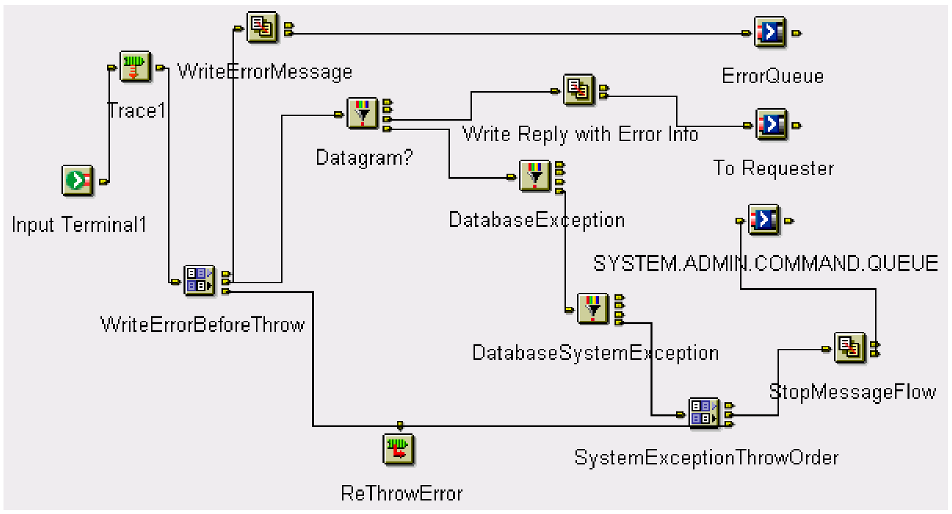
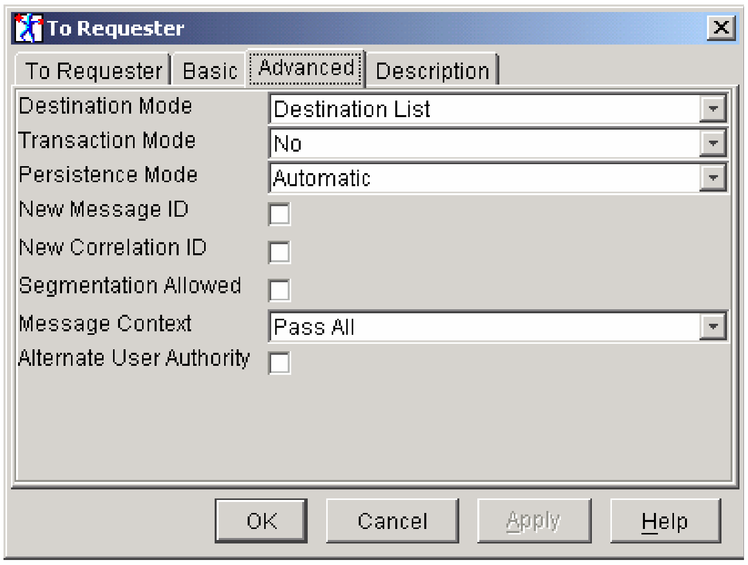


*Figure 6: General purpose exception handling subflow*



| | |
|---|---|
| Destination Mode | Destination List |
| Transaction Mode | No |
| Persistence Mode | Automatic |
| New Message ID | ☐ |
| New Correlation ID | ☐ |
| Segmentation Allowed | ☐ |
| Message Context | Pass All |
| Alternate User Authority | ☐ |

*Figure 7: Altering the Requester MQOutput node*

on this MsgType field and we would like this field to be accurate, reflecting the true message type.

So the 'Write Reply with Error Info' compute node is modified and adds ESQL to populate the reply information to the DestinationList, as shown here.

- Set MQ DestinationList if you want to use the DestinationList instead of Reply To Queue:

```
SET OutputDestinationList.Destination.MQDestinationList.
DestinationData.queueName = InputRoot.MQMD.ReplyToQ;
SET OutputDestinationList.Destination.MQDestinationList.
DestinationData.queueManagerName = InputRoot.MQMD.ReplyToQMgr;
```

Also in the Advanced tab, select Destination and Message and the 'To Requester MQOutput' node is altered, as shown in Figure 7.

## CONCLUSION

Once again, this article serves only as a reference or as a framework upon which to build your requirements. One challenge with the request/reply scenario is the need to send the request or reply message populated with error information back to the requester. In the exception subflow this would mean that it requires the message parser to insert the error information in the original message. If the exception caught was a parser error in the first place, the exception subflow will throw a parser exception in the 'Write Reply with Error Info' compute node. In this case, the subflow will not be able to send the reply message with error information to the requester and the requester will still time out, not receiving the reply, since the write from the hub will fail.

*Alex Au*
*IT Architect, IBM Global Services (USA)*

## Maximizing message availability

Now that MQSeries on OS/390 supports shared queues and queue sharing groups, when should shared channels and generic ports be used instead of, or in conjunction with, either MQSeries clustering or point-to-point channels? This article outlines the available MQSeries technologies, discusses the application and environmental considerations influencing the use of each, and offers recommendations.

### MESSAGING AND PARALLEL PROCESSING

There is a 'good fit' between messaging and parallel processing. That's because it's easy to imagine work requests as messages originating from various client nodes and being distributed to server nodes, where they are processed. In the case of an MQSeries network, messages arrive from the network and are placed on queues to be processed by server applications. As part of the server processing a completion message is often returned to the originating client, though this is not a requirement of the MQSeries architecture.

The beauty of parallel message processing from a business perspective (the only one we care about!) is that it allows you to deliver high throughput and high availability applications. These processing characteristics allow us to meet volume, response time, and availability goals.

In an MQSeries parallel message processing system, if we want to increase the processing rate of messages on the server queues to meet new demand or improve response time, for example, we simply add a new server processing node (more later on exactly what this means). Moreover, an MQSeries parallel message processing system, as Figure 1 illustrates, is highly resilient to failure, since the ability to process messages is shared throughout the system and there is no single point of failure.

It's worth pointing out that, in many cases, the availability of such resilient message processing systems is more important than the ability to add capacity dynamically.

## MQSERIES CLUSTERING

Different parallel processing architectures contain differing amounts of symmetry between the processing nodes. We know that MQSeries' core strengths include its support for a huge range of operating system platforms and network protocols. This means that when we consider a cluster of MQSeries queue managers we're considering the most generalized of parallel architectures, involving queue managers running on different processors types and different operating systems connected together using different network protocols.

MQSeries clustering is often referred to as a 'shared nothing' architecture. This is because the messages processed by an application connected to a server queue manager are localized to that server queue manager; messages are not shared between server queue managers. In the case of clustering, applications connected to client queue managers choose one of the set of queue managers hosting a clustered local queue and the message is sent over a channel to that queue manager. The message is typically processed by the server application and a response message indicating that processing is complete is returned to the client (the ReplyToQueue on the ReplyToQueueManager). MQSeries clustering therefore allows an application to locate any clustered queue or clustered queue manager within an MQSeries network and contains a significant amount of dynamic resource discovery to support these operations.

The shared nothing architecture has advantages and disadvantages. In its favour, applications in a shared nothing architecture behave exactly as they would in a simple one-to-one architecture, except that now there are many more application instances available to process messages. Also, because

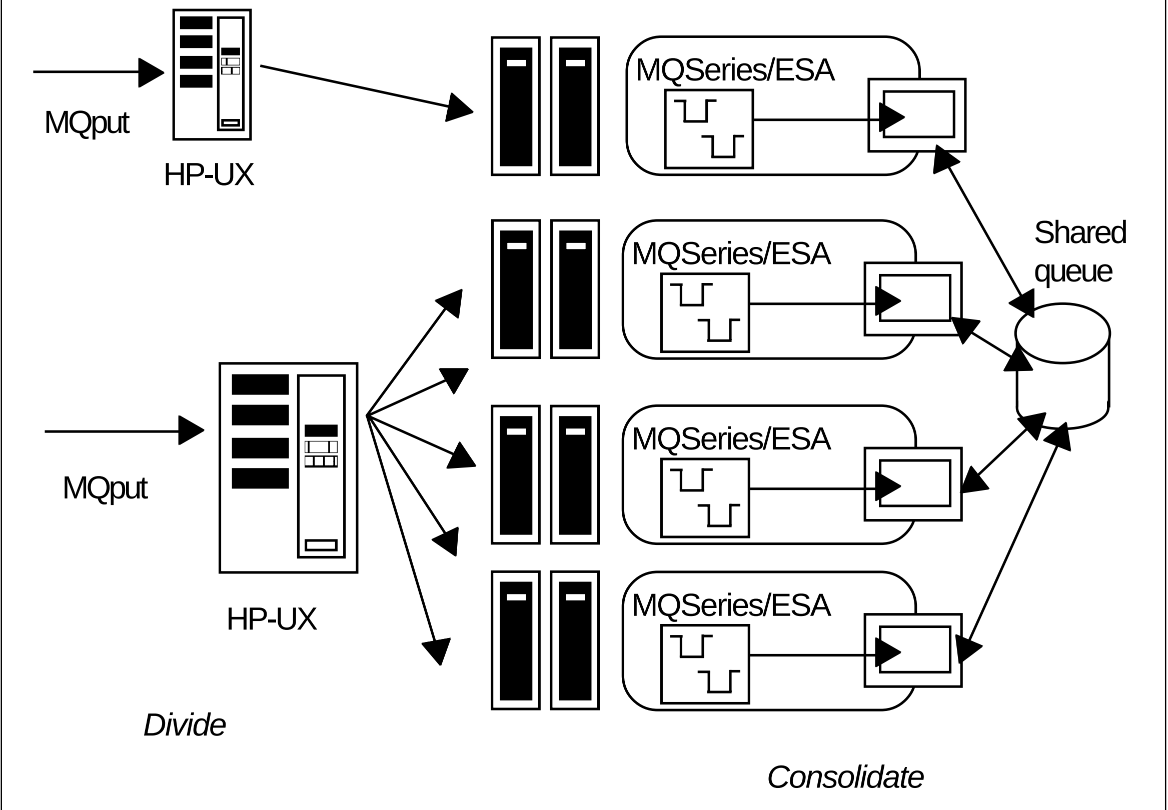


*Figure 1: MQSeries parallel message processing system*

nothing is shared between the server application processing engines the performance scales almost perfectly – so you'll see a 100-server node cluster process messages ten times as fast as an application in an existing MQSeries application architecture.

There's less chance you'll have to change your applications because, as nothing is shared, they can behave exactly as they did when they were the only application server in town. The downside of clustering is also a direct consequence of the shared nothing architecture, which is that, if a server queue manager fails – because of a power failure, for example – messages are trapped (not lost!) within the failed queue manager's datasets until the queue manager is restarted. And for a certain class of users, message availability, representing business availability, can be just as important as scalability of processing. For example, it may represent liquidity of assets in the case of a funds transfer, or response time in the case of human interaction.

## MQSERIES SHARED QUEUES

MQSeries shared queues address many of the same issues associated with parallel processing through use of the z/OS Coupling Facility (CF) and associated networking technologies. In direct contrast to clustering, messages on shared local queues are accessible by all queue managers in a queue sharing group (QSG). This group is intimately associated with the z/OS Coupling Facility. The Coupling Facility provides list structure objects and an associated set of programming interfaces which MQSeries for z/OS uses to store messages and support the MQI for these shared messages (see Figure 2).

When we want to add application processing capacity we can now share the message processing of a queue throughout the QSG by making a local queue shared between all the QSG's queue managers. A message put to a shared queue by one queue manager in the QSG is available to be consumed by any of the other queue managers within the QSG. Multiple instances of server applications can connect to many different queue managers to enable increased processing of messages.

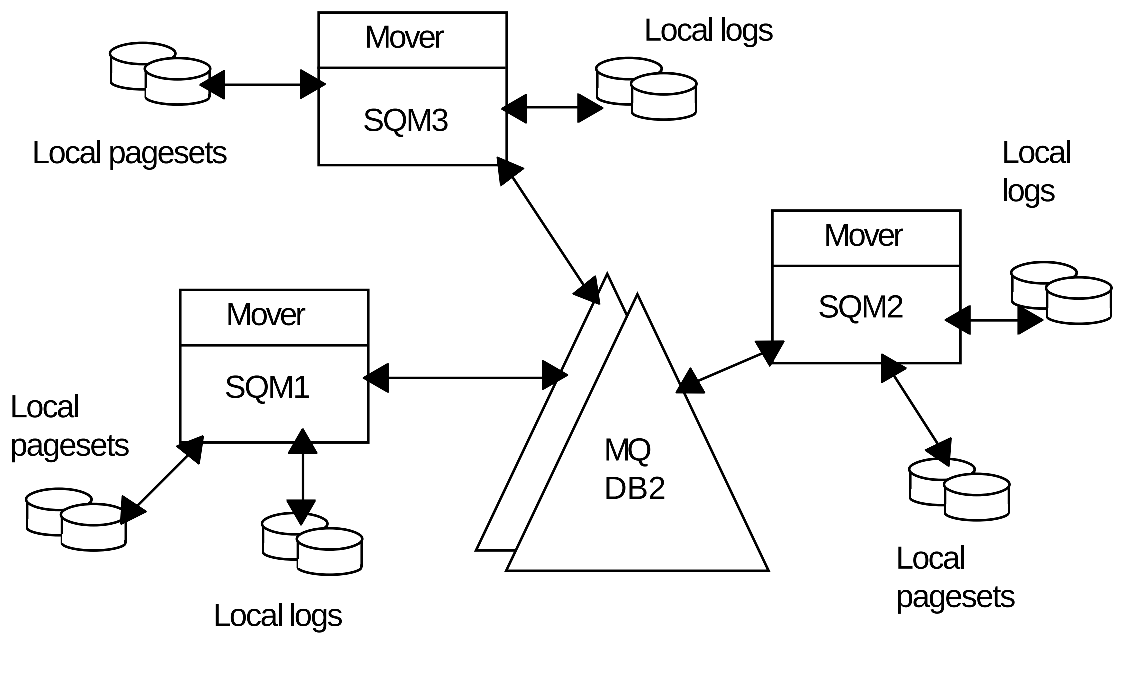


*Figure 2: Storing messages and supporting the MQI for shared messages*

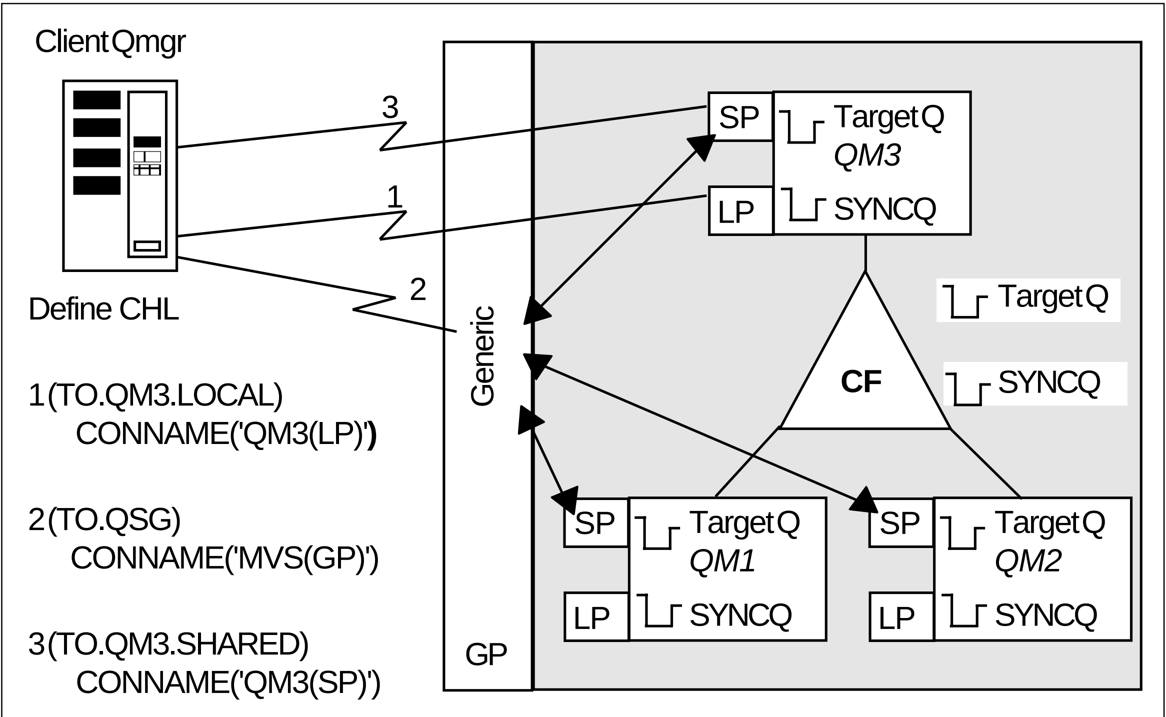


*Figure 3: Inbound architecture*

Shared queues have many of the advantages of clustering in that they provide increased processing capacity. However, they also have the advantage of recovering a message in failure scenarios. This comes directly from the fact that the message is shared, which allows queue managers to recover and make available immediately messages that are in-flight on a failing queue manager. When a queue manager fails, the operating system notifies all other queue managers in the queue-sharing group, enabling them to recover messages that were part of current transactions. This message-level availability can be very significant in any message processing scenarios, as mentioned earlier.

To support this processing at the server there is also a complementary set of network technologies to enable client queue managers (usually non-z/OS) to connect to any member of the QSG. This is necessary because, although messages are highly recoverable, once they are on queues we need to ensure that the queue-sharing group is highly visible and available from a network perspective. Channels on remote queue managers specify the network address of the queue-sharing group in the CONNAME attribute. This CONNAME is mapped using an appropriate technology (Sysplex

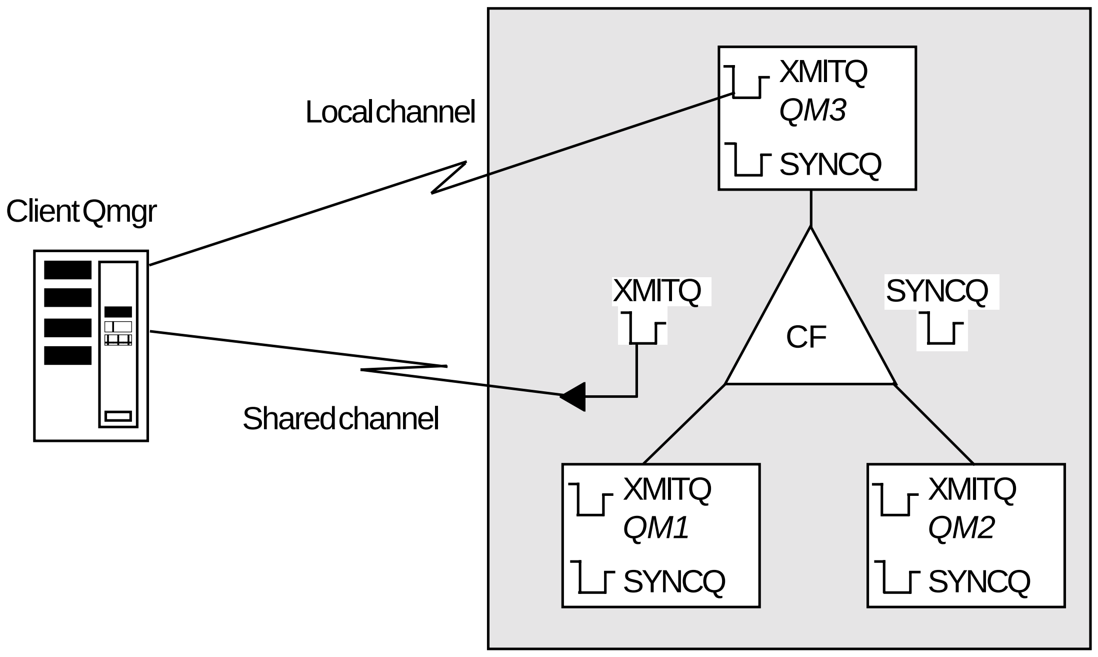


*Figure 4: Outbound architecture*

Distributor, Virtual IP addressing (VIPA), or Workload Manager/Domain Name System (WLM/DNS)), and a channel session from a network queue manager is established with a particular shared listener (new for V5.2) on a particular queue manager, as the inbound architecture illustrated in Figure 3 shows. In this way, if one network path isn't available to the queue sharing group, and hence to the server application, a path can be made through a different listener and queue manager.

Message processing responses are returned to the MQSeries network via a shared transmission queue (see Figure 4). With outbound responses it is important that failure of any one channel initiator does not inhibit return of the response message and that peer channel recovery allows the same sender (or server channel) to automatically start on a different queue manager and take over message delivery to the same client queue manager as the previously failed channel.

Resynchronization is the cornerstone of once-only message delivery and channel resynchronization messages for inbound and outbound channels are now stored in shared sync queues. This means that channels in each direction can resynchronize with any member of the QSG. In summary, when combined, shared queues, shared listeners, and peer channel recovery provide a high-performance, highly scalable, and high-availability message processing system.

## MQSERIES DESIGN CONSIDERATIONS

Table 1 summarizes the main considerations for deciding which channel topology to use, depending upon application, business, and environmental criteria.

Application criteria encompass cases where the nature of the application (ie whether it holds state or not) or the size of the message it receives or generates affects the infrastructure design.

Business criteria encompass the balance between the cost of application downtime and/or the cost of delayed (or trapped) messages against the cost of the overall installation.

Environmental criteria address the skills needed to support a particular implementation topology and those that are available within the organization.

## APPLICATION CONSIDERATIONS

We can see that MQSeries now offers a variety of ways to implement MQI applications on OS/390; the one you choose will depend upon the application considerations detailed below.

- The shared queue (messages in the Coupling Facility) support in MQ/390 V5.2 is only for non-persistent messages; V5.3 adds persistent message support. In both cases the message size is limited to 63 Kbytes.
  - As non-persistent messages on shared queues are not lost when the queue manager(s) using them are stopped, an application that is starting may process long-lived non-persistent messages. In some cases it may be inappropriate to use shared queues for an application because the messages do not ‘go away’ and an existing application rebuilds all transactions after a failure.
- Where affinities exist between the requester of a service and the service provider the implementation must be able to ensure that the

| **Consideration** | **Clustering channels** | **Generic channels** | **Pt-to-pt** |
|---|---|---|---|
| *Application* | | | |
| Persistent messages | Yes | Yes with V5.3 | Yes |
| Messages >63K (including replies) | Yes | No | Yes |
| Affinities (state retained by application) | Yes (Bind_on_open) | Only under application control | Yes |
| Large messages (say >1MB) | Yes but... Response problems may occur if these are mixed with short messages. Plus impact of running multiple 'large message' channels on sending box must be considered. Plus, do I want to clone applications which handle large messages? | No | Yes |
| Application style | Real-time and batch-oriented | Real-time | Real-time or batch-oriented |
| *Business* | | | |
| V high availability app req'd (inc failure tolerance) | Yes but... Messages may be trapped | Yes | No |
| Cloned application | Required | Required | Not required |
| *Environmental* | | | |
| Generic access used elsewhere in Sysplex | n/a | Skills already developed | n/a |
| Clustering used elsewhere | Skills already developed | n/a | n/a |
| Number inbound channels | High number of message sources may use resources inefficiently or require use of workload exit in all message sources | Workload may not be balanced if there is a low number of message sources | n/a |
| Scope | Multi-platform implementation; system-wide architecture | Shared queues reside on OS/390 only, but are accessed using normal MQSeries channel definitions | Normal MQSeries channel definitions |

*Table 1: Deciding which channel topology to use*

requester's messages are always received by a server that has the state available. This is the case if point-to-point channels are used, all messages from the requester being routed to the same server.

- Clustering can handle application affinities without application changes by use of the Bind_on_Open options.
- A shared queue implementation may require application changes to handle affinities and take advantage of parallel processing.
- Though a shared queue solution can support a failover mode of working by having the application queue opened for exclusive use, it is important to remember that at the point of failover other application states may need to be maintainted to provide application-level peer recovery.

• Processing of large messages in an MQSeries network often necessitates special design considerations (eg separate channels) that should be reviewed before simply transferring an application architecture to a parallel implementation. For example, in an image archiving system, immediate message availability may not be of prime importance.

• As Coupling Facility storage is a comparatively expensive resource of limited maximum size the total amount of storage that may be occupied by messages on shared queues must be considered. The CF is shared between the operating system components and the application subsystems using data sharing, eg DB2, CICS, IMS.

MQSeries applications designed to gather messages for later processing (ie batch-oriented applications) can store large amounts of message data before the batch processing is started. As this type of application is not response-time critical, delivering the messages via point-to-point channels or via clustered channels is appropriate.

For real-time applications the amount of message data that may arrive during an application outage, say for update or because of failure, needs to be considered. This may lead to choosing a solution that requires additional CF storage or one which uses clustering and can manage trapped messages.

## BUSINESS CONSIDERATIONS

If very high availability (say >99.99%) is needed considerable effort will be required to ensure that overall implementation will allow redundancy of all components of the system. This activity is expensive in design, hardware, and software and should only be undertaken for business-critical applications. The MQSeries aspects of this design will certainly be secondary to the application design considerations and probably secondary to the server program, whether it be in CICS, IMS, or any other OS/390 environment.

## ENVIRONMENTAL CONSIDERATIONS

The items in this section highlight cases where an MQSeries solution can be achieved by exploiting different MQSeries features, using the specific solution which uses skills already available in the enterprise or which might be acquired for other reasons. In a design where the business consequences of trapped messages mandate the use of shared queues the channel implementation chosen may be either clustering or generic channels; this decision may be based on the technical skills available within the IT department. For instance, if clustering is used for an existing application, a bias towards a clustering solution is to be expected. Whereas if a virtual IP addressing technique is in use in another application a solution based on generic channels would be the first approach.

The multiplicity of choices allows MQSeries to support various server application architectures, which are often the driving force of the overall system design.

## RECOMMENDATIONS

The authors believe that any large-scale MQSeries user will exploit varying quantities of these three technologies according to the requirements of the enterprise and specific business applications. Specifically:

- The solution will not be implemented as a ‘big bang’ but as a phased migration of existing applications to the new environment, which will be shaped by the major application provider (eg CICS or IMS) rather than by the middleware (MQSeries).

- Solutions which are running successfully as far as response, availability, and reliability are concerned should have a lower priority for migration. If significant application changes may be required to remove affinities it may be appropriate to leave these unchanged.
- In general, large messages and the applications that consume them are better run in a controlled environment (single designated machine and separate MQSeries channels). MQSeries Source IP Addressing function (introduced in V5.3) and Sysplex services such as Dynamic VIPA or Sysplex Distributor could be used to facilitate automated switch-over of this type of application in the event of failure.
- Clustering can handle larger messages than Shared Queues/Generic Channels; if the maximum size of a reply message from a legacy application cannot be guaranteed to be less than 63K it may be better to use clustering than to redesign the application.
- Only a shared queue solution can assure that no messages are trapped by being held on a queue that cannot be served until an environment is restarted.
- With only a few message sources, for example where branch office clients are supported on a box which communicates with the OS/390 application or where a Unix application and an OS/390 application are exchanging messages, the use of generic ports may not give workload balancing of messsages; rather, it is the communications sessions that are balanced across the members of the QSG. However, as it is generally true that the cost of message delivery is much less than the work that they generate, this may not be a significant issue.

## CONCLUSION

With the introduction of clustering and shared queues MQSeries has the flexibility to underpin successfully most application topologies required to support business applications in an OS/390 environment.

*John B Jones, BSc, MSc, MIEE, CEng*
*Anthony J O'Dowd, BSc, Dip Phys, PhD.*
*IBM Hursley (UK)*

## MQ news

IBM has begun selling a business integration package from New Era of Networks that includes adapters for MQSeries Integrator and WebSphere MQ Integrator. It features a new price structure and reduced prices, an extended range of adapters and platform support, and withdrawal of obsolete adapters.

New Era's adapters connect WebSphere MQIntegrator with specific off-the-shelf applications using industry standards, including SAP R/3 applications to DB/2, Oracle, and SQLServer OLTP environments, and complement those already available from IBM.

The NEON Adapter for EDI adds EDI capabilities into electronic application integration architectures and, when used with WebSphere MQ Integrator or MQSeries Integrator, allows sites to exchange EDI business documents with trading partners from an EAI server.

*For more information contact your local IBM representative.*
URL: http://www-4.ibm.com/software/ts/mqseries

* * *

Dirig Software has released a Specific Application Manager (SAM) for IBM WebSphere MQ. Dirig's WebSphere MQ SAM provides preconfigured rules, policies, thresholds, and performance-activated alerts to help administrators manage WebSphere MQ transactions, messages, and queues.

WebSphere MQ SAM works in conjunction with Dirig agents to provide data statistics that can help IT personnel maintain optimal performance levels of WebSphere MQ environments. Dirig offers additional management capabilities for managing WebSphere applications, DB2 databases, and other e-business applications.

*For more information contact:*
Dirig Software, One Indian Head Plaza, 6th Floor, Nashua, NH 03060, USA.
Tel: +1 603 889 2777.
Fax: +1 603 889 2471.
URL: http://www.dirig.com

* * *

MQSoftware has released a new version of its Q Pasa! middleware management software. Version 3 includes a Java-based management console that provides better integration, enhanced usability, and easier management of event and history rules. It also provides reusable templates to help users automate routine administration.

Q Pasa! 3.0 monitors WebSphere Application Server 4.0 and 5.0 and supports CrossWorlds, WebSphere Business Integrator, and WebSphere MQEveryplace.

*For more information contact:*
MQSoftware,1660 South Highway 100, Suite 400, Minneapolis, Minnesota 55416, USA.
Tel: +1 952 3458720.
Fax: +1 952 345 8721.

MQSoftware, Surrey Technology Centre, 40 Occam Road, Surrey Research Park, Guildford, Surrey, GU2 7YG, UK.
Tel: +44 1483 295400.
Fax: +44 1483 573704.

* * *

xephon