MQ update

63

September 2004

In this issue

MQ Update

Published by
Xephon Inc
PO Box 550547
Dallas, Texas 75355
USA

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

Editor
Trevor Eddolls
E-mail: trevore@xephon.com

Publisher
Nicole Thomas
E-mail: nicole@xephon.com

Disclaimer

Subscriptions and back-issues

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

Contributions

When Xephon is given copyright, articles published in *MQ Update* are paid for at the rate of $160 (£100 outside North America) per 1000 words and $80 (£50) per 100 lines of code for the first 200 lines of original material. The remaining code is paid for at the rate of $32 (£20) per 100 lines. To find out more about contributing an article, without any obligation, please download a copy of our *Notes for Contributors* from www.xephon.com/nfc.

MQ Update on-line

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

MQ Messaging using SQL

With DB2 Version 8, MQ messaging operations can be performed using SQL statements. This feature mainly involves User Defined Functions (UDFs) that invoke MQ using the MQ Application Messaging Interface (AMI). With a single SQL, we can read all the messages from a message queue and populate a database or a table, or extract the relevant columns from a specific row in a table and send out a message. Effectively this allows easy integration between the messaging system and the database.

MQ AMI

With MQSeries Version 2.2, applications on the mainframe can interface with MQ using the Application Messaging Interface (AMI). This is in addition to the IBM Message Queue Interface (MQI) and Java Messaging Service (JMS) and chiefly aims at abstracting the message handling to the middleware layer.

To send or receive a message using AMI, an application should specify the following:

1. What is being sent – the message itself in structured or free form, sent from one program to another.
2. Where the message is going to/coming from – the 'services' part, which typically points to a local or remote MQSeries queue. Services can also represent a distribution list managed by middleware.
3. How the message is to be handled – the 'policy' that encapsulates message options, like priority, timeout, number of retries, whether the operation is part of a unit of work.

Policies and services are typically stored in a repository external to the application and can be maintained without impacting the application code. The policies can be used by many applications and can be controlled at the enterprise level.

DB2 AND MQ AMI

DB2 provides UDF for handling the following capabilities of WebSphere MQ:

- Send and forget
- Read or receive
- Request and response
- Publish and subscribe.

In line with the AMI requirements, the key parameters for the MQ UDFs are:

1. Msg-data – message in a CLOB/VARCHAR variable.
2. Send-service – identifying the location to which the message has to be sent.
3. Receive-service – identifying the location from which the message has to be read.
4. Topic-list – giving the list of topics corresponding to the message and which are used in functions related to publish and subscribe.
5. Publish-service – specifying the MQ publisher to whom the message has to be published.
6. Subscriber-service – specifying the logical destination for the messages corresponding to the topic.
7. Correl-ID – correlation identifier used for associating the response with the requests.

Note that WebSphere MQ functions have to be registered in the DB2 database server before they can be invoked via SQL (see the *DB2 Installation Guide* for details of setting this up).

For example, the following is the SQL used to read a message from a queue:

```
SELECT MQREAD('MYSERVICE','MYPOLICY') FROM TABLE;
```

where MQREAD is the MQ UDF, MYSERVICE gives the location from which the message has to be read, and MYPOLICY gives details of how the message has to be handled.

MQ UDF SCALAR FUNCTIONS

The following is a list of MQ scalar functions:

- MQSEND – sends the message to the location corresponding to the send-service. For request-response mode, the correlation identifier has to be specified.
- MQREAD – returns the message from the head of the queue without removing it from the queue.
- MQREADCLOB – similar to MQREAD; instead of VARCHAR, the message is returned as a CLOB.
- MQRECEIVE – returns the message from the head of the queue and also removes it from the queue. If in request-response mode, the correlation identifier has to be specified if the corresponding message is to be returned.
- MQRECEIVECLOB – similar to MQRECEIVE. Instead of VARCHAR, the message is returned as a CLOB.
- MQPUBLISH – publishes the message to the publisher corresponding to the service.
- MQSUBSCRIBE – allows the user to subscribe to the message corresponding to a topic list.
- MQUNSUBCRIBE – allows the user to unsubscribe from the topic list.

The following are some points worth noting:

- If the service or policy is not specified in the MQ functions, DB2.DEFAULT.SERVICE and DB2.DEFAULT.POLICY are used.
- Unlike RECEIVE functions, READ functions do not remove the message from the queue.

- In the case of VARCHAR, the maximum size is 4,000 bytes, whereas CLOB allows a maximum of 1MB.
- If a correlation identifier is specified, the RECEIVE command returns the message that matches it. Otherwise, the message from the head of the queue is returned.
- READ and RECEIVE functions return a NULL value if no message is available. All other functions return a value of 1 if successful and 0 if not successful.
- The topic-list can include multiple topics delimited by a colon (eg topic 1:topic 2:topic 3).

MQ UDF – TABLE FUNCTIONS

MQ messages can be accessed by an application as if it were a DB2 table, using the MQ table functions.

For example, all the messages from the queue can be returned as a materialized DB2 table using the MQREADALL table function:

```
SELECT * FROM TABLE (MQREADALL()) T;
```

By specifying the number of rows, the function can be used to return the required number of rows only. Along with the message, the meta-data is also returned.

The format of the resultant table is as follows (showing column then data type):

- MSG – VARCHAR(4000)
- CORRELID – VARCHAR(24)
- TOPIC – VARCHAR(40)
- QNAME – VARCHAR(48)
- MSGID – CHAR(24)
- MSGFORMAT – VARCHAR(8).

The other table functions are MQREADALLCLOB, MQRECEIVEALL, and MQRECEIVEALLCLOB .

COMMIT ENVIRONMENT FOR MQ FUNCTIONS

DB2 provides two flavours of these MQSeries functions:

- One supports only single-phase commit and is identified by a schema name of DB2MQ1C.
- The other supports two-phase commit and is identified by a schema name of DB2MQ2C.

With single-phase commit, the DB commit or rollback operations are independent of MQ operations. For example, when the DB2 transaction is rolled back, the messages sent to the queue are not discarded. This would be quite useful when your application needs to roll back the database operations but still send the notification to the user using an MQ message.

In the case of two-phase commit, RRS acts as the transaction coordinator. When the DB operations are rolled back, the messages sent as part of a UOW are also discarded.

Note that WLM has to be customized for running MQSeries UDF support and each flavour of the functions should run under a separate WLM environment.

Sasirekha Cota
Tata Consultancy Services (India)

Finding out remote queue manager names

INTRODUCTION

Large WebSphere MQ (WMQ) networks usually have several local system administrators (first and second-level Unix and Windows application support) and centralized specialists for base products (eg third-level WMQ support). The local administrators may have some basic WMQ knowledge. This knowledge may be sufficient to manage queue managers during normal operations (like starting and stopping) and to create some WMQ objects (such as queues or a channel). But in general they will not have enough know-how to solve problems like channels being in retry mode, performance problems, looking for missing messages, and so on.

Perhaps one of the local administrators knows a little more about WMQ than their colleagues in the team, but he or she could be out of the office temporarily or on vacation when a problem occurs. In such situations, or in the case of trouble-shooting, the local administrators have to call the specialists. These specialists usually need access to the WMQ queue manager to solve the problem. In a well-documented environment, maybe the specialists open a list and get the connection data for the queue manager, connect to it using GUI tools like MQExplorer or MQMON (IBM SupportPac MO71), and solve the problem.

Unfortunately, real life is not that easy; such a queue manager list may not exist or not be up-to-date. In this case, the WMQ specialists have to ask for the name of the queue manager, IP address, and port to connect to it, eg with MQMON. Possibly the local administrator will not know these parameters or even what a queue manager is. The WMQ specialist may now guide the administrator in finding out these parameters, but often it would be much easier for the specialist to do this by themself. A solution to this problem is described in this article.

DESCRIPTION OF THE TOOL

Before using FindQManager

It is quite easy to find out the IP address of the system running a WMQ queue manager. Any system administrator will know the IP addresses of the machines that s/he is responsible for. Even operators or developers would at least know the DNS name of these machines. It is a little more difficult to find out the WMQ listener port – but using the WMQ default port of 1414 or some value near it (let's say 1415 or 1416) or another well-known port range of your company, could be helpful in many cases.

The last parameter a WMQ specialist has to know is the name of the WMQ queue manager. Asking the local administrators is not always an option. Using the known or guessed parameters above, the WMQ specialists may now find out the connection attributes themselves by using the tool FindQManager. If this tool is successful, it prints out a valid connect string, which can be used to set up a GUI tool like MQMON for WMQ administration.

How FindQManager works

The program FindQManager was developed in Java and finds out the name of a remote queue manager via PCF commands. You have to know (or guess) the IP address and port of the WMQ listener and the tool tries to find out the name of the remote queue manager. The connect string used by FindQManager is built from the three command line parameters – address, port, and channel. A script called findqmgr(.cmd) calls this Java program using different input parameters. In the examples below, it tries the channels MY.ADMIN.SVRCONN, SYSTEM.ADMIN.SVRCONN, and SYSTEM.DEF.SVRCONN.

If a connection is successful, the script prints out the name of the queue manager and the successful connect string. Assuming that most WMQ installations on client/server platforms use the default port 1414 and have a channel SYSTEM.ADMIN.SVRCONN defined or at least the channel SYSTEM.DEF.SVRCONN defined and enabled, this tool finds out the names of most of the queue managers in the network.

The script findqmgr(.cmd) may be called with at least an IP address or DNS name, and optionally one or more port numbers. If no port number is specified, the script tries the default WMQ port 1414. Otherwise, it loops over all the specified port numbers and tries to connect to a WMQ listener. The script prints out the first successful connect string for each specified port number.

Building the class file

Use the Java compiler to create the class file. It should not matter where you compile or execute it (because it's Java!). I compiled it on AIX and executed it on Windows and AIX.

Install javac FindQManager.java

This is how to install FindQManager.

First you need Java support for WMQ. This is integrated within WMQ 5.3. If you use Version 5.2 or lower you need to install the IBM SupportPac MA88.

To use PCF commands with Java, you have to install the Java classes provided with IBM SupportPac MS0B. Copy the file com.ibm.mq.pcf.jar to the WMQ Java path, eg <WMQ_Installpath>\Java\lib for WMQ 5.3 on Windows.

The next step is to copy the file FindQManager.class to the same Java path. You may have to define some system environment parameters.

```
- set MQ_JAVA_DATA_PATH=<WMQ_Installpath>
```

eg 'C:\Program files\IBM\WebSphere MQ'.

```
- set MQ_JAVA_INSTALL_PATH= <WMQ_Installpath>\Java

- set CLASSPATH=
<WMQ_Installpath>\Java\lib;<WMQ_Installpath>\Java\lib\com.ibm.mq.pcf.jar;
<WMQ_Installpath>\Java\lib\providerutil.jar;<WMQ_Installpath>\Java\lib\com.ibm.mqjms.jar;
<WMQ_Installpath>\Java\lib\ldap.jar;<WMQ_Installpath>\Java\lib\jta.jar;
<WMQ_Installpath>\Java\lib\jndi.jar;<WMQ_Installpath>\Java\lib\jms.jar;
<WMQ_Installpath>\Java\lib\connector.jar;<WMQ_Installpath>\Java\lib\fscontext.jar;
<WMQ_Installpath>\Java\lib;<WMQ_Installpath>\Java\lib\com.ibm.mq.jar
```

Last but not least, copy the file findqmgr.cmd (or findqmgr on Unix) into a local directory (eg C: or /home/mqm). You may now run this script within a command shell.

Call FindQManager

It is possible to call the Java program directly from the command line, but I wrote a script, which is easier to customize, and this script itself calls the Java program. I created a Windows and a Unix version of the script. Call the script by issuing the following command:

```
findqmgr <IP address or DNS name> [<port (optional)>]
```

Afterwards you will see the name of the queue manager and the connect string, or an error message with possible reasons why this information could not be found.

DESCRIPTION OF THE CODE

The code consists of three parts, the Java-Code FindQManager.java, a Windows command script called findqmgr.cmd (which calls the Java class), and a Korn shell script, findqmgr (which works in the same way as the Windows script, but on Unix systems).

Description of the scripts

First the script initializes the parameter SCRIPT_ERRORS. This parameter indicates that, for each specified port, a queue manager has been found (value NO), or at least one port could not be associated with a queue manager (value YES).

The next step of the script is to check whether at least an IP address or DNS name has been specified. If not, the script ends with an error message. When an address has been specified, the script checks further command line parameters. These will be interpreted as port numbers and tested one by one. If no port number is specified, the script uses the WMQ default port 1414.

Now the script calls the Java program FindQManager, using the first SVRCONN channel, and redirects the output to the file output.txt. Afterwards the script searches in this output file for the string ‘Connect string’ and prints it. When this string has been found and the Java program has run successfully, the name of the queue manager is read from the output file.

If the Java program is unsuccessful, the script calls it again using the next channel in the list. An error message is displayed if no connection can be established with any of the channels. Otherwise the first successful connect string is printed and the script checks the next port number. If at least one of the address/port pairs fails, a number of possible reasons are shown when the script has finished.

Description of the Java code

The Java code is quite simple. It requires exactly three command line parameters: host address, IP port, and SVRCONN channel. The program sets up a connect string with the format:

```
<SVRCONN channel>/TCP/<host address>(<IP port>)
```

It tries to connect with this connect string and sends an inquire queue manager command. If successful, the connect string and the name of the WMQ queue manager are printed out.

Listing of findqmgr.cmd

```
@echo off

REM
REM Copy this file to a local directory (eg C:) and run it in a cmd
REM   shell.
REM
REM Run this script now with:
REM
REM    C:\findqmgr <IP address or DNS name> [<port> [<port> [...]]]
REM
REM One or more ports are optional. If no port is specified, the MQ
REM default port 1414 will be used.
REM
set SCRIPT_ERRORS=no
REM
REM Check the command line parameters - at least an IP address or DNS
name.
REM
if "%1" == "" goto script_error

set ADDRESS=%1
shift 1

set PORT=1414

if "%1" == "" goto check_address
set PORT=%1

:check_address
REM
REM Try first a connect using channel MY.ADMIN.SVRCONN.
REM
java FindQManager %ADDRESS% %PORT% MY.ADMIN.SVRCONN > output.txt 2>
error.txt
```

```
findstr "Connect string" output.txt

if %ERRORLEVEL% == 0 goto finish
REM
REM First connect was unsuccessful, try now channel
SYSTEM.ADMIN.SVRCONN.
REM
java FindQManager %ADDRESS% %PORT% SYSTEM.ADMIN.SVRCONN > output.txt 2>
error.txt

findstr "Connect string" output.txt

if %ERRORLEVEL% == 0 goto finish
REM
REM First connect was unsuccessful, try now channel SYSTEM.DEF.SVRCONN.
REM
java FindQManager %ADDRESS% %PORT% SYSTEM.DEF.SVRCONN > output.txt 2>
error.txt

findstr "Connect string" output.txt

if %ERRORLEVEL% == 0 goto finish
REM
REM Error output - no connection has been established.
REM
echo No queue manager information has been found for following address:
echo  IP address: %ADDRESS%
echo  IP port: %PORT%

set SCRIPT_ERRORS=yes

goto next_address

:script_error
REM
REM Error output - missing command line parameter.
REM
echo "usage: %0 <IP address or DNS name> [<port> [<port [...]]]"

goto script_exit

:finish
REM
REM Find the string "Queue manager" in the PCF output.
REM
findstr "Queue manager" output.txt

del output.txt error.txt

:next_address
```

```
shift 1

if "%1" == "" goto script_finish

set PORT=%1

goto check_address

:script_finish

if "%SCRIPT_ERRORS" == "no" goto script_exit

echo One or more errors occurred. Check the following items with
echo the local system administrators:
echo  - is there a queue manager running
echo  - is a listener running using this port
echo  - is the command server started
echo  - are the channels locked or removed
echo  - is a TCP wrapper running
echo  - is a security exit installed
echo  - is there a firewall to pass

:script_exit
```

Listing of findqmgr

```
#!/bin/ksh
#
# Copy this file to a local directory (eg $HOME) and run it in a shell.
#
# Run this script now with:
#
#    $HOME/findqmgr <IP address or DNS name> [<port> [<port> [...]]]
#
# One or more ports are optional. If no port is specified, the MQ
# default port 1414 will be used.
#
SCRIPT_ERRORS="no"
#
# Check the command line parameters - at least an IP address or DNS
name.
#
if [ "$1" = "" ]
then
   # Error output - missing command line parameter.
   echo "usage: $0 <IP address or DNS name> [<port> [<port> [...]]]"

   exit 1
```

```
fi

ADDRESS=$1
shift 1

PORT=1414

while [ "$1" != "" ]
do
   found="no"
   PORT=$1
   shift 1

    SVRCONN_LIST="MY.ADMIN.SVRCONN SYSTEM.ADMIN.SVRCONN
SYSTEM.DEF.SVRCONN"
   #
   # Try now to connect using one SVRCONN channel.
   # Print out the connection string and queue manager, if found.
   #
   for channel in $SVRCONN_LIST
   do
      if [ "$found" != "yes" ]
      then
     java FindQManager $ADDRESS $PORT $channel > output.txt 2> error.txt

         CONNECT_STRING='grep "Connect string" output.txt'
         QUEUE_MANAGER='grep "Queue manager" output.txt'

         rm output.txt error.txt

         if [ "X$CONNECT_STRING" != "X" ]
         then
            echo
            echo $CONNECT_STRING
            echo $QUEUE_MANAGER

            found="yes"
         fi
      fi
   done
   #
   # Error output - no connection has been established.
   #
   if [ "$found" = "no" ]
   then
      cat <<EOF

No queue manager information has been found for following address:
 IP address: $ADDRESS
 IP port: $PORT
EOF

      SCRIPT_ERRORS="yes"
   fi
done

if [ $SCRIPT_ERRORS = "yes" ]
then
   cat <<EOF

One or more errors occurred. Check the following items with
the local system administrators:

 - is there a queue manager running
 - is a listener running using this port
 - is the command server started
 - are the channels locked or removed
 - is a TCP wrapper running
 - is a security exit installed
 - is there a firewall to pass
EOF
   fi

echo

exit 1
```

Listing of FindQManager.java

```
/**********************************************************************/
/* October, 16th 2003                                                 */
/* Hubert Kleinmanns                                                  */
/* Senior Consultant                                                  */
/**********************************************************************/
/* Purpose: This program is meant to find out the name of a queue     */
/* manager.                                                           */
/*                                                                    */
/* Run this program:                                                  */
/*                                                                    */
/*    Required input parameters:                                      */
/*       1. DNS name or IP address                                    */
/*       2. WebSphere MQ port                                         */
/*       3. Channel of type SVRCONN                                   */
/*                                                                    */
/*    Output text:                                                    */
/*       1. Address (like the MQSERVER environment)                   */
/*       2. Name of the queue manager                                 */
/**********************************************************************/
```

```
import java.io.*;
import com.ibm.mq.*;
import com.ibm.mq.pcf.*;
/**
 * PCF example class showing use of PCFAgent and com.ibm.mq.pcf
 * structure types to generate and parse a PCF query.
 *
 */
public class FindQManager
{
   public static void main (String[] args)
   {
      int[]          qmgrAttributes =
      {
         CMQCFC.MQIACF_ALL
      };
      PCFAgent       pcfAgent;
      PCFParameter[] qmgrParameters =
      {
         new MQCFIL (CMQCFC.MQIACF_Q_MGR_ATTRS, qmgrAttributes)
      };
      MQMessage[]    answer;
      MQCFH          pcfHeader;
      PCFParameter   qmgrPar;
      String         connectString;

      try
      {
         // Connect a PCFAgent to the specified queue manager
         if (args.length == 3)
         {
            connectString = args[2] + "/TCP/'" + args[Ø] +"(" +
               args[1] + ")'";

           pcfAgent = new PCFAgent (args[Ø], Integer.parseInt (args[1]),
               args[2]);

            // Use the PCF agent to send the request
            answer = pcfAgent.send (CMQCFC.MQCMD_INQUIRE_Q_MGR,
               qmgrParameters);

            pcfHeader = new MQCFH (answer[Ø]);

           // Check the PCF header (MQCFH) in the first response message
            if (pcfHeader.reason == Ø)
            {
               for (int i = Ø; i < pcfHeader.parameterCount; i++)
               {
```

```
               // Walk through the returned attributes
               qmgrPar = PCFParameter.nextParameter (answer[Ø]);

               if (qmgrPar.getParameter() == CMQC.MQCA_Q_MGR_NAME)
               {
             System.out.println ("Connect string: " + connectString);
                  System.out.println ("Queue manager: " +
                     qmgrPar.getValue ());
               }
            }
         }
         else
         {
            System.out.println ("PCF error:\n" + pcfHeader);

         // Walk through the returned parameters describing the error
            for (int i = Ø; i < pcfHeader.parameterCount; i++)
            {
         System.out.println (PCFParameter.nextParameter (answer[Ø]));
            }
         }

         // Disconnect
         pcfAgent.disconnect ();
      }
      else
      {
         System.out.println ("Missing parameter:\n");
System.out.println ("usage:\t" + args[Ø] + "address port channel");
         System.out.println ("\t\taddress: DNS name or IP address.");
         System.out.println ("\t\tport: WMQ listener port.");
        System.out.println ("\t\tchannel: Channel of type SVRCONN.");
      }
   }

   catch (ArrayIndexOutOfBoundsException abe)
   {
      System.out.println ("ArrayIndexOutOfBoundsException");
   }

   catch (NumberFormatException nfe)
   {
      System.out.println ("NumberFormatException");
   }

   catch (MQException mqe)
   {
      System.out.println ("MQException");
   }
```

```
        catch (IOException ioe)
        {
           System.out.println ("IOException");
        }
     }
  }
```

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

Monitoring WebSphere MQ Integrator Broker message flows

This article describes different options to monitor the processing of message flows in WebSphere MQ Integrator Broker. It provides an overview of the possibilities and describes what kind of monitoring can be achieved.

ABSTRACT

The monitoring of message flows covers a wide area. It starts with checking the availability of processes and resource managers needed for message flow processing. It spans from the checking of processing characteristics to monitoring the status of individual messages. Monitoring can be divided into system monitoring performed by IT support staff and business process monitoring performed by business operations staff.

System monitoring checks the healthiness and processing characteristics of the resources involved in the message flow processing. Business process monitoring checks the health and state of a business process. This can be done post-mortem using reporting of consolidated business characteristics, or in real-time for individual messages.

This article provides just an overview of different monitoring methods. Depending on your requirements, you may need to

take a closer look at which methods you need and how to integrate them with your monitoring infrastructure. You may need to combine system monitoring and business process monitoring techniques to achieve your goals.

INTRODUCTION

Once you have developed a message flow and want to get it into production, you need to know how to monitor the message flow. But different people have different understandings of what monitoring is or they have different goals for their monitoring requirements. A wide set of requirements have been evaluated as part of a monitoring discussion for the product WebSphere Business Integration for Financial Networks (abbreviated to WebSphere BI for FN) on z/OS (for product details see http://www.ibm.com/software/integration/wbifn). Most of the discussions apply to all other WebSphere MQ Integrator Broker message flows and WebSphere MQ Integrator Broker supported platforms. WebSphere BI for FN consists of an infrastructure that allows you to deliver products on top of WebSphere MQ Integrator Broker and extensions that exploit the infrastructure and deliver access to different financial networks, for example the Secure IP Network provided by SWIFT (for details about SWIFT see http://www.swift.com).

Based on the requirements, two different general kinds of monitoring option have been identified: IT or system monitoring and business process monitoring, or business monitoring for short. The following sections describe the different kinds of monitoring followed by possible ways to achieve them.

SYSTEM MONITORING

System monitoring requirements are about checking the health of a system. ‘Health’ refers to whether the processing is working OK or not. The simplest forms of system monitoring are notifications or events if something unexpected happens. Higher requirements for system monitoring require knowing whether the processing takes place within a predefined set of parameters and raising an event if the parameters are not met. Such parameters could be informational. For example: ‘my average message processing rate is 10 messages per second, but between 10am and 11am I expect to process 25 messages per second; please inform me if the actual message rate differs by more than 20%.’

For system monitoring you are usually looking to those resources that are critical for delivering the function that is performed by the message flow. Resources besides the message flows themselves, for example databases or WebSphere MQ message queues, are used by the message flows to perform their function.

System monitoring is usually performed by IT support organizations. Once problems are detected, people from the IT support department need instructions about what to do to resolve the abnormal situation.

BUSINESS PROCESS MONITORING

Business process monitoring is much more sophisticated than system monitoring. It is about the healthiness of a business process. It can be divided into post-mortem analysis of the messages that passed the system and real-time business process monitoring. Post-mortem analysis is also called reporting. Reporting needs to summarize information from all information messages in a given time period.

Business processes usually span multiple steps, where a message flow could represent one step within the overall business process. Real-time business monitoring requirements are about monitoring individual messages. For a business process, it is necessary to know where in the processing a message currently is and what the current state of the processing step is. Other interesting information is, for example, whether a specific message reached a specified processing step within the specified time. If not, an event indicating this problem is required.

In contrast to system monitoring, business process monitoring is performed by business operations staff. Underlying business monitoring is the assumption that all IT components are working properly. Even if this is the case, the business processes can still have problems.

Some business processes consist of a series of steps that are monitored using system monitoring means. From these single monitoring results some tools are available that can deduce the state of a business process. From this you can imagine that the transition from system monitoring to business monitoring is fluid.

SYSTEM MONITORING OVERVIEW

This section provides an overview of system monitoring capabilities for message flows. System monitoring can be done at different levels:

- Operating system level
- Resource level
- Broker level
- Message flow level.

The following sections describe the monitoring capabilities at each level.

System monitoring at the operating system level

Message flows are processing within a broker. The broker itself and all resource managers that are needed to process the messages, for example WebSphere MQ queue managers for messages or a database manager for persistent data, are processes or jobs within an operating system. The existence of these processes and whether they consume CPU resources are already indications of the processing status of the processes. Depending on the operating system capabilities, it can be assumed that all required programs are up and running. For example, z/OS provides a component called Automated Restart Manager (ARM). This can assure you that all the required processes are up and running, or, if they stop for any reason, automatically restart the process. DB2, WebSphere MQ, and WebSphere MQ Integrator Broker support ARM. Other operating systems may provide similar capabilities.

For message flow processing, other applications are needed to send messages to the message flow and to consume messages produced by the message flow. These applications and their resource managers also need to be monitored to get a complete view of health at the operating system level.

System monitoring at the resource level

For message flow processing different resource managers can be involved. It is possible, for example, to monitor at the WebSphere MQ level, the database level, or any other resource manager level. The actual resources that can be monitored depend on the resources that your message flow accesses. The following paragraphs show examples of monitoring at the WebSphere MQ level and at the database level.

Already at the WebSphere MQ level, you have some monitoring options. Examples are that you can monitor the status of a WebSphere MQ queue manager, message queues, or channels. This can be done for example with a set of commands that are issued against the WebSphere MQ queue manager. On distributed environments, such a script can run using runmqsc, a program provided by WebSphere MQ. On z/OS, WebSphere MQ provides a similar program, called CSQUTIL.

Checking, for example, whether queues are open for input, meaning that an application has issued an MQOpen function to get messages from the queue, is a good indication of whether an application is available to process messages. It is an indication of whether the application is *actually* processing messages. For such purposes you can look at the number of messages in a message queue, represented by the message queue depth. If the queue depth is constantly increasing, this indicates that the application is not processing, or not processing enough, messages. If the queue depth is decreasing, this is usually a clear indication that messages are being processed.

Another way to monitor at the queue manager level is by exploiting WebSphere MQ events. A queue manager can issue three different kinds of event: queue manager, channel, and performance events. Queue manager and channel events normally report starting or stopping of these elements. An example of a performance event is the queue depth high event. For a message queue, the message queue high event is reported if the queue manager detects that the message queue contains more than a defined number of messages. For reporting, the queue manager issues an event message onto an event queue. For a complete list of events, their meaning, event queues, and how to enable events, please refer to the appropriate WebSphere MQ documentation.

For monitoring the events you can write an application that gets the event messages from the event queues. The event messages are coded in a WebSphere MQ defined self-defining data structure, called Programmable Command Format (PCF). Once interpreted, the events allow the detection of situations where the system is not working as expected.

To avoid writing your own program you can exploit already existing monitoring program products that are available on the market. Such programs can monitor a queue manager, its queues, and channels. Such products are available, for example, from IBM Tivoli, Candle, MQSoftware, and BMC. There is also a WebSphere MQ SupportPac, MS0K (which can be found at http://www.ibm.com/software/integration/support/supportpacs), which allows WebSphere MQ events to be displayed. To allow the monitoring of your message flow, you need to identify the queue managers, message queues, and channels that are required for your message flow to perform its function.

Similar to WebSphere MQ, you could also use monitoring tools for other resource managers to get an indication of the health of your system. If your message flow is doing database operations, eg WebSphere BI for FN has an audit functionality that writes entries into a database table, a database monitoring tool can provide information about the number of inserts into this table or find out whether tables are getting full. If no more inserts are made, when you expect that messages should be processing, you ought to look at your message flow processing. Various database monitoring tools are available on the market.

System monitoring at the WebSphere MQ Integrator Broker level

WebSphere MQ Integrator Broker in general is also a resource manager. It provides the run-time environments for message flow. These environments are execution groups in a message broker. Brokers, execution groups, and assignments of message flows to an execution group are done using the WebSphere MQ Integrator Broker Control Center. Once your message flows are deployed, the Control Center provides an operations view. In this window, the Control Center shows the actual status of brokers, execution groups, and message flows as known to the configuration manager. The status indicates whether a message flow is started or stopped. An example of such status is shown in Figure 1.

Using the operations pane, you get only started/stopped status information. If a message flow is shown as started this does not provide an indication of whether the message flow is really working and processing messages. A simple example is that

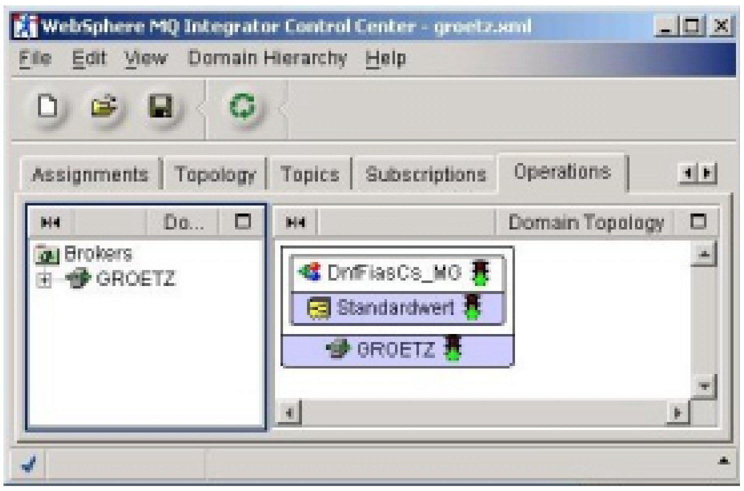


Figure 1: WebSphere MQ Integrator Broker Operations view

your message flow is deployed to the broker but you have not yet defined its input queue. In the operations pane the message flow is shown in green, meaning that the flow has started. To get further information about the processing of message flows you also need to look at the broker's error log. The error log is platform dependent, but on most operating systems the SYSLOG service is used to issue error messages. Looking at the error log provides you with information about processing errors that might have occurred. Usually the error log is de-central, meaning that you have to log-on to the machine on which the broker is running. If you have multiple brokers, for example for throughput or availability reasons, this could be a little cumbersome.

With WebSphere Business Integration Message Broker Version 5.0 (WebSphere BI MB), the follow-on version to the WebSphere MQ Integrator Broker product, the broker provides you with an additional function to monitor your message flow processing. This functionality is called statistics and accounting. Once enabled, the broker regularly provides statistics information about the processing of message flows. Such information includes, for example, the number of messages processed and the time needed to process the messages. For a detailed list of information provided by the statistics and accounting function of WebSphere BI MB, please refer to the appropriate documentation.

Accounting and statistics can be enabled for individual message flows, all message flows within an execution group, or the whole broker. The information is collected by the broker without any change to the message flow. A summary of the information is written periodically to one of up to three different locations. Possible destinations are the broker's user trace files, a broker-defined topic for publications, and Systems Management Facility (SMF) on z/OS. The broker allows two different kinds of statistics: archive statistics and snapshot statistics. Archive statistics are meant more for accounting purposes, while the snapshot statistics should be used to monitor the message processing of a message flow. Depending on the destination, different monitoring options are possible. Writing the information to the user trace or SMF is more for post-mortem analysis, while publication allows more real-time monitoring.

The statistics information provides good information about how many messages are processed within the statistics interval. But the broker provides just the data. You need either to develop a tool that displays the results or compares it with the expected values. There are already some monitoring products available that hook into the accounting and statistics data and display the results. An example for such a program is PathWAI from Candle.

System monitoring at the message flow level

You can also add monitoring to your message flows. Different methods are available. The simplest method is to participate in the broker's error event handling. To do this, you can produce message flow exceptions so that your processing detects the problems. These are then reported by the broker in the same way as the broker internal exceptions.

Such error events are usually written to a system error log. That's why such exceptions are not processed automatically. Some exceptions are of interest to monitoring programs, which react to them.

The product WebSphere BI for FN implemented a different approach, the main difference being that instead of broker exceptions, WebSphere BI for FN events are used. These are similar to broker exceptions but they are enriched with additional information to better categorize the events and allow the filtering of these events for monitoring. WebSphere BI for FN provides a set of nodes that allow the issuing of an event. One node that converts broker exceptions to WebSphere BI for FN events is included in this set. The events in WebSphere BI for FN are published using WebSphere BI for FN defined topics.

Once published, any subscriber can get the events. Based on the topic structure, a monitoring program can register for all events or just those events that are of interest to it. WebSphere BI for FN already provides a set of monitoring programs. One such program can be used to look on-line at the events that are of interest to an actual user. Another such monitoring program is a message flow that can subscribe to all interesting events. This message flow receives the events and writes them to the systems error log on a central system.

In addition to publishing the event using a WebSphere BI for FN defined topic, the events are also stored in an event database. An event administration message flow is provided by WebSphere BI for FN that allows users to list and maintain events within the event database. In contrast to on-line monitoring using published events, this approach allows post-mortem analysis of problems.

Some monitoring tools allow users to monitor message flow processing on-line. Such tools include, for example, IBM Tivoli Monitoring for Business Integration and BMC's Patrol for WebSphere MQ Integrator. To enable your flow to be monitored by most of these tools, you can include a special monitoring node in you message flow. Such nodes are specific to each monitoring tool. Once integrated into a message flow, the node collects statistics about the flow processing and reports these to a component that displays the results. Usually such tools also allow the expected behaviour of the flow to be defined, for example when it has to be up and how many messages are expected to be processed. In case there are significant deviations from the expected values, the monitoring tools inform the operator to look at the origin of the deviation.

BUSINESS PROCESS MONITORING OVERVIEW

This section provides an overview about business process monitoring capabilities for message flows.

Post-mortem analysis

Post-mortem analysis or reporting is required for the analysis of all messages that have been processed. Such analysis is usually repeated at pre-defined intervals, eg the reports are produced daily or weekly. The reports usually summarize the content of messages that have been processed within the reporting period. What sort of information should be reported depends on the kinds of message that are processed in the message flow. It can be, for example, the total number of messages, the value represented by the messages (either total or sorted by customer), the items referenced in the messages, or similar. From such reports you can see longer-term trends and react accordingly. For example if the reports show that the message volume from a specific customer is increasing, you may plan to provide additional processing capacity.

These examples show that the reporting depends on the content of the messages that are processed. In general there are two different ways to provide the data for reporting. In the first method, the message flow prepares the information while the messages are processed. This approach can always be used if you know in advance what data is needed for the reports. It is a relatively fast method, but it is also inflexible should you need other or additional information.

If you don't know in advance what information is needed for a report or you cannot afford the resources required to prepare the information while your message flows are processing the message, eg because they would reduce the message throughput at peak times, you can produce reports based on the information stored during processing. With this approach, there are also two different methods you can use. WebSphere BI for FN provides the means for both ways – called message auditing and message warehousing. Using message audit the messages as they are at a specific point during processing are stored as binary streams in a message audit database. The message data is usually stored at the beginning of the message processing (eg shortly after the MQInput node), or at the end of processing (for example near the MQOutput node), or at both locations.

Storing the message in a searchable format is called message warehousing. Message warehousing can be done in different ways. The simplest way is to collect the base for reporting by extracting the data from each message and storing it, eg in a database. More sophisticated approaches store the complete message, eg using the WebSphere MQ Integrator Broker Warehouse node. Or you can have a combination of both approaches, as implemented by WebSphere BI for FN.

Using the WebSphere MQ Integrator Broker Warehouse node, the messages are stored in binary format in a warehouse table. You can also use the node to store a set of fields from the message in a warehouse table. The WebSphere BI for FN Message Warehouse nodes convert all messages into XML format before storing them in a message warehouse table. For storing the data, an XMLCLOB column is used. XMLCLOB is a user-defined data type defined by DB2 XML Extender. Using this technique, the complete message can be used to search for messages or to get information about the messages that need to appear in a report. To simplify searching, together with the complete message, a set of pre-defined search fields can be set when storing the data in the message warehouse.

Once stored, a reporting program is run at off-peak times and it processes the entries collected by the flow. Depending on the area you're working in, there are probably already a set of programs that can dynamically generate reports based on your message data. If the program works on message audit data, the program must be able to interpret and parse the messages themselves. If the program works on a message warehouse, the program can access the fields using standard SQL means.

Real-time business monitoring

A pre-requisite for real-time business monitoring is that you have defined your business processes and you may run them through a processing engine. Within such a business process, one or more message flows can execute processing steps. Individual process instances can then issue messages to invoke the message flows. To be able to correlate the message with the process instance, each of the messages needs to have an indicator in the message that identifies the process instance it belongs to. Such identification is called Process Instance ID.

Most business process execution engines include a real-time business monitoring tool. An example of such a monitoring tool is WebSphere Business Integration Monitor. (For details about WebSphere Business Integration Monitor see http://

www.ibm.com/software/integration/wbimonitor.) Similar to system monitoring, the flow needs to provide the processing information about a message to the monitoring program when integrating a message flow into the process monitoring. This can be done by integrating a message processing node into your message flow. An example for such an integration using a message processing node is available as WebSphere MQ Integrator Broker SupportPac IB01. (You can find the SupportPac IB01 at http://www.ibm.com/software/integration/support/supportpacs.) When using this node in your flow you can provide a mapping of data elements from elements in the message to the information needed for monitoring. This information is then stored in the database of the WebSphere Business Integration Monitor. The monitoring program can then display the status.

Integrating a message processing node for a specific business process monitor can be done if you already know your business monitoring infrastructure. However, this is not the case for products and solutions on top of WebSphere MQ Integrator Broker. Different customers may have different business monitoring infrastructures. Therefore with WebSphere BI for FN you can choose an alternative way. WebSphere BI for FN defines a message format for control information that can be passed to a WebSphere BI for FN enabled message flow. The control information is located in the MQRFH2 header of a message. Parts of this control information are three optional fields – message group id, external reference, and business group id. If these fields are exploited by the sending program, for example by storing the Process Instance ID in one of these optional fields, they are stored in separate columns for message audit and message warehouse entries. When you are looking for a specific message, your business monitor can probe these database tables to see whether they contain entries with your Process Instance ID. Based on where and when during message processing message warehouse or message audit are invoked, the monitor can deduce the state of the process and where your message is currently.

CONCLUSION

It is already possible to start monitoring the processing of message flows. The greater your monitoring requirements are, the more information you must provide to fulfil them. Many system monitoring tasks can already be performed by using standard monitoring programs. You just need to identify the resources involved in your message flow processing and you need to provide information about what can be deduced when looking at them.

The more you need to monitor the business characteristics of the messages that you process, the more work you have to do to integrate with the monitoring program. For WebSphere BI for FN, an analysis of how to achieve this has been described. From this it can be deduced that the same should also be possible for all other message flows.

For detailed monitoring you need to take a closer look at your monitoring requirements and you may have to check how this can be achieved with the monitoring infrastructure available at your company.

Michael Groetzner
IBM (Germany)

Using the distribution list to send a message to multiple destinations

INTRODUCTION

This is a Java console application that uses the distribution list to put message into multiple destinations. The destinations can be combinations of local, alias, or remote queues. This article is intended for users who are familiar with MQ operations like **put** and **get**; it will deal directly with the distribution list classes and their methods.

DISTRIBUTION LIST

Distribution lists allow a message to be sent to multiple destinations in a single **put** call. Multiple queues can be opened using a single **open** and a message can then be put to each of those queues using a single **put**.

In distribution lists, the response records contain the specific completion code and reason code for each destination. The completion code MQCC_FAILED indicates that no message has been put on any destination queue successfully. If the completion code is MQCC_WARNING, the message has been successfully put on one or more of the destination queues. If the return code is MQRC_MULTIPLE_REASONS, the reason codes are not all the same for every destination. Therefore, it is recommended to use the MQRR structure to determine which queue or queues caused an error and the reasons for each.

When an **open** call is issued, generic information is taken from the MQ Object Descriptor (MQOD). When a message is put on the queues, generic information is taken from the MQ Put Message Option structure (MQPMO) and the MQ Message Descriptor (MQMD). Specific information is given in the form of MQ Put Message Records (MQPMRs). MW Response Records (MQRR) can receive a completion code and reason code specific to each destination queue.

Put Message Records (MQPMR)

The MQPMR structure is used to specify various message properties for a single destination when a message is being put to a distribution list.

Response Records (MQRR)

The MQRR structure is used to receive the completion code and reason code resulting from the open or put operation for a single destination queue, when the destination is a distribution list.

In Java, MQ structures like MQOD, MQPMR, MQRR, etc are collapsed into the parameters of a class and their corresponding methods.

When the new message instance is created, all the MQMD parameters are automatically set to their default values. The put method of MQQueue also takes an instance of the MQPutMessageOptions class as a parameter. This class represents the MQPMO structure.

CODE SAMPLE

```
/**********************************************************************/
/* Program name: MQMsgDistribution                                    */
/* Author : Balaji SR                                                 */
/* MQ Administrator                                                   */
/*eFunds International India Pvt. Ltd.                                */
/* mail Id: balaji_srajan@yahoo.com                                   */
/**********************************************************************/
/*                                                                    */
/* Function:                                                          */
/* This application uses the distribution list to put message into   */
/* destinations in a single call. Queue manager name, port number,    */
/* and the server multiple connection channel are hardcoded inside    */
/* the program. It takes queue name(s) arguments and uses the         */
/* distribution list to send message to all the queue(s)              */
/*                                                                    */
/*        Invocation:                                                 */
/*        java mqDList queue1, queue2, queue3                         */
/*        Enter the message to be distributed to the queue(s)....     */
/*        <Message for the Distribution queues>                       */
/**********************************************************************/

import com.ibm.mq.*;
import java.util.*;
import java.io.*;

public class MQMsgDistribution {

      // QMGR object
      private MQQueueManager          mqQueueManager;
      // Queue object
      private MQQueue                queue;
      // Open options
      private int                    openOptionInquire;
      // host name
      private String                 hostName;
      // server connection channel
      private String                 channel;
      // port number on which the QMGR is running
      private String                 port;
```

```
        // queue manager name
        private String                  qmgrName;
        // queue name
        private String                  qName;
        // distribution list
        private MQDistributionList      distList;
        // distribution list item
        private MQDistributionListItem  dListItem[];

        public static void main(String arg[])
        {

              try{

                    if (arg.length == 0)
                    {
     System.out.print("Please enter the arguments in the order of...\n" );
                   System.out.print(" queue1 [queue2] [queue3] .... \n" );
                    System.exit(1);
                    }

                 MQMsgDistribution mqDistriList = new MQMsgDistribution();
                    mqDistriList.init(arg);

                    }
              catch( Exception e)
              {
              e.printStackTrace();
              }
        }

        private void init(String[] args)
        {

              try{
                    this.mqInit( args );
              }
              catch( Exception e)
              {
                    e.printStackTrace();
              }
        }

  private void mqInit(String[] mqArguments )
        {
              try
              {
                    //initialization of the MQ parameters
              //host name local or remote system
                    hostName            = "localhost";
```

```
                // queue manager to get connected to
                      qmgrName           = "QMGR2";
                     // port number on which the queue manager listens
          port                   = "1415";
                // sever connection channel thru which the application
                     // communicates
                     channel                  = "SYSTEM.DEF.SVRCONN";
          //prints the MQ environment variables
                System.out.println("MQ distribution list .....");
                System.out.println("—————————————————————————");
            System.out.println("host name            : " + hostName);
            System.out.println("QMGR name            : " + qmgrName);
            System.out.println("port number            : " + port);
          System.out.println("channel                  : " + channel);
                System.out.println("—————————————————————————");

           dListItem = new MQDistributionListItem[mqArguments.length];
                mqOperations(mqArguments);
          }
          catch (Exception exp)
          {
                System.out.println("error in mqInit....\n");
                exp.printStackTrace();
          }
    }

    private void mqOperations(String[] qNames)
  {      // connect, open & put-distribution list, close & disconnects
          try
          {
                if ( mqConnect() == true )
                      {      // queue manager connection
                      // opens the queue & puts
                            mqOpen(qNames);

                      // close the queue
                      mqClose();
                      // disconects the queue manager
                      mqDisconnect();
                                   }// end of if
          }
          catch (Exception exp)
          {
                System.out.println("error in mqOperations.....");
                exp.printStackTrace();
          }


    }      //mqOperations ends here
```

```
      private boolean mqConnect()
      {      // Connection to the queue manager
            try
            {
                  MQEnvironment.hostname      = hostName;
            MQEnvironment.channel      = channel;
MQEnvironment.port            = Integer.parseInt(port);

                  System.out.println( hostName + "  —————— " + channel +
                              " ——————  " + port);

            mqQueueManager = new MQQueueManager(qmgrName);
            System.out.println("Queue manager : " + qmgrName +
                              " connect successful ");
                  return true;

            }

            catch ( MQException mqExp)
            {
              System.out.println("error in queue manager connect....");
                  System.out.println("QMGR Name : " + qmgrName);
                  System.out.println("CC   : " + mqExp.completionCode );
                  System.out.println("RC   : " + mqExp.reasonCode);
                  return false;
                  // if connects fails it shouldn't procedure further...
            }
            catch (Exception exp)
            {
                  System.out.println("error in queue manager connect");
                  exp.printStackTrace();
                  return false;
            }
      }

      private void mqOpen(String[] queueNames)
      {      // opens the distribution list and put the message.
            // put message options
      MQPutMessageOptions putMessageOptions = new MQPutMessageOptions();
            try
            {
                  // distribution list
                  for (int i=0; i < queueNames.length ; i++)
                  {
                        dListItem[i] = new MQDistributionListItem();
                        dListItem[i].queueName = queueNames[i] ;
                  }

                  // open options
            int openOption = 0;
                  openOption = MQC.MQOO_OUTPUT |
                              MQC.MQOO_FAIL_IF_QUIESCING ;

            // open the distribution lists - queue(s)
                  distList =
mqQueueManager.accessDistributionList(dListItem, openOption);
System.out.println("queue(s) open for distribution list successful...");
                  System.out.println("number of successful open : " +
                              distList.getValidDestinationCount());

                  System.out.println("number of open failed :  " +
                                 distList.getInvalidDestinationCount());

                 MQMessage message = new MQMessage();      //for message
                  // set the message formate to String
                  message.format =  MQC.MQFMT_STRING;

                  try
                  {
               //for capturing the input (message) from the command line
                        // from user

                  BufferedReader             bufRead;
                        InputStreamReader      inStmRead;
                    inStmRead       = new InputStreamReader(System.in);
                        bufRead      = new BufferedReader(inStmRead);
                        String       putMessage = "";
          System.out.println("Enter the message to be distributed to the
                        queue(s)....");
             //putting the message into the distribution list - queue(s)
                        putMessage = bufRead.readLine();
                        message.clearMessage();
                        message.writeString(putMessage);
                        distList.put(message, putMessageOptions);
                      System.out.println( "put message successful... ");
                      System.out.println("number of successful put : " +
                                    putMessageOptions.knownDestCount);

                    System.out.println("number of unsuccessful put : " +
                              putMessageOptions.invalidDestCount);

    System.out.println("number of unknown (remote) destination count : "
                              + putMessageOptions.unknownDestCount);

                  } //end of try

                  catch (IOException e)
                  {
       System.out.println("IOException during put: " + e.getMessage());
```

```
                } //end of catch

            }
            catch ( MQException mqExp)
            {
        System.out.println("error in distribution queue - open & put ....");
                System.out.println("CC    : " + mqExp.completionCode );
                System.out.println("RC    : " + mqExp.reasonCode);
                System.out.println("number of put failed : " +
                            putMessageOptions.invalidDestCount);
                System.out.println("number of successful put : " +
                            putMessageOptions.knownDestCount);
      System.out.println("number of unknown (remote) destination count : " +
                      putMessageOptions.unknownDestCount);
                //  MQRC_MULTIPLE_REASONS - multiple reasons
                if ( mqExp.reasonCode == 2136)
                {
                      for ( int i=0;i<dListItem.length;i++)
                      {
                            if (dListItem[i].completionCode != 0)
                            {
                             System.out.println("Queue name      : " +
                                        dListItem[i].queueName);

                            System.out.println("Reason code      : " +
                                        dListItem[i].reasonCode);
                            }  // end of inner if
                      }  // end of for loop
                } // end of if - reason code 2136
            }  // end of catch - MQException

      } //end of mqOpen

      private void mqClose()
      {      // close the queue
            try
            {
                  distList.close();
                  System.out.println("Close queue successful.....");
            }
            catch (MQException mqExp)
            {
                  System.out.println("Error in closing queue ....");
                  System.out.println("Queue Name : " + qName);
                  System.out.println("CC    : " + mqExp.completionCode );
                  System.out.println("RC    : " + mqExp.reasonCode);
            }
      } // end of mqClose

      private void mqDisconnect()
      {      // disconnect to queue manager
            try
            {
                  mqQueueManager.disconnect();
                  System.out.println("Queue manager : " + qmgrName + "
                                    disconnect successful ");
            }

            catch ( MQException mqExp)
            {
           System.out.println("Error in queue manager disconnect....");
                  System.out.println("QMGR Name : " + qmgrName);
                  System.out.println("CC    : " + mqExp.completionCode );
                  System.out.println("RC    : " + mqExp.reasonCode);
            }
      } // end of mqDisconnect
}
```

The following classes are used to create distribution lists:

- MQQueueManager
- MQQueue
- MQMessage
- MQDistributionList
- MQDistributionListItem
- MQPutMessageOptions
- MQException.

MQDISTRIBUTIONLIST

A distribution list represents a set of open queues to which messages can be sent using a single call to the put method. Methods getInvalidDestinationCount and getValidDestinationCount are used for determining the successful opens:

- GetInvalidDestinationCount returns the number of items in the distribution list that failed to open successfully.
- GetValidDestinationCount returns the number of items in the distribution list that were opened successfully.

MQDISTRIBUTIONLISTITEM

An MQDistributionListItem represents a single item (queue) within a distribution list. Variables are completionCode, queueManagerName, queueName, and reasonCode:

- completionCode – the completion code resulting from the last operation on this item. If it was an open operation, the completion code relates to the opening of the queue. If it was a put operation, the completion code relates to the attempt to put a message on to this queue.
- queueName – the name of the queue to be used in the distribution list.
- reasonCode – the reason code resulting from the last operation on this item. If it was an open operation, the reason code relates to the opening of the queue. If it was a put operation, the reason code relates to the attempt to put a message onto this queue.

MQPUTMESSAGEOPTIONS

This class contains options that control the behaviour of MQQueue.put. The following fields in the MQPMO are rendered as the member variables knownDestCount, unknownDestCount, and invalidDestCount in the MQPutMessageOptions class:

- knownDestCount – an output field set by the queue manager to the number of messages that the current call has sent successfully to queues resolving to local queues.
- invalidDestCount – an output field set by the queue manager to the number of messages that could not be sent to queues in a distribution list. The count includes queues that failed to open as well as queues that were opened successfully, but for which the put operation failed.
- unknownDestCount – an output field set by the queue manager to the number of messages that the current call has sent successfully to queues resolving to remote queues.

For example:

1 If the queue resolves to a local queue, knownDestCount is set to 1 and the other two count fields are set to 0.

2 If the queue resolves to a remote queue, unknownDestCount is set to 1 and the other two count fields are set to 0.

These fields are primarily intended for use with distribution lists.

MQEXCEPTION

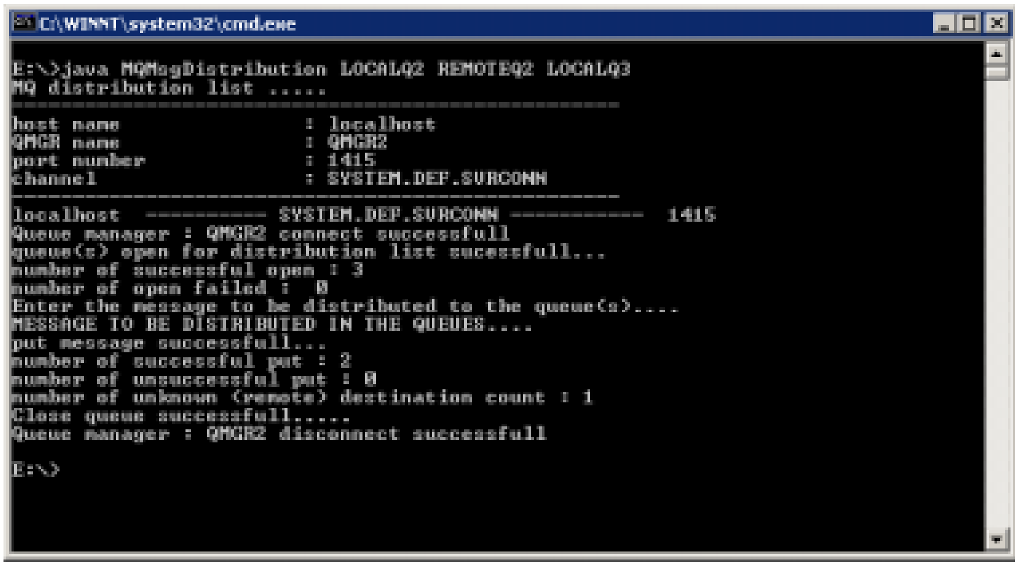
An MQException is thrown whenever a WebSphere MQ error occurs. Variables are completionCode and reasonCode.

- completionCode – gives the completion code of the MQ call made.
- reasonCode – describes the error.

Example 1 output:

```
> java MQMsgDistribution LOCALQ2 REMOTEQ2 LOCALQ3
```

The message is sent to two queues (two local and one remote



```
E:\>java MQMsgDistribution LOCALQ2 REMOTEQ2 LOCALQ3
MQ distribution list .....
------------------------------------------------------------------
host name                    : localhost
QMGR name                    : QMGR2
port number                  : 1415
channel                      : SYSTEM.DEF.SVRCONN
------------------------------------------------------------------
localhost  ---------- SYSTEM.DEF.SVRCONN -----------  1415
Queue manager : QMGR2 connect successfull
queue(s) open for distribution list sucessfull...
number of successful open : 3
number of open failed :  0
Enter the message to be distributed to the queue(s)....
MESSAGE TO BE DISTRIBUTED IN THE QUEUES....
put message successfull...
number of successful put : 2
number of unsuccessful put : 0
number of unknown (remote) destination count : 1
Close queue successfull.....
Queue manager : QMGR2 disconnect successfull

E:\>
```

Figure 1: Example 1 output

queue).

Figure 1 shows that the local queue put is successful:

```
number of successful put : 2
```

The remote queue put is successful:

```
number of unknown (remote) destination count : 1
```

Example 2 output:

```
> java MQMsgDistribution LOCALQ2 REMOTEQ2 LOCALQ3
```

The message is sent to two queues (two local and one remote queue).

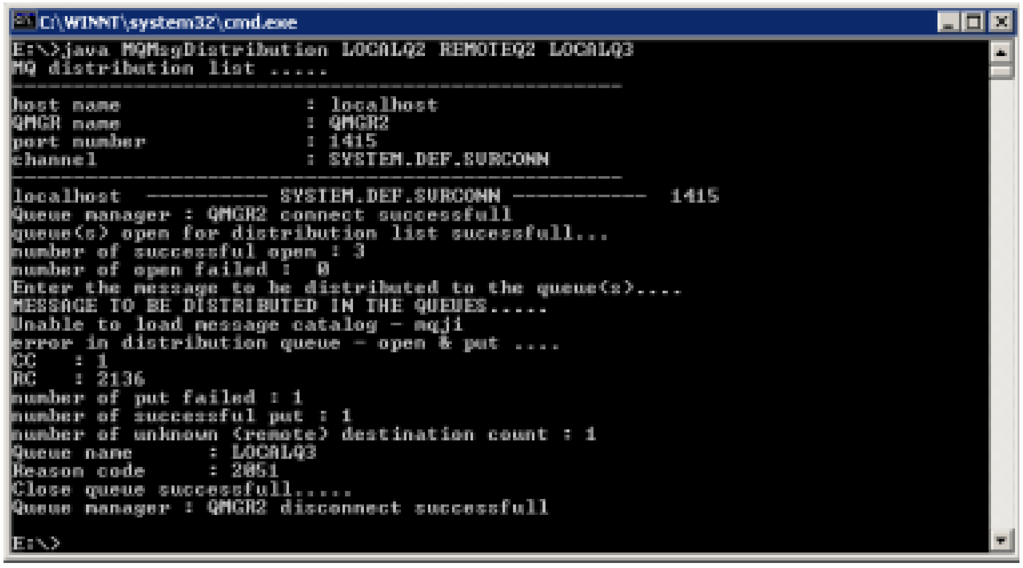
Figure 2 shows that the local queue put is successful:

```
number of successful put : 1
```

The second local queue put failed:

```
number of put failed : 1
Queue name      : LOCALQ3
Reason code     : 2Ø51 (MQRC_PUT_INHIBITED)
```

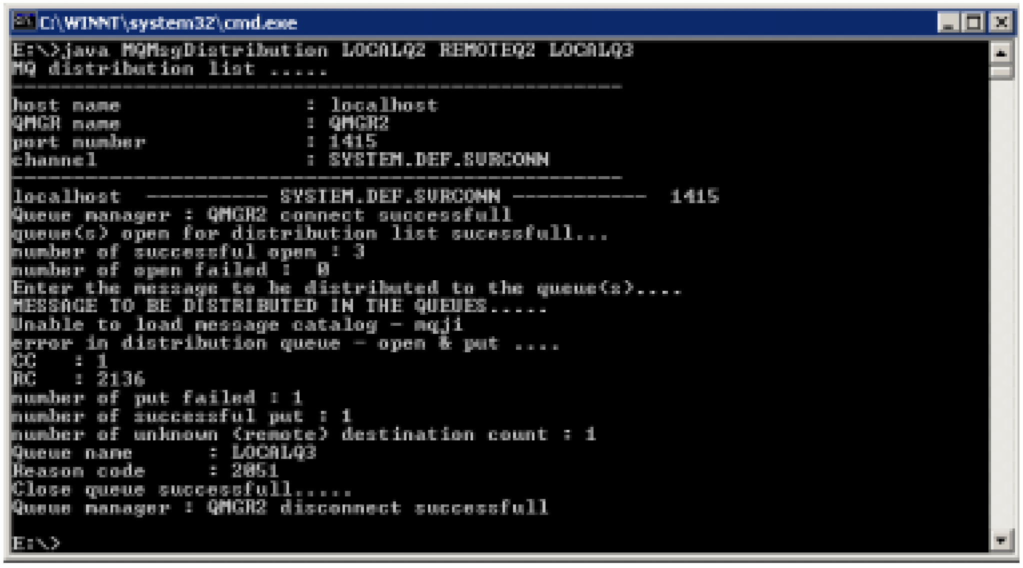
The remote queue put is successful:



```
E:\>java MQMsgDistribution LOCALQ2 REMOTEQ2 LOCALQ3
MQ distribution list .....
--------------------------------------------------
host name                  : localhost
QMGR name                  : QMGR2
port number                : 1415
channel                    : SYSTEM.DEF.SURCONN
--------------------------------------------------
localhost  ---------- SYSTEM.DEF.SURCONN -----------  1415
Queue manager : QMGR2 connect successfull
queue(s) open for distribution list sucessfull...
number of successful open : 3
number of open failed :  0
Enter the message to be distributed to the queue(s)....
MESSAGE TO BE DISTRIBUTED IN THE QUEUES.....
Unable to load message catalog - mqji
error in distribution queue - open & put ....
CC    : 1
RC    : 2136
number of put failed : 1
number of successful put : 1
number of unknown (remote) destination count : 1
Queue name       : LOCALQ3
Reason code      : 2051
Close queue successfull.....
Queue manager : QMGR2 disconnect successfull

E:\>
```

Figure 2: Example 2 output



```
E:\>java MQMsgDistribution LOCALQ2 REMOTEQ2 LOCALQ3
MQ distribution list .....
--------------------------------------------------
host name                  : localhost
QMGR name                  : QMGR2
port number                : 1415
channel                    : SYSTEM.DEF.SVRCONN
--------------------------------------------------
localhost  ---------- SYSTEM.DEF.SVRCONN -----------  1415
Queue manager : QMGR2 connect successfull
queue(s) open for distribution list sucessfull...
number of successful open : 3
number of open failed :  0
Enter the message to be distributed to the queue(s)....
MESSAGE TO BE DISTRIBUTED IN THE QUEUES.....
Unable to load message catalog - mqji
error in distribution queue - open & put ....
CC    : 1
RC    : 2136
number of put failed : 1
number of successful put : 1
number of unknown (remote) destination count : 1
Queue name       : LOCALQ3
Reason code      : 2051
Close queue successfull.....
Queue manager : QMGR2 disconnect successfull

E:\>
```

Figure 3: Example 3 output

```
number of unknown (remote) destination count : 1
```

Example 2 output:

```
> java MQMsgDistribution LOCALQ2 REMOTEQ2 LOCALQ3
```

The message is sent to two queues (two local and one remote queue).

Figure 3 shows that the local queue put failed:

```
number of put failed : 2
Queue name       : LOCALQ2
Reason code      : 2053 (MQRC_Q_FULL)
Queue name       : LOCALQ3
Reason code      : 2051 (MQRC_PUT_INHIBITED)
```

The remote queue put is successful:

```
number of unknown (remote) destination count : 1
```

Balaji SR
MQ Administrator
eFunds International (India)

webMethods has announced that IBM technology users can extend the ROI of their existing investments by leveraging webMethods' standards-based integration products. webMethods also announced that it has joined IBM's PartnerWorld program to better support customers using IBM's software and hardware, especially those running Linux on mainframes.

webMethods offers a product set that allows customers to access mainframe data through a variety of methods, all of them non-invasive to the mainframe applications. The webMethods Enterprise Services Platform is able to easily expose these integrations as Web services, allowing customers to incorporate these critical mainframe assets into service-oriented architecture initiatives without the need for extensive coding or development.

webMethods provides strong integration capabilities to IBM WebSphere Application Server and WebSphere MQ.

For further information contact:
webMethods, 3930 Pender Drive, Fairfax, VA 22030, USA.
Tel: (703) 460 2500.
URL: http://www.webmethods.com/meta/default/folder/0000003377?hiddenRequest=true&pressReleaseDetails_param0=6290.

* * *

ILS Technology has introduced xCoupler, a new turnkey solution that enables industrial manufacturers to connect logic controllers on the factory floor directly into the company's message queueing or database systems. This product is designed to replace the existing but more expensive and problematic use of personal computers (PCs) put in place just to handle logic controller device drivers and the resulting data manipulation to multiple systems in an enterprise.

xCoupler MQLink links LOGIC CONTROLLER data to WebSphere MQ and Microsoft Message Queuing (MSMQ).

For further information contact:
ILS Technology, 5300 Broken Sound Blvd, Suite 150, Boca Raton, FL 33487, USA.
Tel: (561) 982 9898.
URL: http://ilstechnology.com/products/xCoupler.html.

* * *

ICEsoft Technologies has announced that Triversity has adopted ICEbrowser as its primary HTML rendering engine for Transactionware Enterprise.

Transactionware Enterprise is a J2EE solution for point of sales (POS) and point-of-interaction applications in the retail enterprise. Both ICEbrowser and Triversity Enterprise Framework fully integrate with IBM's J2EE infrastructure - WebSphere Application Server 5, WebSphere MQ, and DB2 UDB.

For further information contact:
ICEsoft Technologies, Suite 300, 1717 10th St NW, Calgary, Alberta T2M 4S2, Canada.
Tel: (877) 263 3822.
URL: http://www.icesoft.com/products/icebrowser.html.

* * *