57

MQ update

April 2004

In this issue

3 Are you connected?

10 Viewing a queues properties from the Java console

19 Customizing WebSphere MQSeries to run with Veritas Cluster Server

36 Visual debugging in IBM WebSphere Business Integration Message Broker V5.0 Toolkit

45 MQ news

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

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

Subscriptions and back-issues

A year's subscription to *MQ Update*, comprising twelve monthly issues, costs $380.00 in the USA and Canada; £255.00 in the UK; $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.

Printed in England.

Are you connected?

This short article aims to highlight a potential pitfall for those of you writing Java client programs using the MQ-supplied interface.

The question is, how do you ensure that if there is a connection problem the program will handle any errors and reconnects gracefully?

Basically, every time the queue manager on AIX was restarted, our MQ/Java client programs running on Linux were not able to re-establish a connection and continually received return code 2009. The problem was exacerbated because at the time we were using the Unix-supplied listener INETD (which spawned a separate process for each channel request); it has now been replaced by the MQ supplied listener **runmqlsr**, which, since V5.3, is the recommended listener.

The IBM manual *WebSphere MQ using Java* (ref SC34-6066-01) has a description of the following items within class MQQueueManager:

```
Variables:
isConnected
public boolean isConnected
True if the connection to the queue manager is still open.

Methods:
isConnected
public boolean isConnected()
Returns the value of the isConnected variable.
```

It seems from the above description that the **isConnected()** method is a good way to check whether the connection to the queue manager still exists.

Unfortunately, it is not!

The following MQ Java program, called test2009, demonstrates the problem.It displays the value of **isConnected()** as well as my own separately maintained field called *connected*. Indeed, it shows that when the queue manager is stopped, the

isConnected() method returns TRUE, contrary to the description in the manual.

My own variable detects when a return code 2009 occurs and sets its value to FALSE.

A call to the IBM Support Centre soon revealed an explanation of the cause of the problem.

It turns out that the **isConnected** variable, and hence the method, gets updated only when the program specifically connects and disconnects! In other words, it does not get updated if the connection is dropped. For those of you who have access to IBM internal documents, it is described in document RTA000165151.

I have requested a change to the manual, which has been accepted by IBM.

TEST2009 JAVA

```
import com.ibm.mq.*;
import java.io.*;
import java.lang.*;
//*********************************************************************
// Program Name : test2009
// Function     : Test program to demonstrate programming method of
//                coping with MQ client connection problems.
// Author       : Ruud van Zundert, January 2004
// Invocation   : java test2009 parms
// Parameters   : The following parameters are required
//                parm1 : MQ queue manager name
//                parm2 : MQ queue name
//                parm3 : MQ SVRCONN channel name
//                parm4 : MQ listener port number
//                parm5 : IP address/DNS name where queue manager resides
//*********************************************************************
public class test2009
{
  public String UsrId = "";
  public String Pswd = "";
  public MQQueueManager qMgr;                    // Queue manager
  public MQQueue queue;
  public MQGetMessageOptions gmo;
  public MQMessage msg;
  public int openOptions;
  public int loop=0;
```

```
  public int i=0;
  public boolean connected=false;
  public String msgText;

  public static void main( String[] args ) {

    System.out.print("*** Java MQ client utility to read messages ***");
    System.out.print("*** Author: Ruud van Zundert ***\n");
    if (args.length < 4) {
      System.out.print("Please supply all 5 required parameters\n");
      System.out.print("1=qmgr,2=queue,3=channel,4=port,5=ip addr/DNS
name\n");
      return;
    }

    test2009 instance = new test2009();
    instance.init( args );

    boolean keepgoing=true;

    while ( keepgoing ) {
      keepgoing = instance.getmsgs ( args );
    }

    System.out.print("*** end of utility ***\n");

  }

  // Initialize the environment.
  void init( String[] args ) {
  // Set up MQ client environment
  //                             (ie normally would be in env variables)

 // Tracing - currently commented out
 // try {
 //   FileOutputStream
 //   traceFile = new FileOutputStream("test2009.trc");
 //   MQEnvironment.enableTracing(2,traceFile);
 // }
 // catch (FileNotFoundException ex) {
 //   MQEnvironment.enableTracing(2);
 // }


    MQEnvironment.hostname = args [ 4 ];
    MQEnvironment.channel  = args [ 2 ];
    MQEnvironment.port     = Integer.parseInt( args[3] );
    MQEnvironment.userID   = UsrId;
    MQEnvironment.password = Pswd;
```

```
  } // end of init

// Get messages
  boolean getmsgs( String[] args ) {
    boolean gotMsg, keepgoing=true;
    int cnt=0;
    loop++;
    try {
      System.out.print("\nLoop "+loop+" Sleeping for 3 seconds... ");
      Thread.sleep(3000);
      System.out.print("\nworking storage connection status
"+connected+"\n");

      if ( !connected ) {
      System.out.print("\nattempting (re)connect...\n");
      qMgr = new MQQueueManager( args [ 0 ] );      // connect to qmgr
      System.out.print("\nconnect successful\n");
      connected=true;                       // only reached if it worked
      gmo = new MQGetMessageOptions();          // 'get' message options
     openOptions = MQC.MQOO_FAIL_IF_QUIESCING + MQC.MQOO_INPUT_AS_Q_DEF;

        queue    = qMgr.accessQueue( args [ 1 ], openOptions,
                                null,           // default q manager
                                null,           // no dynamic q name
                                null );         // no alternate user id

      } // end if

      do {
      gotMsg = true;
      msg    = new MQMessage();               // Obtain message buffer
      queue.get( msg, gmo);                   // get message from queue

        cnt++;                                // increment message count

        i       = msg.getMessageLength();
      msgText = msg.readString(i);
      System.out.print("\n"+cnt+"="+msgText+"\n");
      if ( msgText.length() > 3 && msgText.indexOf("stopstop") >= 0 ) {
        System.out.print("\n*** Request to stop received ***\n");
        gotMsg = false;
      }

      } while ( gotMsg );

      queue.close();
      qMgr.disconnect();                          // disconnect from qmgr
      System.out.print("\ndisconnected\n");
      connected=false;
      keepgoing=false;
```

```
    } // end try

    catch (MQException ex) {
      gotMsg = false;                                  // did not get msg
      switch (ex.reasonCode) {
      case 2033:
        break;
      case 2009:
        System.out.println("\n*** Connection broken  ***\n");
        if ( qMgr != null ) {
          System.out.print("\nisConnected status "+qMgr.isConnected());
        }
        connected=false;
        break;
      default:
        System.out.println("\nMQ error. " + ex.completionCode + " Reason
code " + ex.reasonCode+"\n");
      }

    }
    catch (java.io.IOException ex) {
      System.out.println("/nAn error occurred whilst getting from the
message buffer: " + ex);
    }
    catch (InterruptedException ex) {
      System.out.println("/nAn error occurred related to sleep " + ex);
    }

    return keepgoing;

  } // end of getmsg

}  // end of program
```

RUNNING THE PROGRAM

Obviously, to run the program you'll need to properly set up your environment.

This program has successfully run on both Windows 2000 and XP as well as AIX 5.1.

For the Windows environment:

- Install Java runtime (and SDK if you want to change the code).

- Install MQ V5.3 and apply the latest CSD.
- If necessary, set up the PATH and CLASSPATH environment variables.

Enter this command:

```
java test2ØØ9 T1 LQT1 SCONT1 14141 localhost
```

where *T1* is the queue manager, *LQT1* is a local queue, *SCONT1* is a server connection channel, *14141* is the port, and *localhost* is the hostname. Choose your own appropriate values.

These results are shown in the following output, which has been annotated with actions taken during the test.

```
*** Java MQ client utility to read messages ****** Author: Ruud van
Zundert ***

Loop 1 Sleeping for 3 seconds...
working storage connection status false

attempting (re)connect...

connect successful
*** PROGRAM PICKS UP 3 MESSAGES ***
1=test message 1
2=test message 2
3=test message 3

Loop 2 Sleeping for 3 seconds...
working storage connection status true

Loop 3 Sleeping for 3 seconds...
working storage connection status true
*** AT THIS POINT QUEUE MANAGER IS STOPPED ***
MQ error. 2 Reason code 2162

Loop 4 Sleeping for 3 seconds...
working storage connection status true

*** Connection broken  ***

isConnected status true
Loop 5 Sleeping for 3 seconds...
working storage connection status false
*** NOTE THAT THE WORKING STORAGE FIELD IS 'FALSE'
*** WHEREAS isConnected IS STILL 'TRUE'
```

```
attempting (re)connect...

*** Connection broken  ***

isConnected status true
Loop 6 Sleeping for 3 seconds...
working storage connection status false

attempting (re)connect...

MQ error. 2 Reason code 2Ø59

Loop 7 Sleeping for 3 seconds...
working storage connection status false

Loop 8 Sleeping for 3 seconds...
working storage connection status false
*** AT THIS POINT QUEUE MANAGER IS STARTED ***
attempting (re)connect...

connect successful
*** ADD 3 MORE MESSAGES TO QUEUE VIA AMQSPUT ***
Loop 9 Sleeping for 3 seconds...
working storage connection status true

1=test message 1
2=test message 2
3=test message 3

Loop 1Ø Sleeping for 3 seconds...
working storage connection status true

Loop 11 Sleeping for 3 seconds...
working storage connection status true
*** ADD MSG TO QUEUE WITH 'stopstop' TO STOP PROGRAM ***
1=stopstop

*** Request to stop received ***

disconnected
*** end of utility ***
```

CONCLUSION

Do not rely on the **isConnected()** method to test whether the MQ/Java client is currently connected. Instead, test for MQ return code 2009 and reconnect.

The test2009.class file can be downloaded from www.xephon.com/extras/test2009.class.

Ruud van Zundert (ruudvz@btclick.com)
Independent Consultant (UK)

Viewing a queues properties from the Java console

Here is an MQSeries-Java Interface that enables users to view the properties of queues via the Java console.

```
/*********************************************************************/
/* Program name: mqAttrInquiry                                       */
/* Author : Balaji SR                                                */
/*********************************************************************/
/* Function:                                                         */
/*  This is a Java console application, which will take Queue Name   */
/*  and Queue Manager name as arguments. This will display the queue */
/* attributes on the console                                         */
/*********************************************************************/

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

public class mqAttrInquiry {
      private MQQueueManager mqQueueManager;         // for QMGR object
      private MQQueue queue;                  // for Queue object
      private int openOptionInquire;          // Open options
      private String hostName;                // for host name -> QMGR
      private String channel;           // server connection channel
      private String port;
                              // port number on which the QMGR is running
      private String qmgrName;
      private String qName;

      public static void main(String arg[])
      {
            try{
                  if ( arg.length == 0)
                  {
      System.out.print("Please enter the argument in the order of \n" );
      System.out.print("Queue name  Queue manager name");
```

```
                System.exit(1);
            }

            mqAttrInquiry mqinqset = new mqAttrInquiry();
            System.out.println("Queue name   " + arg[Ø]);
            System.out.println("Queue manager name   " + arg[1]);
            mqinqset.init(arg[Ø], arg[1]);
            }
      catch( Exception e)
      {
            e.printStackTrace();
      }
}

public void init(String queueName, String QMGRName)
{
      try{
            System.out.println("In init");
            this.mqInit(queueName, QMGRName);
            this.getConnected();
      }
      catch( Exception e)
      {
            e.printStackTrace();
      }
}

private void mqInit(String queueName, String QMGRName)
{      // Initiation of the MQ parameter
      hostName      = "localhost";
      port            = "1414";
      qmgrName      = QMGRName;
      channel           = "SYSTEM.DEF.SVRCONN";
      qName           = queueName;
}

public void getConnected() throws Exception
{           // gets connected to the Queue & checks the queue
            // depth high event & if the event is set,
      // it start the broker & send mails
      try
      {
            mqConnect();
            mqOpen();
            chexQType();
            mqClose();
            mqDisconnect();
      }

      catch (Exception exp)
```

```
            {
                  exp.printStackTrace();
            }

      }      //getConnected ends here

      private void mqConnect() throws Exception
      {      // 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("Qmgr : " + qmgrName + " connection successful ");

            }

            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);
            }
      }

      private void mqDisconnect() throws Exception
      {      // disconnect to queue manager
            try
            {
                  mqQueueManager.disconnect();
System.out.println("Qmgr : " + qmgrName + " disconnection 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

      private void mqOpen() throws MQException
      {
```

```
            try
            {
                  int openOption = Ø;
                  openOption = MQC.MQOO_INQUIRE;
queue = mqQueueManager.accessQueue(qName, openOption,null,null,null);
                  System.out.println( "Open queue sucessfull... ");

            }
            catch ( MQException mqExp)
            {
                  System.out.println("Error in opening queue ....");
                  System.out.println("Queue Name : " + qName);
                  System.out.println("CC   : " + mqExp.completionCode );
                  System.out.println("RC   : " + mqExp.reasonCode);
            }

      } //end of mqOpen

      private void mqClose() throws MQException
      {
            try
            {
                  queue.close();
                  System.out.println("Close queue successfull.....");
            }
            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 chexQType() throws MQException
      {      // for checking the queue type
            try
            {
                  int[] qSelectors = new int[1];
                  int[] qIntAttrs = new int[1];
                  byte[] qCharAttrs = new byte[1];
                  int MQIA_Q_TYPE = 2Ø;
                  qSelectors[Ø] = MQIA_Q_TYPE ;

                  qInquire(qSelectors,qIntAttrs,qCharAttrs);
                  qType(qIntAttrs[Ø]);
            }
            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 chexQType

     private void qAliasInquiry() throws MQException
     {      // to get the Base queue name for Alias queue
          try
          {
                int MQCA_BASE_Q_NAME = 2002;
                int MQ_Q_NAME_LENGTH = 48;
                int[] qAliasSelectors = new int[1];
                int[] qAliasIntAttrs = new int[0];
                byte[] qAliasCharAttrs = new byte[MQ_Q_NAME_LENGTH];
                qAliasSelectors [0] = MQCA_BASE_Q_NAME ;

                System.out.println(" in qAliasInquiry ...");

           qInquire(qAliasSelectors, qAliasIntAttrs, qAliasCharAttrs);
                System.out.println(" Base queue name  : " + new String
(qAliasCharAttrs));
          }

          catch (MQException mqExp)
          {
                System.out.println("Error in qAliasInquiry....");
                System.out.println("Queue Name : " + qName);
                System.out.println("CC   : " + mqExp.completionCode );
                System.out.println("RC   : " + mqExp.reasonCode);
          }
     }      // end qAliasInquiry

     private void qRemoteInquiry() throws MQException
     {      // to get the Remote queue properties
          try
          {
                int MQCA_REMOTE_Q_MGR_NAME = 2017;
                int MQCA_REMOTE_Q_NAME = 2018;
                int MQCA_XMIT_Q_NAME = 2024;
                int MQ_Q_NAME_LENGTH = 48;
                int MQ_Q_MGR_NAME_LENGTH = 48;
                int[] qAliasSelectors = new int[3];
                int[] qAliasIntAttrs = new int[0];
                byte[] qAliasCharAttrs = new byte[MQ_Q_MGR_NAME_LENGTH
+ MQ_Q_NAME_LENGTH + MQ_Q_NAME_LENGTH ];
                qAliasSelectors [0] = MQCA_REMOTE_Q_MGR_NAME  ;
                qAliasSelectors [1] = MQCA_REMOTE_Q_NAME ;
```

```
                qAliasSelectors [2] = MQCA_XMIT_Q_NAME ;

                System.out.println(" in qRemoteInquiry...");

            qInquire(qAliasSelectors, qAliasIntAttrs, qAliasCharAttrs);
                System.out.println(" Base queue name  : " + new String
(qAliasCharAttrs, Ø,
 MQ_Q_MGR_NAME_LENGTH ));
                System.out.println(" Base queue name  : " + new String
(qAliasCharAttrs,
 MQ_Q_MGR_NAME_LENGTH , MQ_Q_MGR_NAME_LENGTH ));
                System.out.println(" Base queue name  : " + new String
(qAliasCharAttrs,
 MQ_Q_MGR_NAME_LENGTH + MQ_Q_NAME_LENGTH,  MQ_Q_NAME_LENGTH ));
           }

           catch (MQException mqExp)
           {
                System.out.println("Error in qRemoteInquiry....");
                System.out.println("Queue Name : " + qName);
                System.out.println("CC   : " + mqExp.completionCode );
                System.out.println("RC   : " + mqExp.reasonCode);
           }
     }      // end qAliasInquiry

     private void  qLocalInquiry () throws MQException
     {
           //int currentDepth =Ø;

           try
           {
                //queue type / usage
                int MQIA_USAGE = 12;
                 //queue inquire
                int MQIA_DEF_PRIORITY = 6;
                int MQCA_Q_DESC = 2Ø13;
                int MQ_Q_DESC_LENGTH = 64;
                int MQIA_DEF_PERSISTENCE = 5;
                int MQIA_MAX_Q_DEPTH = 15;
                int MQIA_CURRENT_Q_DEPTH = 3;
                int MQIA_TRIGGER_CONTROL = 24;

                //for Event
                int MQIA_Q_DEPTH_HIGH_EVENT       = 43;
                int MQIA_Q_DEPTH_MAX_EVENT       = 42;
                int MQIA_Q_DEPTH_HIGH_LIMIT  = 4Ø;
                int MQIA_Q_DEPTH_LOW_EVENT   = 44;
                int MQIA_Q_DEPTH_LOW_LIMIT   = 41;

                //for Triggering
```

```
                  int MQCA_INITIATION_Q_NAME = 2008;
                  int MQCA_PROCESS_NAME = 2012;
                  int MQ_PROCESS_NAME_LENGTH = 48;
                  int MQ_Q_NAME_LENGTH = 48;
                  int MQIA_TRIGGER_TYPE = 28;

                  int[] selectors = new int[14];
                  int[] intAttrs = new int[11];
                  byte[] charAttrs = new byte[MQ_Q_DESC_LENGTH +
MQ_Q_NAME_LENGTH  +
 MQ_PROCESS_NAME_LENGTH ];

                  selectors[1] = MQCA_Q_DESC;
                  selectors[4] = MQIA_DEF_PERSISTENCE ;
                  selectors[5] = MQIA_MAX_Q_DEPTH ;
                  selectors[6] = MQIA_CURRENT_Q_DEPTH;

                  //for triggering
                  selectors[3] = MQCA_INITIATION_Q_NAME;
                  selectors[7] = MQIA_TRIGGER_CONTROL;
                  selectors[8] = MQIA_TRIGGER_TYPE ;
                  selectors[9] = MQCA_PROCESS_NAME ;

                  //for Event
                  selectors[0] = MQIA_Q_DEPTH_HIGH_EVENT;
                  selectors[2] = MQIA_Q_DEPTH_MAX_EVENT;
                  selectors[10] = MQIA_Q_DEPTH_HIGH_LIMIT ;
                  selectors[11] = MQIA_Q_DEPTH_LOW_EVENT ;
                  selectors[12] = MQIA_Q_DEPTH_LOW_LIMIT ;
                  selectors[13] = MQIA_USAGE ;

                  qInquire(selectors,intAttrs,charAttrs);

                  if ( intAttrs[10] == 0 )
                  {
                        System.out.println("Queue usage  Normal ");
                  }
                  else if ( intAttrs[10] == 1 )
                  {
                        System.out.println("Queue usage  XMIT ");
                  }

                System.out.println("Default Priority = " + intAttrs[0]);
                  System.out.println("Description : " + new
String(charAttrs,0,MQ_Q_DESC_LENGTH));

               System.out.println("Q_DEPTH_MAX_EVENT = " + intAttrs[1]);

                  if ( intAttrs[1] == 1 )
                  {      // Event enabled
```

```
                System.out.println("Q monitoring event is enabled " );
                      System.out.println ("Q high depth event  "  +
enableDisable(intAttrs[1]));
              System.out.println ("Q high depth limit " + intAttrs [7]);
                      System.out.println ("Q low depth event  "  +
enableDisable(intAttrs[8]));
               System.out.println ("Q low depth limit " + intAttrs [9]);
                  }

               System.out.println("Q_DEF_PERSISTENCE = " + intAttrs[2]);
                  System.out.println("Q_MAX_Depth = " + intAttrs[3]);
                System.out.println("CURRENT_Q_DEPTH = " +  intAttrs[4]);
//         System.out.println("MQIA_TRIGGER_CONTROL = " +  intAttrs[5]);

            //Trigger is set On then displays the trigger properties....
                  if ( intAttrs[5] == 1 )
                  {
                        System.out.println("Trigger is On");
           //variable decalration for getting the Trigger control values
                   System.out.println("TRIGGER_TYPE = " +  intAttrs[6]);
                        if ( intAttrs[6] == 3)
                            System.out.println("TRIGGER_TYPE is Depth");
                        else if ( intAttrs[6] == 2)
                            System.out.println("TRIGGER_TYPE is Every");
                        else if ( intAttrs[6] == 1)
                            System.out.println("TRIGGER_TYPE is First");
                        else if ( intAttrs[6] == Ø)
                             System.out.println("TRIGGER_TYPE is None");

                        //Initiation queue name
                        System.out.println("Init Q Name : " + new
 String(charAttrs,MQ_Q_DESC_LENGTH,MQ_Q_NAME_LENGTH ));
                   System.out.println("Process defination Name : " + new
 String(charAttrs,MQ_Q_DESC_LENGTH + MQ_Q_NAME_LENGTH,
MQ_PROCESS_NAME_LENGTH));
                  }

//queue inquire ends here

                  queue.close();
            }
            catch ( MQException mqExp)
            {
                  System.out.println("Error in Inquiry queue ....");
                  System.out.println("Queue Name : " + qName);
                  System.out.println("CC   : " + mqExp.completionCode );
                  System.out.println("RC   : " + mqExp.reasonCode);
            }
      }      // currDepth ends here
```

```
     private void qInquire(int[] selectors , int[] intAttrs , byte[]
charAttrs ) throws
 MQException
     {      // MQ queue inqiury
           try
           {
                 System.out.println(" in qInquire  ....");
                 queue.inquire(selectors,intAttrs,charAttrs);
           }
           catch (MQException mqExp)
           {
                 System.out.println("Error in Inquiry queue ....");
                 System.out.println("Queue Name : " + qName);
                 System.out.println("CC   : " + mqExp.completionCode );
                 System.out.println("RC   : " + mqExp.reasonCode);
           }
     }      //end of qInquire

     private String enableDisable(int parameter)
     {// to check the argument is enabled or disabled
           String str="";
           if ( parameter == 1 )
                 str = "Enabled";

           else if (parameter == Ø )
                 str = "Disabled";

           return str;

     }// end of enableDisable

     private void qType(int qTypeInt)
     {
           try
           {
                 //Queue Types
                 if ( qTypeInt == 1 )
                 {
                       System.out.println(" Local Queue  ");
                       qLocalInquiry ();
                 }
                 else if (qTypeInt == 3)
                 {
                       System.out.println(" Alias Queue  ");
                       qAliasInquiry();
                 }
                 else if (qTypeInt == 6)
                 {
                       System.out.println(" Remote Queue  ");
                       qRemoteInquiry();
```

```
                  }
                  else if (qTypeInt == 6)
                  {
                        System.out.println(" Cluster Queue  ");
                        System.out.println(" Clusrter queue are
supported in this version  ");
                  }
            }
            catch (Exception e)
            {
                  System.out.println("Error in qType function  ");
                  e.printStackTrace();
            }
      }      //end of qType
}
```

Balaji SR (balaji_srajan@yahoo.com)
MQ Administrator
eFunds International (India)

Customizing WebSphere MQSeries to run with Veritas Cluster Server

INTRODUCTION

On a recent contract, I was called on to support WebSphere MQSeries (WS MQ) running on Solaris servers controlled by a Veritas Cluster Server (VCS). In order to understand better the implications of running WS MQ under VCS control, I began looking through the IBM WebSphere MQSeries Support libraries for SupportPacs dealing with MQ and Veritas.

I quickly found the IBM SupportPac MC6A, *Configuring MQSeries for Sun Solaris with Veritas Cluster Server* at http://www-306.ibm.com/software/integration/support/supportpacs/product.html.

It seemed too good to be true, and after reviewing the contents with the VCS administrator, I found that it was: the current version

of VCS is well beyond the VERITAS Version 1.3 that MC6A was created to support. In the current releases of VCS, many of the functions that previously required agents to be written are now system functions of VCS itself. The concepts portrayed in MC6A were quite good, but the practical implementation was going to be very different.

In the following sections, I will be dealing with the WS MQ requirements for bringing queue managers under the control of VCS. I will not be dealing with the specific coding or scripting requirements of VCS. I believe that most WS MQ administrators will be dealing with VCS experts, as I was on my assignment. These system administrators understand VCS, but are probably not familiar with WS MQ.

THE ENVIRONMENT

The client environment I was working with ran the following software packages and service levels:

- Solaris V2.8 Release 22.
- WS MQSeries 5.3 with service pack U487899.
- Veritas Cluster Server 3.5 MP1.

The hardware configuration was:

- Two WS MQ servers, running a single WS MQ instance as a Portal, sharing D2 disk arrays with 36GB SCSI disks. These were configured as Raid 0 and then mirroring was implemented with Veritas to effectively create a Raid 10 environment.
- Two WS MQ servers, running two WS MQ instances App1 and App2, sharing T3 disk arrays with 18GB FCAL disks. As described above, they were configured as Raid 0 and then mirroring was implemented with Veritas to effectively create a Raid 10 environment.

This environment is shown in Figure 1.

The two sets of queue managers are handled in different ways:

- The Portal servers are an active–inactive pair.
- App1 and App2 (A1 and A2 in Figure 1) are an active–active pair.

An active – inactive pair means, in VCS terms, that one queue manager will run, on either one of the two portal machines. If the first server fails, then Veritas moves the image to the second server and processing continues. When the first server is restored, the queue manager can be moved back to the first server at any point in time. Normally, fallback would occur during the normal maintenance window.

An active – active pair means, again in VCS terms, that queue

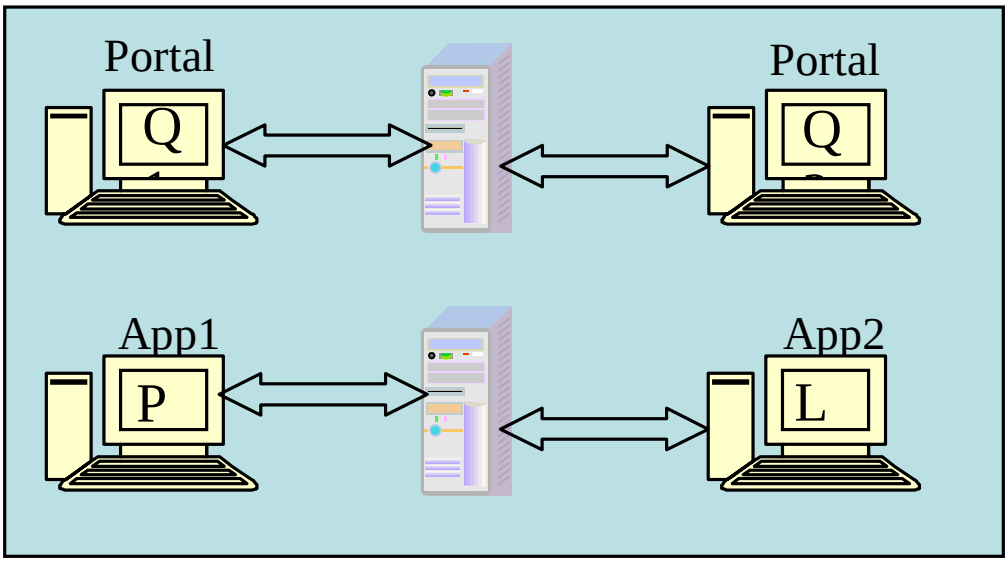

Figure 1: Environment diagram

managers exist on both the App1 and App2 servers, and, if one machine fails, VCS then moves the image of the failing environment to the surviving environment to continue application

processing. When the failed environment is restored, the image that has been moved is then moved back, either at the next scheduled maintenance window or earlier if desired by the application client base.

In order to make MQSeries compatible with VCS, several criteria need to be met:

- It is necessary for VCS to be able to determine whether the WS MQ queue managers are running in a given environment.
- The binary libraries for the WS MQ instance must be available to all the servers that will host queue managers.
- The libraries that allow the WS MQ instance to determine the configuration and status of a given queue manager must be on shared devices so that when a VCS failover occurs, the back-up server can access them to restart the failed WS MQ objects.

Once MQSeries has been installed on a server, and the queue managers created and customized, based on the application requirements, it is then possible to make the installation VCS compliant.

TRANSFORMING THE WS MQ INSTALLATION INTO A VCS-COMPLIANT SYSTEM

In order to transform a normal WS MQ installation into a VCS-compliant installation, the three areas listed above need to be addressed.

IS WS MQ RUNNING?

In order to determine whether WS MQ is running on a given server, VCS needs to be able to check whether a process is active or inactive. It is possible for VCS to check the status of a given process or group of processes. Alternatively, a command that will force a response from the application being managed by VCS can be used for this purpose. First, let's look at the

processes that WS MQ creates on start up in the Solaris environment:

```
[/opt/customer/home/mqm] -> ps –ef | grep mqm
     mqm 22227 22220  0   Dec 30 ?        0:16 amqzlaa0 –mP -fip0
     mqm 22756 22226  0   Dec 30 ?        0:00 /opt/mqm/bin/amqrmppa
-m P
     mqm 22224 22220  0   Dec 30 ?        0:00 /opt/mqm/bin/amqrrmfa
-t2332800 -s2592000 -p2592000 -g5184000 -c3600 –m P
     mqm 22226 22220  0   Dec 30 ?        0:00 /opt/mqm/bin/runmqchi
-m P
     mqm 22220     1  0   Dec 30 ?        0:02 amqzxma0 -m P
     mqm 22232     1  0   Dec 30 ?        0:00 amqpcsea P
     mqm 14394     1  0   Dec 18 ?        0:00 vmstat 3500
     mqm 22222 22220  0   Dec 30 ?        0:00 amqhasmx P /var/mqm
     mqm 22225 22220  0   Dec 30 ?        0:00 /opt/mqm/bin/amqzdmaa
-m P
     mqm 22221 22220  0   Dec 30 ?        0:00 /opt/mqm/bin/amqzfuma
-m P
     mqm 22230     1  0   Dec 30 ?        0:00 runmqlsr -m P -t tcp
-p 1414
     mqm 22229     1  0   Dec 30 ?        0:00 runmqlsr -m P -t tcp
-p 1415
     mqm 22223 22220  0   Dec 30 ?        0:00 amqzllp0 -mP ?
     mqm 22228     1  0   Dec 30 ?        0:00 runmqlsr -m P -t tcp
-p 1416
```

As you can see, the list of processes is somewhat involved, and the list may vary depending on what extra processing requirements are placed on the WS MQ instance. For example, you will notice that three MQ Listeners are running in this instance. This is because of channel requirements for differing external communications sources. Since the list of processes may change as application processing requirements change, having VCS check for a list of processes may lead to false positive or negative reactions as the personality of MQ evolves.

Another alternative for VCS validation would be to look for the primary process in the MQ instance. From the listing above, you will notice that Process ID 22220 (amqzxma0) is the parent process for a number of other processes. It would be possible to code the VCS monitor to look for the amqzxma0 process and thus determine whether WS MQ is active or not. However, you can also see from the listing that a number of other WS MQ processes are started external to amqzxma0. amqpcsea (the

command server) and runmqlsr are examples of processes that are critical to the functioning of WS MQ, but are not spawned by amqzxma0. So, again, if you depend on a static list, you may end up with incorrect reactions from VCS as the MQ instance matures.

Finally, if you specify a list of processes that must be up in order to have VCS declare WS MQ 'active', you still have not ensured that the MQ instance is reacting to external requests for service. Processes can be running on Solaris, but stalled for a number of reasons. In these situations, you would like VCS to detect that there are problems and that failover needs to occur.

So, instead of coding and maintaining lists of processes, the alternative is for VCS to issue a command on the server that WS MQ will accept and that will produce a message indicating the state of the instance. Consider the following scripts:

```
[/opt/customer/home/mqm/Scripts] -> ls -al
total 30
drwxr-xr-x   2 mqm      mqm           512 Nov 18 13:45 .
drwxr-xr-x  13 mqm      mqm          1024 Dec 19 10:38 ..
-rwxr-xr-x   1 mqm      mqm           819 Nov 18 11:43 QMUP.sh
-rw-r--r--   1 mqm      mqm           564 Nov 18 11:35 qmst.mqc

[/opt/customer/home/mqm/Scripts] -> cat QMUP.sh
#!/usr/bin/ksh

############################################################
#
#       Run the MQSC interface against P
#
############################################################

runmqsc P < /opt/customer/home/mqm/Scripts/qmst.mqc

[/opt/customer/home/mqm/Scripts] -> cat qmst.mqc
************************************************************
*
*       qmst.mqc
*
*       This file provides the input commands necessary for
*       MQSC processing to determine the status of the Queue
*        Manager
*
************************************************************
```

```
display qmgr
```

The QMUP.sh script uses the MQSC input file of qmst.mqc in order to run a display command against the WS MQ instance using the RUNMQSC administrative application. When WS MQ is active, the following output is produced:

```
[/opt/customer/home/mqm/Scripts] -> ./QMUP.sh
5724-B41 (C) Copyright IBM Corp. 1994, 2002.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager P.

       :******************************************************
       : *
       : *       qmst.mqc
       : *
       : *       This file provides the input commands
       : *          necessary for MQSC processing to determine the
       : *       status of the Queue Manager
       : *
       : ******************************************************
       :
     1 : display qmgr
AMQ8408: Display Queue Manager details.
   DESCR(Portal QUEUE MANAGER)
   DEADQ(DLQ.P)                            DEFXMITQ( )
   CHADEXIT( )                             CLWLEXIT( )
   CLWLDATA( )                             REPOS(PROD)
   REPOSNL( )
   SSLKEYR(/var/mqm/qmgrs/P/ssl/key)
   SSLCRLNL( )                             SSLCRYP( )
   COMMANDQ(SYSTEM.ADMIN.COMMAND.QUEUE)    QMNAME(P)
   CRDATE(2003-12-24)                      CRTIME(11.22.24)
   ALTDATE(2003-12-24)                     ALTTIME(11.22.32)
   QMID(P_2003-12-24_11.22.24)             TRIGINT(999999999)
   MAXHANDS(256)                           MAXUMSGS(10000)
   AUTHOREV(DISABLED)                      INHIBTEV(DISABLED)
   LOCALEV(DISABLED)                       REMOTEEV(DISABLED)
   PERFMEV(DISABLED)                       STRSTPEV(ENABLED)
   CHAD(DISABLED)                          CHADEV(DISABLED)
   CLWLLEN(100)                            MAXMSGL(99999)
   CCSID(819)                              MAXPRTY(9)
   CMDLEVEL(530)                           PLATFORM(UNIX)
   SYNCPT                                  DISTL(YES)
       :
One MQSC command read.
No commands have a syntax error.
All valid MQSC commands were processed.
```

However, if the WS MQ instance is inactive or non-responsive, the following output is produced:

```
[/opt/customer/home/mqm/Scripts] -> ./QMUP.sh
5724-B41 (C) Copyright IBM Corp. 1994, 2002.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager P.


AMQ8146: WebSphere MQ queue manager not available.

No MQSC commands read.
No commands have a syntax error.
All valid MQSC commands were processed.
```

Having VCS check for the AMQ8146 message, shown above, is relatively simple (according to our VCS expert). This gives us a method for VCS to determine whether WS MQ is active and responsive. Remember, the external processes like the MQ Command Server and the MQ Listeners may not be present or responsive even though WS MQ accepts and returns values for the RUNMQSC command string. Additional checks can be designed to ensure that critical processes of the WS MQ instance are active and working.

VALIDATING THE WS MQ BINARIES…

A WS MQ instance on Solaris uses two sets of libraries for execution. The first group of directories and files is located in */var/mqm*:

```
[/var/mqm] -> ls -al
total 24
drwxrwxr-x   9 mqm      mqm          512 Dec 12 12:21 .
drwxr-xr-x  34 mqm      mqm         1024 Dec 10 14:51 ..
drwxrwxr-x   2 mqm      mqm          512 Nov 17 15:25 config
drwxrwxr-x   3 mqm      mqm          512 Nov 17 15:25 conv
drwxrwxrwx   2 mqm      mqm          512 Dec 12 12:19 errors
drwxrwxr-x   2 mqm      mqm          512 Nov 17 15:25 exits
drwxrwxr-x   2 mqm      mqm          512 Dec 12 12:59 log
 -rw-rw-r--   1 mqm      mqm         2211 Dec 12 13:13 mqs.ini
drwxrwxr-x   3 mqm      mqm          512 Dec 12 13:00 qmgrs
drwxrwxrwx   2 mqm      mqm          512 Nov 17 15:25 trace
```

The directories and files in */var/mqm* are used to control the 'personality' of the WS MQ instance. The *mqs.ini* file contains a

list of all the queue managers that have been created on the server. The log directory contains the WS MQ execution logs. Other directories are used for supporting files of various types and purposes.

The second group of directories and files is located in */opt/mqm*:

```
[/opt/mqm] -> ls -al
total 34
dr-xr-xr-x  12 mqm      mqm          512 Nov 20 11:14 .
drwxr-xr-x  29 mqm      mqm         1024 Dec 10 14:52 ..
dr-xr-xr-x  14 mqm      mqm          512 Nov 17 15:25 READMES
drwxr-xr-x  12 mqm      mqm          512 Nov 17 15:25 README.client
dr-xr-xr-x   2 mqm      mqm         1536 Nov 20 11:14 bin
dr-xr-xr-x   2 mqm      mqm         2048 Nov 20 11:14 inc
drwxr-xr-x   4 mqm      mqm          512 Nov 17 15:25 java
dr-xr-xr-x   3 mqm      mqm         1024 Nov 20 11:14 lib
dr-xr-xr-x   2 mqm      mqm          512 Nov 17 15:25 licenses
  -rw-r--r--   1 mqm      mqm           18 Sep 27 01:46 ptf_installed
  -rw-r--r--   1 mqm      mqm          182 Sep 27 01:46 ptf_supersede
dr-xr-xr-x   6 mqm      mqm         1536 Nov 20 11:14 samp
dr-xr-xr-x   6 mqm      mqm          512 Nov 17 15:27 ssl
dr-xr-xr-x   2 mqm      mqm          512 Nov 17 15:25 tivoli
```

The directories and files in */opt/mqm* are used to run the WS MQ instance itself and include the bin directory where all the application binaries are stored. Additionally, the *java* and *lib* directories are used to support external applications that access WS MQ.

With access to these two groups of directories, it is possible to start and run the queue managers that have been defined within the WS MQ instance. One of the critical files in this process is the */var/mqm/mqs.ini* file. An example of this file follows:

```
[/var/mqm] -> cat mqs.ini
#********************************************************************#
#*                                                                  *#
#* <START_COPYRIGHT>                                                *#
#* Licensed Materials - Property of IBM                             *#
#*                                                                  *#
#* 63H9336                                                          *#
#* (C) Copyright IBM Corporation 1994, 2000                         *#
#*                                                                  *#
#* <END_COPYRIGHT>                                                  *#
#*                                                                  *#
#********************************************************************#
#********************************************************************#
```

```
#* Module Name: mqs.ini                                               *#
#* Type       : WebSphere MQ Machine-wide Configuration File        *#
#* Function   : Define WebSphere MQ resources for an entire machine *#
#*                                                                   *#
#*********************************************************************#
#* Notes      :                                                      *#
#* 1) This is the installation time default configuration           *#
#*                                                                   *#
#*********************************************************************#
AllQueueManagers:
  #*********************************************************************#
#* The path to the qmgrs directory, below which queue manager data  *#
#* is stored                                                         *#
#*********************************************************************#
   DefaultPrefix=/var/mqm

ClientExitPath:
   ExitsDefaultPath=/var/mqm/exits

LogDefaults:
   LogPrimaryFiles=3
   LogSecondaryFiles=2
   LogFilePages=1024
   LogType=CIRCULAR
   LogBufferPages=0
   LogDefaultPath=/var/mqm/log
QueueManager:
   Name=P
   Prefix=/var/mqm
   Directory=P
```

In order to start a queue manager, or run a utility against it, you must have an entry in the *mqs.ini* file for the queue manager that you wish to work with. The last four lines in the listing above are the minimum definition of a queue manager.

SHARING WS MQ CONFIGURATION LIBRARIES

In order for VCS to correctly execute the RUNMQSC command mentioned above, it will be necessary for the WS MQ instance on a server to know about any queue manager(s) that you will wish to validate. WS MQ uses the *mqs.ini* file to verify that a queue manager name is valid for utility or start processing. If you attempt to execute the RUNMQSC command against a queue manager, say X, which does not have an entry in the *mqs.ini* file, you get a message similar to the following:

```
[/var/mqm] -> runmqsc X
5724-B41 (C) Copyright IBM Corp. 1994, 2002.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager X.


AMQ8118: WebSphere MQ queue manager does not exist.

No MQSC commands read.
No commands have a syntax error.
All valid MQSC commands were processed.
```

Since the VCS validation script for WS MQ should generate the message AMQ8146 as indicated above, the output produced is incorrect and will cause problems when VCS attempts to verify that an instance of queue manager X is not running.

If we update the *mqs.ini* file, referenced above with the minimum information required to define queue manager X, we should see a definition similar to the following:

```
[/var/mqm] -> cat mqs.ini
#********************************************************************#
#*                                                                  *#
#* <START_COPYRIGHT>                                                *#
#* Licensed Materials - Property of IBM                             *#
#*                                                                  *#
#* 63H9336                                                          *#
#* (C) Copyright IBM Corporation 1994, 2000                         *#
#*                                                                  *#
#* <END_COPYRIGHT>                                                  *#
#*                                                                  *#
#********************************************************************#
#********************************************************************#
#* Module Name: mqs.ini                                             *#
#* Type       : WebSphere MQ Machine-wide Configuration File        *#
#* Function   : Define WebSphere MQ resources for an entire machine *#
#*                                                                  *#
#********************************************************************#
#* Notes      :                                                     *#
#* 1) This is the installation time default configuration           *#
#*                                                                  *#
#********************************************************************#
AllQueueManagers:
  #******************************************************************#
#* The path to the qmgrs directory, below which queue manager data  *#
#* is stored                                                        *#
#********************************************************************#
   DefaultPrefix=/var/mqm
```

```
ClientExitPath:
   ExitsDefaultPath=/var/mqm/exits

LogDefaults:
   LogPrimaryFiles=3
   LogSecondaryFiles=2
   LogFilePages=1024
   LogType=CIRCULAR
   LogBufferPages=0
   LogDefaultPath=/var/mqm/log
QueueManager:
   Name=P
   Prefix=/var/mqm
   Directory=P
QueueManager:
   Name=X
   Prefix=/var/mqm
   Directory=X
```

Figure 10 – mqs.ini Updated

When the RUNMQSC utility is executed to validate the status of queue manager X, output similar to the following is produced:

```
[/var/mqm] -> runmqsc X
5724-B41 (C) Copyright IBM Corp. 1994, 2002.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager X.


AMQ8146: WebSphere MQ queue manager not available.

No MQSC commands read.
No commands have a syntax error.
All valid MQSC commands were processed.
```

Now that VCS can verify whether an instance of a queue manager is running, we next need to make the changes that will allow 'failed over' queue managers to be started when server outages occur.

The objects that define the processing capabilities or 'personality' of a queue manager are defined in a discrete set of libraries within the */var/mqm* directory structure as shown below:

```
[/var/mqm] -> ls -al
total 28
drwxrwxr-x   9 mqm      mqm           512 Jan  7 16:19 .
drwxr-xr-x  35 root     sys          1024 Nov 17 15:21 ..
drwxrwxr-x   2 mqm      mqm           512 Sep  5 14:12 config
drwxrwxr-x   3 mqm      mqm           512 Sep  5 14:12 conv
drwxrwxrwx   2 mqm      mqm          3072 Jan 13 10:38 errors
```

```
drwxrwxr-x   2 mqm      mqm          512 Sep  5 14:12 exits
drwxrwxr-x   4 mqm      mqm          512 Jan  7 16:19 log
 -rw-rw-r--   1 mqm      mqm         2279 Jan  8 14:48 mqs.ini
drwxrwxr-x   5 mqm      mqm          512 Jan  7 16:19 qmgrs
drwxrwxrwx   2 mqm      mqm          512 Sep  5 14:12 trace
```

For each queue manager defined to the WS MQ instance, subdirectories are created in the */var/mqm/qmgrs* directory and in the */var/mqm/log* directory. Additionally, if site-specific exits have been coded, these will be located in the */var/mqm/exits* directory. When a queue manager has been defined using the GUI, command line, or RUNMQSC utility, a new subdirectory is created in the */var/mqm/qmgrs* directory:

```
[/var/mqm/qmgrs] -> ls -al
total 12
drwxrwxr-x   5 mqm      mqm          512 Jan  7 16:19 .
drwxrwxr-x   9 mqm      mqm          512 Jan  7 16:19 ..
drwxrwxr-x   8 mqm      mqm          512 Sep  5 15:59 @SYSTEM
lrwxrwxrwx   1 mqm      mqm           27 Dec 12 16:49 P
drwxrwxr-x  21 mqm      mqm          512 Jan 12 15:22 L
```

In the code above, two queue managers, P and L, have been created and the associated subdirectories have been created. Additionally, when the queue managers are created, new subdirectories are created in the */var/mqm/log* directory:

```
[/var/mqm/log] -> ls -al
total 1Ø
drwxrwxr-x   4 mqm      mqm          512 Jan  7 16:19 .
drwxrwxr-x   9 mqm      mqm          512 Jan  7 16:19 ..
lrwxrwxrwx   1 mqm      mqm           25 Dec 12 16:5Ø P
  drwxrwx--   3 mqm      mqm          512 Jan  7 16:18 L
```

These directories exist on the server where the WS MQ instance has been installed, and are essential to execute the functions of the queue manager. If you refer back, the *mqs.ini* entry for a queue manager contains three parameters:

- Name – this corresponds to the WS MQ queue manager name.
- Prefix – this is the directory path to the WS MQ libraries.
- Directory – this is the name of the subdirectory that contains the ‘personality’ files.

It has already been established above that updating the *mqs.ini* file is essential for the RUNMQSC utility to decide that a queue manager can be run on a given server under a WS MQ instance. In order for the queue manager to actually run, the WS MQ instance must be able to access the files in the */var/mqm* subdirectories (qmgrs and log) that define the queue manager components.

In the system design shown in Figure 1, the portal queue manager and the application queue managers each have a shared disk array. This is configured so that both systems see the shared arrays and the high-level directory structures. So, from the App1 server the shared disk array shows both of the high-level directories:

```
[/<shared array path>] -> ls -al
total 8
drwxr-xr-x   4 mqm      mqm          512 Oct  9 15:01 .
drwxr-xr-x  19 root     root         512 Jan  6 21:08 ..
drwxr-xr-x   2 root     other        512 Oct  9 15:01 App1
drwxr-xr-x   2 root     other        512 Oct  9 15:01 App2
```

When you change directory to the App1 high-level directory in the shared disk array, you can then see the WS MQ */var* directory as well as the */var/qmgrs* and */var/log* subdirectories where the files that are to be shared across both servers will exist:

```
[/<shared array path>/App1] -> ls -al
total 124900
drwxr-xr-x   5 mqm      mqm         1024 Dec 15 11:03 .
drwxr-xr-x   4 mqm      mqm          512 Oct 17 10:17 ..
drwxr-xr-x   4 mqm      mqm           96 Dec 12 16:58 var

[/<shared array path>/App1/var] -> ls -al
total 2
drwxr-xr-x   4 mqm      mqm           96 Dec 12 17:19 .
drwxr-xr-x   7 mqm      mqm         1024 Dec 15 10:45 ..
drwxr-xr-x   3 mqm      mqm           96 Dec 12 17:19 log
drwxr-xr-x   3 mqm      mqm           96 Dec 12 17:19 qmgrs
```

If you change the directory to the App2 high-level directory on the shared device, and failover has not yet been triggered by VCS, then you see only the directory structures as shown:

```
[/<shared array path>/App2] -> ls -al
```

```
total 4
drwxr-xr-x   2 root     other        512 Oct  9 15:Ø1 .
drwxr-xr-x   4 mqm      mqm          512 Oct  9 15:Ø1 ..
```

Now that all the disk definitions have been described, let's consider how we modify the WS MQ installation to allow the queue manager to be moved from one server to another.

Since the personality of the queue manager is defined by the subdirectory entries in */var/mqm/qmgrs* and */var/mqm/log*, we can move those directories to the */var* directory on the shared disk array (as shown below) so that either system can see the appropriate directories as needed. In Unix, the easiest way to move the directories is to use the **tar** utility to package up the directory and all its subdirectories, and then use the **tar** utility again to extract the files and put them into their new location. The commands, if you have created a queue manager named P, will be similar to the following:

```
cd /var/mqm/qmgrs

tar cf PQMGR.tar P

mv PQMGR.tar /<shared array path>/App1/var/qmgrs

cd /<shared array path>/App1/var/qmgrs

tar xf PQMGR.tar

cd /var/mqm/log

tar cf PLog.tar P

mv PLog.tar /<shared array path>/App1/var/qmgrs

cd /<shared array path>/App1/var/qmgrs

tar xf PLog.tar
```

After you have packaged the directories and copied them to the shared disk array, you need to update the local libraries so that WS MQ understands where the new physical directories reside. Remove the existing physical directories in the */var/mqm/qmgrs* and */var/mqm/log* directories for the queue managers that are to be VCS managed, and insert symbolic links pointing to the

shared disk arrays instead. The command to create a symbolic link in Solaris is:

```
ln –s (existing target) (softlink name)
```

So in the environment we have described, the command sequence that you would use to create the links between the WS MQ installation directories for queue manager P and the shared disk array would be:

```
cd /var/mqm/qmgrs

ln –s /<shared disk array path>/App1/var/qmgrs/P P

cd /var/mqm/log

ln –s /<shared disk array path>/App1/var/log/P P
```

In addition to defining the queue manager entries for the local system, in this case App1, you will also need to define the queue manager entries for the remote queue manager, which runs on App2, since it will be started on the local environment in the event of a VCS failover. Modifying the commands listed above, to point to the App2 directories, and changing the queue manager name from P to L, will allow you to create softlink entries for the remote system after creating the softlink for the local system. A softlink does not have to have a target that can be resolved, until an application attempts to access the link.

After you have run both sets of softlink definition commands, you will have created the directory entries listed below on the local disks where the WS MQ instance has been created:

```
[/var/mqm/qmgrs] -> ls -al
total 10
drwxr-xr-x   3 mqm      mqm           512 Dec 12 17:21 .
drwxr-xr-x   9 mqm      mqm           512 Jan  7 16:55 ..
drwxr-xr-x   8 mqm      mqm           512 Sep  9 10:54 @SYSTEM
lrwxrwxrwx   1 mqm      mqm            29 Dec 12 17:21 P -> /<shared disk
path>/App1/var/qmgrs/P
lrwxrwxrwx   1 mqm      mqm            29 Dec 12 17:20 L -> /<shared disk
path>/App2/var/qmgrs/L

[/var/mqm/log] -> ls -al
total 8
```

```
drwxr-xr-x   2 mqm      mqm           512 Dec 12 17:20 .
drwxr-xr-x   9 mqm      mqm           512 Jan  7 16:55 ..
lrwxrwxrwx   1 mqm      mqm            27 Dec 12 17:20 P -> /<shared disk
path>/App1/var/log/P
lrwxrwxrwx   1 mqm      mqm            27 Dec 12 17:20 L -> /<shared disk
path>/App2/var/log/L
```

Once your VCS administrator has created the necessary scripts and entries to define the WS MQ environment to VCS, the changes outlined above should allow you to fail over the WS MQ queue managers from one system to another.

CONCLUSION

In addition to the definitions listed above, there are operational considerations that need to be planned for a failover scenario. These include application recovery for those programs that work with WS MQ, restarting channels within the WS MQ instance, and re-establishing connections between the WS MQ queue manager(s) and their external clients. The implementation of a WS MQ cluster can improve the resiliency of the managed environment and minimize the mean time to failure resolution.

Issues of VCS script installations and customization have been deliberately omitted, because those skills can normally be found in environments where VCS is used. The issue of failing over network connections from one server to another has also been omitted, because this belongs in the domain of your network and security administrators.

However, once you have implemented the recommendations listed above, your WS MQ instance and the associated queue managers will be VCS compliant, and you will be able to manage them just as any other application or server that can be controlled by VCS.

Aaron Cain
Independent Consultant
The Performance Edge Limited (UK)

Visual debugging in IBM WebSphere Business Integration Message Broker V5.0 Toolkit

This article is most suited to those with some previous working experience and knowledge of using IBM MQSeries Integrator V2.1 and WebSphere MQ products. The WebSphere Business Integration Message Broker V5.0 will be referred to as Message Broker V5.0 throughout the article.

The Visual Debugger has received a makeover in Message Broker V5.0. All the debugging for message flows is done in the Flow Debug perspective. This new design also introduces the IBM Agent Controller, which is a requirement for debugging and will be explained later.

The following sections will describe what the Agent Controller is and how to use the Flow Debug perspective to debug a message flow. 'VDBMessageFlow' and 'VDBSubMessageFlow' message flows are used, with the assumption that they exist in the workspace, for illustrative purposes.

REQUIREMENTS

The following set-up is required before proceeding:

1. A configuration manager created with its queue manager and listener running.
2. A broker created with its queue manager and listener running. (Less set up is required if configuration manager and broker share the same queue manager and listener.)
3. A domain connection connected to the configuration manager and a broker to deploy to in the toolkit.
4. A deployed message flow to debug.
5. A running Agent Controller.

AGENT CONTROLLER

Once installed the IBM Agent Controller is a daemon process running in the background. The Agent Controller is required to use the Flow Debugger and should be installed on the system where the broker will be used. When a broker is created and started, it registers itself with the Agent Controller running on the system. Thus when a message flow is deployed to an execution group of the registered broker, the user can attach to the Flow Engine of the deployed message flow. Once attached to the Flow Engine, the user can debug the message flows running. The Agent Controller service can be seen in the services window shown in Figure 1.

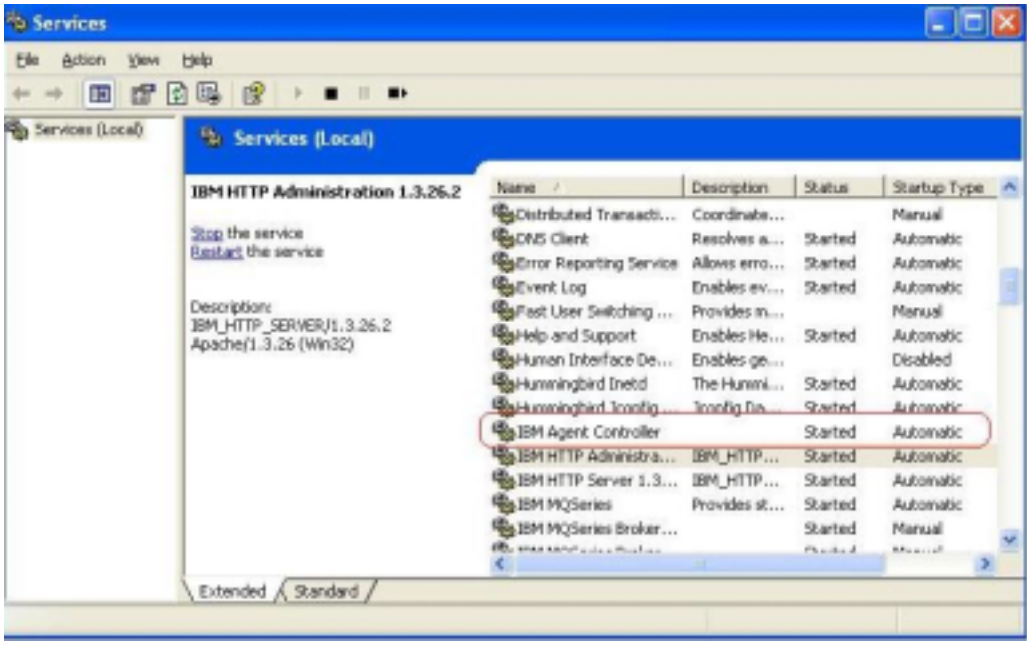

Figure 1: Services window

FLOW DEBUG PERSPECTIVE

The Flow Debug perspective is a dedicated perspective for flow debugging. The following steps will show how to open and use the flow debug perspective for debugging.

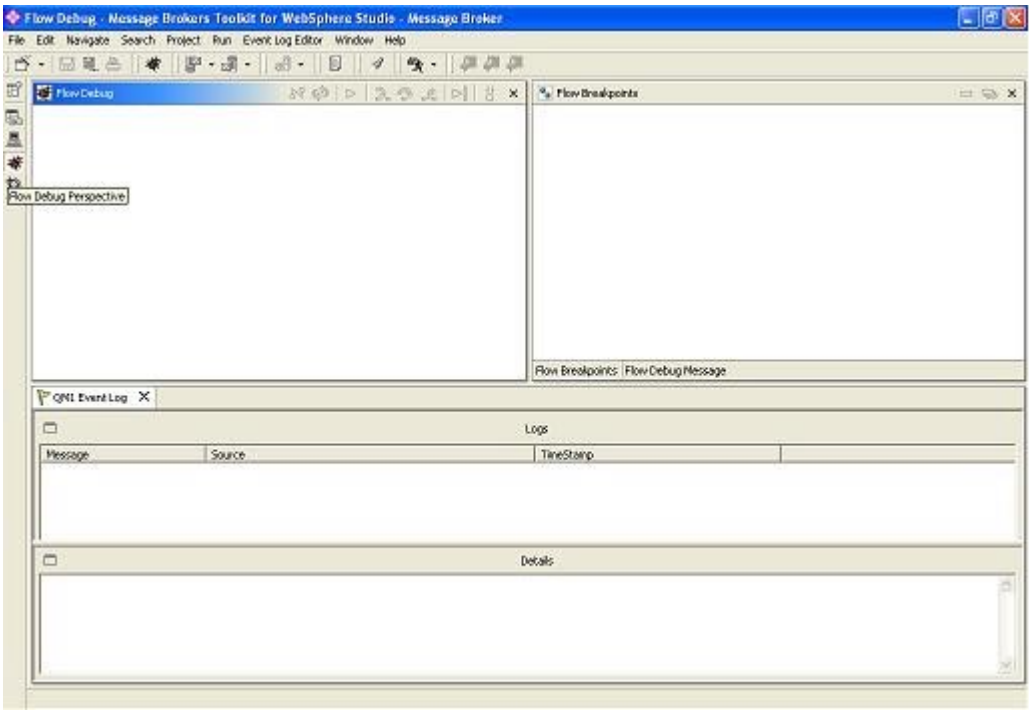

Figure 2: Flow Debug perspective

Set up the flow for debugging:

1 Once the message flow has been successfully deployed, open the Flow Debug perspective – **Window/Open Perspective/Other**. Select *Flow Debug* from the list and

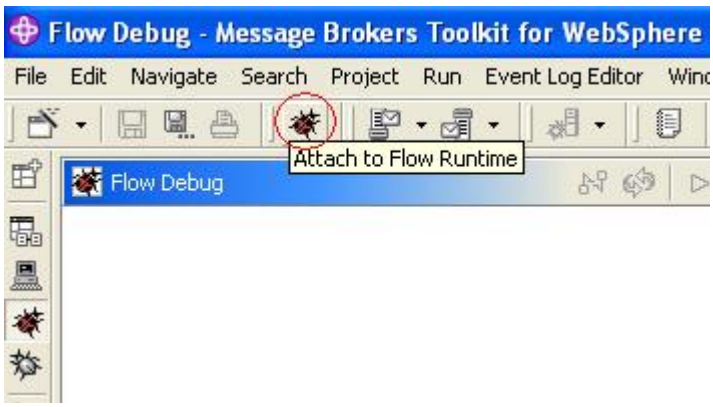

Figure 3: Attach to Flow runtime

click *OK*. The Flow Debug perspective opens as shown in Figure 2.

2 Click on the *Attach to Flow runtime* toolbar button to attach to the execution group where the deployed message flow is running as shown in Figure 3.

3 In the *Attach to the flow engine* window, select *localhost* and click *Next* as shown in Figure 4. This will connect to a broker local to the system. To connect to a remote broker, enter the name or IP address of the machine hosting the remote

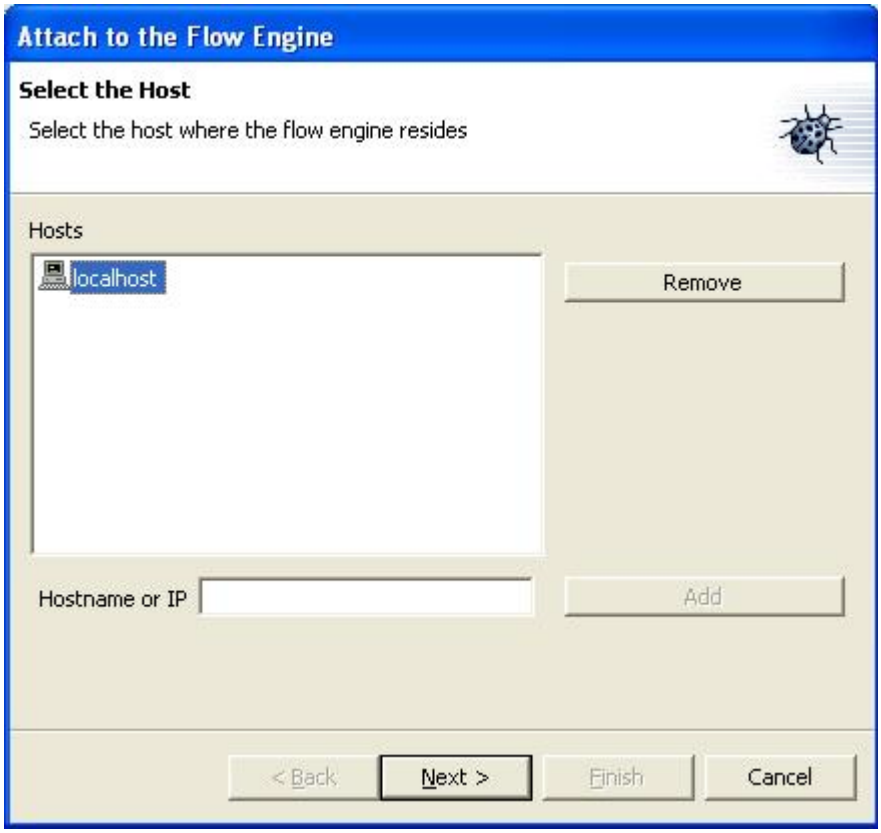

Figure 4: Attaching a host

broker in the *Hostname or IP* box and click *Add*. This will add the remote machine to the list of hosts to be used.

4 Select from the list the flow engine where the flow resides

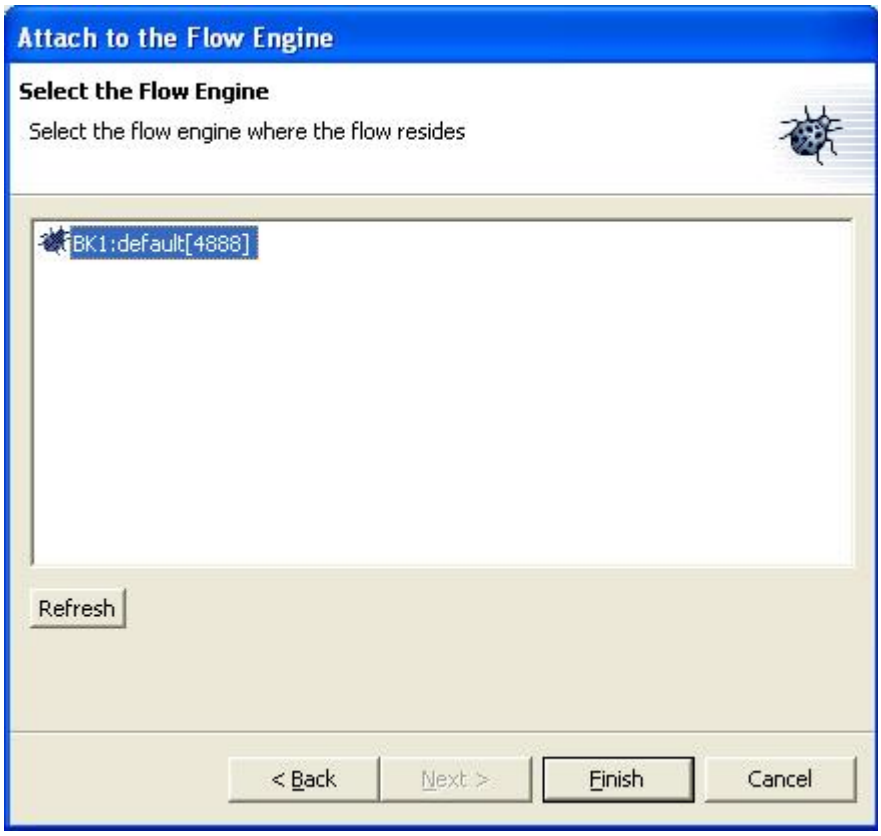

Figure 5: Select the flow engine

and click *Finish*. In the snapshot shown in Figure 5, *BK1* is the name of the broker and *default* is the name of the execution group to where the flow is deployed to. The

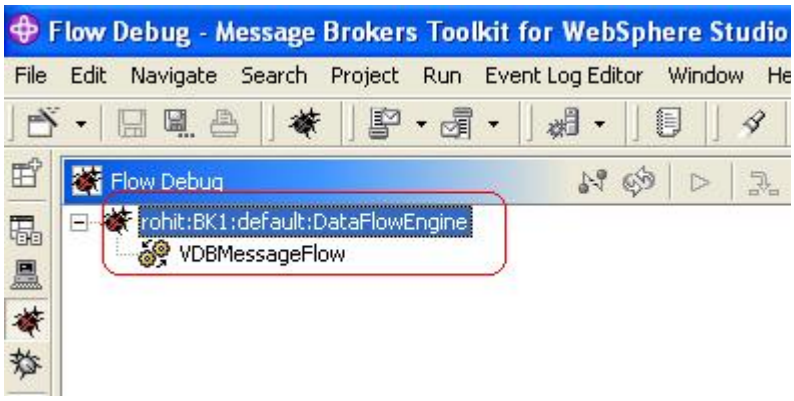

Figure 6: Flow Debug pane

number *4888* in the snapshot is the process id of the broker's actual flow engine process.

5 The message flow appears in the Flow Debug pane shown in Figure 6. It also shows the name of the flow engine. (Note: in the snapshot 'rohit' is the name of the broker's host machine.)

6 Double-click the message flow in the Flow Debug pane to open the message flow in the flow editor pane. To debug the

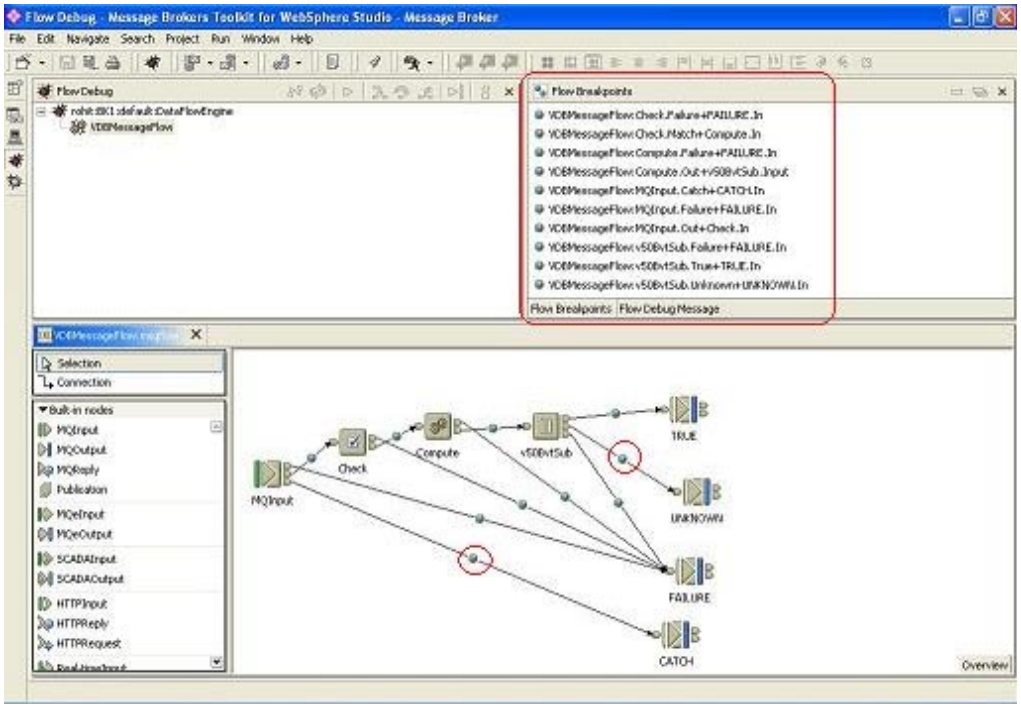

Figure 7: Adding breakpoints

various nodes add breakpoints after or before the nodes. This can be done by right-clicking on a node and selecting *Add Breakpoint before/after* from the list. Breakpoints added to the flow can be seen as a list in the *Flow Breakpoints* pane shown in Figure 7. Breakpoints are circled in the Figure.

Debugging the flow:

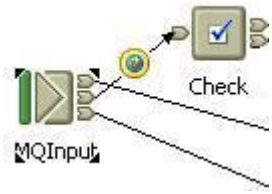

Figure 8: Breakpoint flow

1 Once the required breakpoints have been inserted into the flow, start the message flow (if it is not already started), and put a message on the queue of the input node. The message will go through the flow stopping at the breakpoints in the order it hits them. When the message hits a breakpoint the breakpoint is highlighted by a yellow circle around the breakpoint in the flow editor pane, as shown in Figure 8.

2 When the message stops at a breakpoint, there are various

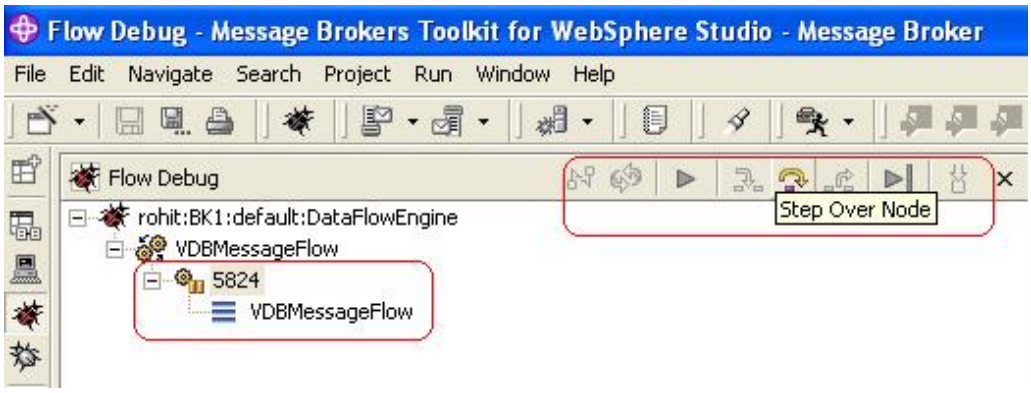

Figure 9: Step Over Node

user actions to choose from. They are on the top-right corner of the Flow Debug pane and include actions like *Step Over Node*, *Step into Subflow*, *Step Out of Subflow*, *Step over node*, *Run to next breakpoint*, and *Run to completion*. *Step Over Node* is illustrated in Figure 9.

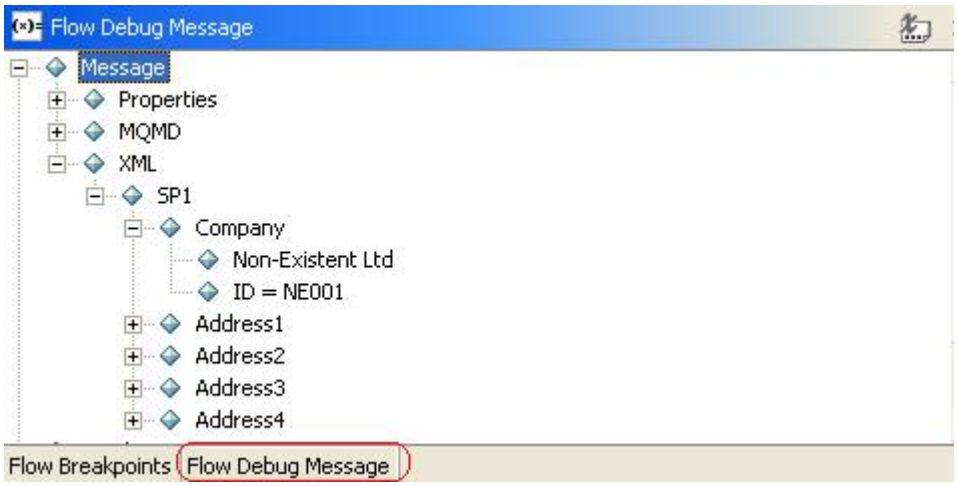

Figure 10: Flow Debug Message pane

3 The state of the message being debugged can be seen under the *Flow Debug Message* tab in the *Flow Debug Message* pane, as shown in Figure 10. Here the user can see how the message is created or changed as it goes

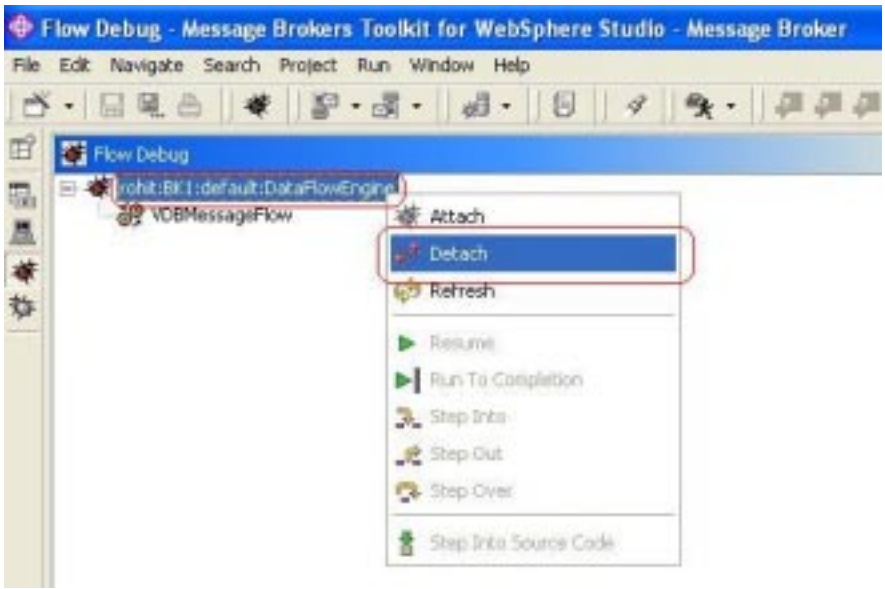

Figure 11: Detach

through the flow. As mentioned earlier you can see the message go through the flow in the editor pane via the yellow circled breakpoints.

4 To detach from the flow engine once debugging is complete, right-click on the connection details in the Flow Debug pane and select *Detach*, as illustrated in Figure 11.

Rohit Bhasin
Software Engineer, WebSphere MQI GUI Test Team
IBM Hursley (UK)

Micro Focus and iWay Software have announced a strategic alliance focused on extending COBOL applications to work with disparate back-end and e-business systems. The companies plan to work together to offer the easiest way to implement integrated software solutions for organizations seeking to unlock existing value from legacy COBOL applications.

The joint solution interoperates seamlessly with all major integration platforms across J2EE and .NET environments. Micro Focus' COBOL source code can be fully reused with iWay's adapters to support any type of integration project, including composite business applications and Web services initiative.

For further information contact:
Micro Focus, Old Bath Road, Newbury, Berks RG14 1QN, UK.
Tel: (01635) 32646.
URL: http://www.microfocus.com/press/releases/20040217.asp.

* * *

KANA has announced KANA portlets, which integrate KANA IQ software (an application for enterprise knowledge management and service optimization) with IBM WebSphere Portal. KANA's portlets for KANA IQ are designed to help companies accelerate successful deployment of knowledge-powered service applications. It was jointly developed with IBM.

KANA portlets empower WebSphere Portal customers to integrate KANA knowledge-powered customer service applications into their WebSphere Portal. Organizations can now leverage KANA IQ to provide agent-assisted and customer self-service while using WebSphere Portal.

KANA IQ is an integral part of the KANA iCARE suite, an enterprise software suite made up of modular CRM applications.

For further information contact:
KANA, 181 Constitution Drive, Menlo Park, CA 94025, USA.
Tel: (650) 614 8300.
URL: http://www.kana.com.

* * *

MQSoftware has announced management and monitoring support for WebSphere MQ for z/Linux for Q Pasa!, the company's real-time middleware monitoring solution, which works with the WebSphere family of products to address the management and delivery of business services.

The move is aimed at allowing organizations to have a monitoring and configuration management solution for properly managing and deploying WebSphere MQ across the enterprise.

Q Pasa! provides broad WebSphere MQ-related platform coverage, to help customers realize the total enterprise reach of all their legacy and middleware applications.

For further information contact:
MQSoftware, 1660 South Highway 100, Suite 400, Minneapolis, MN 55416, USA.
Tel: (952) 345 8720.
URL: http://www.mqsoftware.com/news/newsDetail.jsp?id=54.