77

MQ update

November 2005

In this issue

A itc PUBLICATION

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

Colin Smith
E-mail: info@xephon.com

Disclaimer

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

Subscriptions and back-issues

A year's subscription to *MQ Update*, comprising twelve monthly issues, costs $380.00 in the USA and Canada; £255.00 in the UK; £261.00 in Europe; £267.00 in Australasia and Japan; and £265.50 elsewhere. In all cases the price includes postage. Individual issues, starting with the July 2000 issue, are available separately to subscribers for $33.75 (£22.50) each including postage.

Contributions

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

MQ Update on-line

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

Printed in England.

Control client connections using WebSphere MQ Internet Pass-Thru

INTRODUCTION

One of the features of WebSphere MQ is the ability of an application to connect to a WebSphere MQ server using an MQI channel.

This article explains how, by implementing an MQIPT security exit, WebSphere MQ Internet Pass-Thru (MQIPT) can be used to control when client connections are allowed to be made to a queue manager. Many configuration features are available to MQIPT, but this article focuses only on its security exit. An MQIPT security exit has a different interface from a WebSphere MQ channel exit and does not require a corresponding security exit at the other end of the channel.

By their very nature, client connections are normally short lived, and it is very common to have many client connections active at the same time.

WHAT IS MQIPT?

MQIPT is a category 3 WebSphere MQ SupportPac, which means it is fully supported and can be downloaded for free from http://www.ibm.com/software/integration/support/supportpacs. It was designed to be installed in the Demilitarized Zone (DMZ) of a firewall and act as a 'proxy' for WebSphere MQ traffic. It will accept a connection request from the WebSphere MQ client and route it to the desired destination based on its predefined configuration data. Once MQIPT has established the connection and the handshaking process has completed, WebSphere MQ messages are sent and received as on any other WebSphere MQ channel connection.

The only change required in order to use MQIPT is to the

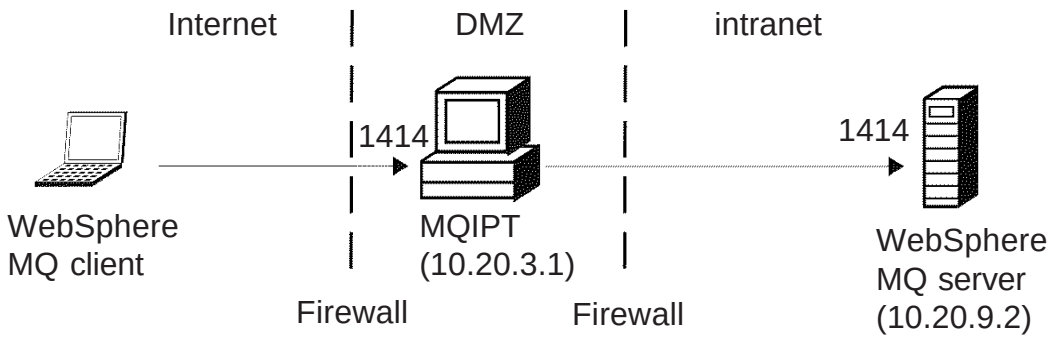

Figure 1: Sample set-up

WebSphere MQ CONNAME of the channel that's being started. In this example, the CONNAME of the CLNTCONN channel must point to the remote MQIPT instead of the WebSphere MQ server. The MQIPT will be configured to connect to the destination queue manager.

The sample set-up is shown in Figure 1.

For this sample, the following assumptions are made:

- You are familiar with defining queue managers, queues, and channels on WebSphere MQ.
- You have already installed a WebSphere MQ client and server.
- The client, QM, and MQIPT are installed on separate machines.
- MQIPT is installed in a directory called *C:\mqipt*.
- You are familiar with putting messages on a queue using the **amqsputc** command.
- The Java SDK is installed on the MQIPT machine.

CONFIGURING WEBSPHERE MQ

On the WebSphere MQ server you need to do the following:

- Define a queue manager called MQIPT.QM1.
- Define a server connection channel called MQIPT.CONN.CHANNEL.
- Define a local queue called MQIPT.LOCAL.QUEUE.
- Start a TCP/IP listener for MQIPT.QM1 on port 1414.

SAMPLE SECURITY EXIT

The security exit will be called when a connection request is received on a specific MQIPT route and the response from the security exit determines whether the connection request is accepted or rejected. For the purposes of this example, change the START_TIME and END_TIME variables to control when client connections are allowed.

This sample code is stored in a file called *ChannelAccess.java* and should be copied to *c:\mqipt\exits* on the MQIPT machine

```
/*
 * Sample program for use with IBM WebSphere MQ internet pass-thru
 * 5639-L92
  * (C) COPYRIGHT International Business Machines Corp., 2005
 * All Rights Reserved * Licensed Materials - Property of IBM
 * This sample program is provided AS IS and may be used, executed,
 * copied and modified without royalty payment by customer
 * (a) for its own instruction and study,
 * (b) in order to develop applications designed to run with an IBM
 *     WebSphere product, either for customer's own internal use or for
 *     redistribution by customer, as part of such an application, in
 *     customer's own products.
 */
import java.io.*;
import java.util.*;
import com.ibm.mq.ipt.IPTException;
import com.ibm.mq.ipt.SecurityExit;
import com.ibm.mq.ipt.SecurityExitResponse;
/**
 * This is a sample test program to show how to use a WebSphere MQ
 * Internet Pass-Thru (MQIPT) security exit to control connections
 * based on the time
 * of day and the name of the channel.
 */
public class ChannelAccess extends SecurityExit {
  /**  Only control connections to this channel   */
```

```
  private static final String CHANNEL_NAME = "MQIPT.CONN.CHANNEL";
  /** Start of business   */
  private static final int START_TIME = 8;
  /** Close of business   */
  private static final int END_TIME = 22;
  /**
   * Default constructor used by MQIPT to create a new instance of this
   * class when a route is started.
   */
  public ChannelAccess() {
  }
  /**
   * This method is called when a route is being started and should be
   * used to perform any initialization (eg reading configuration data)
   * and place itself in a ready state to validate connection requests.
   * @throws IPTException to signal a failure
   */
  public void init() throws IPTException {
    // Trace useful info
    System.out.println("ChannelAccess ready for work");
    // Perform any initialization here
    return;
  }
  /**
   * This method is called to validate a connection request.
   * @return SecurityExitResponse
   */
  public SecurityExitResponse validate() {
    SecurityExitResponse secExitResponse = null;
    // Get channel name
    String channelName = getChannelName();
    try {
      // Check channel name
      if (channelName.equalsIgnoreCase(CHANNEL_NAME)) {
        // Get the current timestamp
        Calendar d = Calendar.getInstance();
        // Get the hour of day
        int hh = d.get(Calendar.HOUR_OF_DAY);
        // Open for buiness ?
        if ((hh > START_TIME) && (hh < END_TIME)) {
          // Allow the connection request
    secExitResponse = new SecurityExitResponse(SecurityExitResponse.OK);
          // Trace result
System.out.println("Connection allowed\n" + secExitResponse.toString());
        }
        else {
          // Reject the connection request
          secExitResponse = new SecurityExitResponse(
                                SecurityExitResponse.NOT_AUTHORIZED);
          // Trace result
```

```
   System.out.println("Connection not allowed - not open for business");
        }
      }
      // Channel name not known
      else {
        // Reject the connection request
        secExitResponse = new SecurityExitResponse(
                              SecurityExitResponse.NOT_AUTHORIZED);
        // Trace result
    System.out.println("Connection not allowed - unknown channel name" +
                          channelName);
      }
    }
    catch (IPTException ie) {
      // Trace result
       System.out.println("Error creating SecurityExitResponse\n" +
ie.toString());
    }
    return secExitResponse;
  }
}
```

The security exit needs to be compiled before it is used. This is achieved by opening a command prompt:

```
c:
cd \mqipt\exits
set CLASSPATH=.;c:\mqipt\lib\MQipt.jar
javac ChannelAccess
```

CONFIGURING MQIPT

MQIPT needs to have a route defined to the target WebSphere MQ server.

Note that the *Destination* and *DestinationPort* properties need to reflect your own server and port address.

1 Define a route with a security exit:

Edit *c:\mqipt\mqipt.conf* and add a route definition:

```
[route]
ListenerPort=1415
Destination=10.20.9.2
DestinationPort=1414
SecurityExit=true
SecurityExitName=ChannelAccess
```

2 Start MQIPT by opening a command prompt:

```
c:
cd \mqipt\bin
mqipt ..
```

The following messages will be seen on the MQIPT console:

```
5639-L92 (C) Copyright IBM Corp. 2000, 2004 All Rights Reserved
MQCPI001 WebSphere MQ internet pass-thru Version 1.3.2 starting
MQCPI004 Reading configuration information from C:\mqipt\mqipt.conf
MQCPI011 The path C:\mqipt\logs will be used to store the log files
MQCPI006 Route 1415 has started and will forward messages to :
MQCPI034 ....10.20.9.2(1414)
MQCPI035 ....using MQ protocols
MQCPI079 ....using security exit c:\mqipt\exits\ChannelAccess
MQCPI080 ......and timeout of 5 second(s)
MQCPI078 Route 1415 ready for connection requests
```

CONFIGURING WEBSPHERE MQ CLIENT

The queue manager and MQIPT have both been configured and are now ready to accept connections. Note that the IP address defined in MQSERVER needs to reflect your own server address. The WebSphere MQ client can be started by:

- Opening a command prompt:

```
SET MQSERVER=TEST.CONN.CHANNEL/TCP/10.20.3.1
```

- Putting a WebSphere MQ message on the queue by issuing the command:

```
amqsputc MQIPT.LOCAL.QUEUE MQIPT.QM1
```

Depending on the values defined for the *START_TIME* and *END_TIME*, this command will succeed or fail.

SUMMARY

Using a specially written security exit, MQIPT can be used to control when client connections to a queue manager are allowed.

Idle connections can be disconnected by using the *IdleTimeout* property.

Removing active connections (after close of business for the day) is probably not recommended and these should be allowed to naturally terminate, either by inactivity after a predefined idle timeout or by the client ending the application.

This scenario can be extended so that MQIPT can be used for QM-to-QM connections as well as client connections.

Phil Blake
Software Engineer
IBM (UK)

Effective ways of debugging WebSphere MQ on distributed platforms

ABSTRACT

WebSphere MQ, formerly MQ Series, has come a long way since its launch in 1993 as a middleware product from IBM. With its vast amount of functionality – supporting multi-platforms, protocols, databases, and programming languages – it is critically important to obtain debug information whenever WebSphere MQ applications fail or don't perform to their specifications. Since the tools and procedures will vary from platform to platform, depending on the problem/situation, it's difficult for an individual or team to have the expertise and appropriate tools on all platforms. In this case, an insight into efficient methods or procedures for debugging WebSphere MQ on distributed platforms is very much required.

Even though WebSphere MQ provides First-Failure Support Technology (FFST – usually called FDC) and trace facility, it will not be much use in the following problematic situations:

- Application hangs
- Memory leaks
- Interoperability problems.

This article gives very detailed procedures for the above-mentioned situations, which will help customers who are working on WebSphere MQ applications. All the required scripts (referred to in this article) are also provided.

PROBLEMATIC SCENARIOS

As mentioned already, WebSphere MQ applications will support several environments, and the problems vary from one environment to another. However, most problems encountered are application hangs, WebSphere MQ command hangs, dumping core files, interoperability issues, and WebSphere MQ API memory leaks. This article does not cover customer-written application problems.

SOLUTION

Detailed information for debugging WebSphere MQ applications during problems and hangs is given by platform. First let us look at application or WebSphere MQ command hangs and core file generation. In these situations even tracing will not help to debug. So, OS-specific commands or debuggers (like **dbx** and **dde**) need to be used. Memory leaks and interoperability problems will be covered in later sections.

APPLICATION HANGS

AIX

In AIX, the debugger **dbx** can be used to find the command that is hanging and the function that is causing the problem. The following **dbx** command sequence will help to identify the problem in depth.

Syntax:

```
dbx -a <process id>
```

```
1    (dbx)where
2    (dbx)map
3    (dbx)thread
4    (dbx)thread info <threadid>
     Repeat this for all the threads.
5    (dbx)thread current <threadid>
6    (dbx)where
     Repeat step 5 and 6 for all the threads.
7    (dbx) detach.
```

Please note that if you **quit**, **dbx** kills the running process. So use the **detach** command to detach the process instead of killing it.

In the case of core file generation, the core file will be passed to the debugger along with the command that is responsible for the core. The command can be found by looking at the core file header.

Syntax:

```
dbx <exe file with full path>  <corefile>
```

```
     dbx> where
     dbx>thread
     dbx>registers  (not must)
     dbx>list
     dbx> what is $t<no>       // for getting thread no 'no' information
     dbx> dump                 // only for analysis because lot of
                                          information will be displayed.
     dbx> detach
```

Solaris

The Solaris platform also provides a **dbx** debugger. In addition to that, the **pstack** and **pmap** commands are available to track the command flow.

Syntax:

```
dbx -a <process id>
```

```
     dbx> where
     dbx>thread
```

```
dbx>registers  (not must)
dbx>list (or)listi
dbx> what is $t<no>       // for getting thread no 'no' information
dbx> dump                 // only for analysis because lot of
                                    information will be displayed.
dbx> detach
```

Usage of the **pstack** and **pmap** commands:

```
/usr/proc/bin/pstack  <process id> <output file>
/usr/proc/bin/pmap  -rlx  <process id> <output file>
```

HP-UX

HP Unix provides the **dde** debugger and **tusc** tools to monitor the command flow. The procedure is given below.

Syntax:

```
dde -ui line -attach <process id>  <exe file full path>
```

```
dde> tb
dde> list thread -full
dde> tb -thread all
dde> quit
```

Tusc can be used as shown below.

Syntax:

```
tusc -aeEvlpT -o <output.file> <pid>
```

```
#Eg: /opt/tusc/bin/tusc -aeEvlpT -o /mqdata/tusc.rcdmqimg.out 23Ø88
```

Linux

All Linux operating systems provide the GNU debugger **gdb**, which can be invoked to monitor the command flow.

Syntax:

```
gdb <exe file with full path>  <corefile>
```

```
gdb>where
gdb>info threads
gdb>info sharedlibrary
gdb>info @t<no>      // where no is thread number.
gdb>detach
```

Please use the **detach** command to detach the process,

instead of using **quit**, which kills the running process.

Compaq Tru64 v4.x or v5.x

Tru64 Unix platforms support the **dbx** debugger and the procedure is given here.

Syntax:

```
dbx -pid <pid of the Qmgr process> <executable>

     >where
     >tlist
     >tstack
     >plist
     >detach
```

Repeat the **dbx** command for all queue managers and application processes.

Windows

Dr Watson is the tool to be used for debugging on Windows. In the command prompt type **drwtsn32 -i** to initialize Dr Watson as the default debugger:

1 Invoke Dr Watson by clicking **Start**, selecting **Run...**, and typing **drwtsn32.exe**.

2 Adjust all the parameters in the dialog box to suit your needs, eg the number of instructions to save, number of errors to save, log file path, etc. These settings are stored in *\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\DrWatson*. View the log file once an application error has occurred.

The log file contains the following information:

1 Error number.

2 System information.

3 Task list.

4 List of modules that the program had loaded.

5 Processor and memory-related information like stack trace, contents of register, etc.

6 And finally, the symbol table.

Looking at the above information, the exact location of an error can be found to debug the MQ application.

MEMORY LEAKS

It's important to monitor the product or application processes to find memory leaks, which impact on the stability and reliability of the product or application. The following procedure helps to find memory leaks of the processes.

AIX

To find memory leaks from any process on the AIX platform, attach the process id to the **svmon** command and send the output to a file (**svmon** supports AIX V5.0 onwards). This command captures and analyses a snapshot of virtual memory. Although the **ps** command provides data to monitor process memory usage, it is better to use the **svmon** command provided by AIX.

Syntax:

```
svmon -P <pid of queue manager>
```

Note: **svmon** should be run as root.

Solaris, HP-UX, and Linux

Most of the Unix platforms provide the **ps** command to monitor the process details, including memory usage. Using the **ps –eal <pid>** or **ps-eaf <pid>** commands, we can collect the memory usage of the process. The script below will help you to collect data for one or more processes on different platforms:

```
#!/bin/ksh
while true
do
ps -elf | grep <Process id> | awk '{ print $10 }' >>  <pid.out>
```

```
ps -aux | grep <Process id> | awk '{ print $10 }' >>  <pid.out>
                                                          // for Linux.

sleep 10
done
```

Note: change the frequence of **sleep *xx*** based on the monitoring time.

Compaq Tru64

On Tru64 platforms, memory leaks of any process can be obtained using the **vmstat** or **ps** commands.

Syntax:

```
vmstat -i <pid>
```

or:

```
ps -eal (m) -p <pid>
```

Windows

Tools are available from third parties to find the memory leaks from processes running on Windows systems.

INTEROPERABILITY PROBLEMS

Most of the WebSphere MQ applications are used to interact with different operating environments through available channel types. It's very difficult to locate interoperability problems because they will arise for a variety of reasons including authentication, protocols, and operating system-specific issues. If the problems are authentication or operating system-related, MQ provides error logs with specific information. Also, the methods described above can be used for debugging. If the problems are related to the protocol, then you need to debug on both protocol (TCP/IP) and application views. All major platforms provide **netstat -a** to check the ports' status and the TCP/IP trace facility allows you to take a protocol trace at both ends.

BIBLIOGRAPHY

- TCP/IP error codes: http://www-3.ibm.com/cgi-bin/db2www/data/db2/udb/winos2unix/support/document.d2w/report?fn=db2m0db2tcp.htm.
- HP-UX tusc tool (HP specific): http://eigen.ee.ualberta.ca/hppd/hpux/Sysadmin/tusc-6.5.
- Dr Watson tool: http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/drwatson_overview.mspx.
- All platform commands: http://www.bhami.com/rosetta.html.

Srihari Kulkarni and Mekala V Reddy
IBM (India)

Obtain MQ queue statistics on the Linux platform

Not all IT departments have the luxury of having a commercial product to benchmark their MQSeries set-up. This article shows a simple but effective way to obtain the following statistical data from MQ queues on the Linux platform:

- Run date and time
- Queue manager name
- Queue name
- Number of MQPUTs during the last interval
- Number of MQGETs during the last interval
- The current queue depth
- The highest queue depth reached during the last interval
- Interval time.

The interval in this context is either the period between running the statistics utility or, if it is the first time, the time since starting the queue manager.

ENVIRONMENT

The utility was specifically written for MQ on the Linux operating system, but also runs under Windows. It was tested under:

- Suse Linux V9.2, kernel 2.6.8-24.17.
- Redhat Enterprise Linux AS release 3 (Taroon Update 4), kernel 2.4.21-27.0.2.
- Windows XP.
- IBM WebSphere MQSeries V5.3, CSD 10 (also tested from CSD6 upwards).

SOURCE

The attached source code, Linux_qstats.c, can be compiled using either MS Visual C++ or the GNU C compiler. If using the GNU compiler, compile and link the source code as follows:

```
gcc -o Linux_qstats Linux_qstats.c -lmqm
```

The interface to MQ is the so-called MQAI, which is an administration interface on top of the PCF interface. This article will not explain how the MQAI works.

The comments in the source code should make it self-explanatory.

RUNNING THE UTILITY

The utility basically takes as input the following parameters:

- Queue manager
- Input file (list of queue names)
- Output file.

Please make sure that the queue names in the input file do not exceed the maximum length of 48 characters. As an enhancement you could allow an '*' to mean 'all queues', but be aware that although this works under Windows, you will need to use '*' for Unix environments.

The output file is a CSV file, and this has been done deliberately. It can then be imported into a spreadsheet and analysed, or, as we have done, can be used as input to a database utility (either DB2 or Oracle) to store in a database.

The command used is basically RESET QSTATS and it seems strange that only MQ on the IBM mainframe has this command as part of the standard MQSC. On the distributed platforms it has to be done programmatically.

An alternative to this utility is to write an API exit.

Whenever the utility is run, the various counters are zeroed, but note that in addition to the RESET QSTATS fields, an inquiry is made into the 'current' queue depth as well.

Be aware that if you do have a third-party product (like QPasa for example), it too issues RESET QSTATS and this cannot be switched off, so you cannot run both this utility and another one issuing the same command. (Well, you can, but the results would be confusing!)

SAMPLE RUN

```
Linux_qstats T840A mqstats_queues.txt stats.out
Linux_qstats: Obtain MQ Queue Statistics
No. of records read 2 & written 2
Linux_qstats - end of processing.
Input File mqstats_queues.txt
RUUD1
RUUD2

Output File stats.out
2005-08-15,19.53.12,T840A,RUUD1,0,0,2,2,8
2005-08-15,19.53.12,T840A,RUUD2,0,0,0,0,8
2005-08-15,20.23.45,T840A,RUUD1,10,0,12,12,1832
2005-08-15,20.23.45,T840A,RUUD2,0,0,0,0,1832
```

CONCLUSION

You don't need an expensive third-party product to obtain MQSeries statistics. In fact, it is often difficult to extract that sort of information from external products.

By running the supplied utility on a regular basis (using, for example, the Unix CRON facility) data can either be stored by appending it onto a CSV file for later analysis in MS Excel, or it can be stored in a database against which SQL statements can be issued.

LINUX_QSTATS.C

```
/*************************************************************************/
/* Program name: Linux_qstats.c                                          */
/* Description : Extract and display MQ Queue Statistics                 */
/* Date        : August 2005 by Ruud van Zundert                         */
/* Includes                                                              */
/*************************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <time.h>
#include <cmqc.h>                          /* MQI                        */
#include <cmqcfc.h>                        /* PCF                        */
#include <cmqbc.h>                         /* MQAI                       */
/* Function prototypes                                                   */
void CheckCallResult(MQCHAR *, MQLONG , MQLONG);
void get_QDepth(MQCHAR *);
void get_Timestamp(char*);
void mqstuff();
MQLONG qDepth;                          /* point-in-time queue depth  */
MQHCONN hConn,hInq;                     /* handle to MQ connection    */
MQLONG reason;                          /* reason code                */
MQLONG connReason;                      /* MQCONN reason code         */
MQLONG compCode;                        /* completion code            */
MQHBAG adminBag = MQHB_UNUSABLE_HBAG;   /* admin bag for mqExecute    */
MQHBAG responseBag = MQHB_UNUSABLE_HBAG;/* response bag for mqExecute */
MQHBAG qAttrsBag;                       /* bag containing q attributes*/
MQHBAG errorBag;                  /* bag containing cmd server error */
MQLONG mqExecuteCC;                     /* mqExecute completion code  */
MQLONG mqExecuteRC;                     /* mqExecute reason code      */
MQLONG i,n;                             /* loop counter               */
MQLONG numberOfBags;              /* number of bags in response bag  */
MQLONG qNameLength;                     /* Actual length of q name    */
```

```
MQLONG O_options,C_options;             /* MQOPEN/MQCLOSE options     */
MQOD   odG = {MQOD_DEFAULT};            /* Object Descriptor for GET  */
MQOD   odI = {MQOD_DEFAULT};            /* Object Descriptor (INQUIRE)*/
MQMD   md = {MQMD_DEFAULT};             /* Message Descriptor         */
MQGMO  gmo = {MQGMO_DEFAULT};           /* get message options        */
MQLONG Select[2];                       /* attribute selectors        */
MQLONG IAV[2];                          /* integer attribute values   */
char   buffer[100];                     /* message buffer             */
MQLONG buflen;                          /* buffer length              */
MQLONG messlen;                         /* message length received    */
char   run_timestamp[27];               /* full run timestamp         */
char   mq_timestamp[27];                /* first mq msg put date/time */
int    This_Queue_OK,readIn=0,readOut=0;
char   ioarea[49],*p;
int    ignore_errors=0;
FILE   *fileout = NULL, *filein = NULL;
MQCHAR qmName[MQ_Q_MGR_NAME_LENGTH+1]=""; /* default QMgr name        */
MQLONG qTime,qHiDepth,qPuts,qGets;      /* returned values            */
MQCHAR qName[MQ_Q_NAME_LENGTH+1];   /*name of queue extracted from bag*/
int readrc=0;
char   c;
/* Function: main                                                     */
int main(int argc, char *argv[])
{
   printf("Linux_qstats: Obtain MQ Queue Statistics\n\n");
   if (argc < 4) {
     printf("Please supply 3 parms: 1=qmgr, 2=input file, 3=output
file.\n*");
     printf("Optional 4th parm, 4=ignore errors y/n(default n)\n");
     exit(4);
   }
   // Connect to the queue manager                                    */
   strncpy(qmName, argv[1], (size_t)MQ_Q_MGR_NAME_LENGTH);
   MQCONN(qmName, &hConn, &compCode, &connReason);
   // Report the reason and stop if the connection failed.            */
   if (compCode == MQCC_FAILED) {
      CheckCallResult("Queue Manager connection", compCode, connReason);
      exit( (int)connReason);
   }
   memset (ioarea, '\0', sizeof(ioarea) );            /* initialize
ioarea   */
   fileout = fopen( argv[3], "a" );            /* open, append, create*/
   get_Timestamp(run_timestamp);
   if (argc > 4) {                         /* should errors be ignored?*/
     if (strncmp(argv[4],"y",1) == 0)
       ignore_errors = 1;
   }
   filein = fopen( argv[2], "r" );         /*open input file read-only*/
   if ( filein == NULL ) {
       printf("unable to open file %s\n", argv[2]);
```

```
        exit(9);
    }
    p=ioarea;                       /* point at io area         */
    n=Ø;                            /* intialize counter        */
    do {
      c = fgetc (filein);           /* get 1 char at a time     */
      if ( c == '\n' ) {            /* newline? yes,process     */
        readIn++;                   /* incr queue input count   */
        mqstuff();                  /* get the mq statistics    */
        p=ioarea;                   /* repoint to io area       */
        memset (p, '\Ø', 2 );       /* init first part          */
        n=Ø;                        /* reinit counter           */
      }
      else {
        *p=c;                       /* store char in io area    */
        p++;                        /* bump up the pointer      */
        n++;                        /* increment counter        */
      }
    } while (c != EOF);             /* read until end of file   */
    fclose(filein);
    /* Disconnect from the queue manager               */
    MQDISC(&hConn, &compCode, &reason);
    CheckCallResult("Disconnect from Queue Manager", compCode, reason);
    fclose(fileout);
    printf("No. of records read %d & written %d\n", readIn, readOut);
    printf("Linux_qstats - end of processing.\n");
    return Ø;
} /* end main */
//---------------------------
void mqstuff()
{
    mqTrim(MQ_Q_MGR_NAME_LENGTH, qmName, qmName, &compCode, &reason);
    /* Create an admin bag for the mqExecute call                      */
    mqCreateBag(MQCBO_ADMIN_BAG, &adminBag, &compCode, &reason);
    CheckCallResult("Create admin bag", compCode, reason);
    /* Create a response bag for the mqExecute call                    */
    mqCreateBag(MQCBO_ADMIN_BAG, &responseBag, &compCode, &reason);
    CheckCallResult("Create response bag", compCode, reason);
    /* Put the generic queue name into the admin bag                   */
    mqAddString(adminBag, MQCA_Q_NAME, MQBL_NULL_TERMINATED, ioarea,
&compCode, &reason);
    CheckCallResult("Add q name", compCode, reason);
     /*****************************************************************/
    /* Send the command to find all the local queue names and queue    */
    /* depths. The mqExecute call creates the PCF structure required,  */
    /* sends it to the command server, and receives the reply from the */
    /* command server into the response bag. The attributes are        */
    /* contained in system bags that are embedded in the response bag, */
    /* one set of attributes per bag.                                  */
     /*****************************************************************/
```

```
   mqExecute(hConn,                    /* MQ connection handle         */
             MQCMD_RESET_Q_STATS,
             MQHB_NONE,                /* No options bag               */
             adminBag,              /* Handle to bag containing commands */
             responseBag,       /*Handle to bag to receive the response*/
             MQHO_NONE,         /* Put msg on SYSTEM.ADMIN.COMMAND.QUEUE*/
             MQHO_NONE,         /* Create a dynamic q for the response  */
             &compCode,         /* Completion code from the mqexecute   */
             &reason);          /* Reason code from mqexecute call      */
   /* Check the command server is started. If not exit.               */
   if (reason == MQRC_CMD_SERVER_NOT_AVAILABLE)
   {
      printf("Please start the command server: <strmqcsv QMgrName>\n");
      MQDISC(&hConn, &compCode, &reason);
     CheckCallResult("Disconnect from Queue Manager", compCode, reason);
      exit(98);
   }
    /*****************************************************************/
   /* Check the result from mqExecute call. If successful find the    */
   /*current depths of all the local queues. If failed find the error.*/
    /*****************************************************************/
   if ( compCode == MQCC_OK )              /* Successful mqExecute    */
   {
     mqTrim(MQ_Q_NAME_LENGTH, qmName, qmName, &compCode, &reason);
     /* Count the number of system bags embedded in the response bag  */
     /* from the mqExecute call. The attributes for each queue are in */
     /* a separate bag.  */
     mqCountItems(responseBag, MQHA_BAG_HANDLE, &numberOfBags,
&compCode, &reason);
     CheckCallResult("Count number of bag handles", compCode, reason);
     for ( i=Ø; i<numberOfBags; i++)
     {
        /*************************************************************/
       /* Get the next system bag handle out of the mqExecute response*/
       /* bag. This bag contains the queue attributes                 */
        /*************************************************************/
       mqInquireBag(responseBag, MQHA_BAG_HANDLE, i, &qAttrsBag,
&compCode, &reason);
       CheckCallResult("Get the result bag handle", compCode, reason);
       /* Get the queue name out of the queue attributes bag          */
     mqInquireString(qAttrsBag, MQCA_Q_NAME, Ø, MQ_Q_NAME_LENGTH, qName,
                     &qNameLength, NULL, &compCode, &reason);
       CheckCallResult("Get queue name", compCode, reason);
     /* Get the time since last reset out of the queue attributes bag */
  mqInquireInteger(qAttrsBag, MQIA_TIME_SINCE_RESET, MQIND_NONE, &qTime,
                      &compCode, &reason);
       CheckCallResult("Get resettime", compCode, reason);
       /* Get the depth out of the queue attributes bag               */
   mqInquireInteger(qAttrsBag, MQIA_HIGH_Q_DEPTH, MQIND_NONE, &qHiDepth,
                      &compCode, &reason);
```

```
        CheckCallResult("Get depth", compCode, reason);
    /*****************************************************************/
    /* Get the no. of MQPUTs since last reset out of the queue attr bag*/
    /*****************************************************************/
      mqInquireInteger(qAttrsBag, MQIA_MSG_ENQ_COUNT, MQIND_NONE, &qPuts,
                       &compCode, &reason);
        CheckCallResult("Get MQPUTs", compCode, reason);
        /*****************************************************************/
        /* Get the depth out of the queue attributes bag                 */
        /*****************************************************************/
      mqInquireInteger(qAttrsBag, MQIA_MSG_DEQ_COUNT, MQIND_NONE, &qGets,
                       &compCode, &reason);
        CheckCallResult("Get MQGETs", compCode, reason);
        /*****************************************************************/
        /* Use mqTrim to prepare the queue name for printing.            */
        /* Print the result.                                             */
        /*****************************************************************/
          get_QDepth(qName);
        if (This_Queue_OK) {
          mqTrim(MQ_Q_NAME_LENGTH, qName, qName, &compCode, &reason);
          fprintf(fileout, "%s,%s,%s,%d,%d,%d,%d,%d\n",
run_timestamp,qmName,qName,qPuts,qGets,qDepth,qHiDepth,qTime  );
          readOut++;
        } /* end-if */
      }  /* end for loop */
    } /* end-if exec */
    else                                        /* Failed mqExecute     */
     if ( !ignore_errors ) {
      printf("Call to get queue attributes failed: Completion Code = %d :
Reason = %d\n",
                    compCode, reason);
      /*****************************************************************/
      /* If the command fails, get the system bag handle out of the    */
      /* mqexecute response bag. This bag contains the reason from the */
      /* command server why the command failed.                        */
      /*****************************************************************/
      if (reason == MQRCCF_COMMAND_FAILED)
      {
        mqInquireBag(responseBag, MQHA_BAG_HANDLE, Ø, &errorBag,
&compCode, &reason);
        CheckCallResult("Get the result bag handle", compCode, reason);
        /*****************************************************************/
        /* Get the completion code and reason code, returned by the    */
        /* command server, from the embedded error bag.                  */
        /*****************************************************************/
   mqInquireInteger(errorBag, MQIASY_COMP_CODE, MQIND_NONE, &mqExecuteCC,
                       &compCode, &reason );
        CheckCallResult("Get the completion code from the result bag",
compCode, reason);
      mqInquireInteger(errorBag, MQIASY_REASON, MQIND_NONE, &mqExecuteRC,
```

```
                        &compCode, &reason);
       CheckCallResult("Get the reason code from the result bag",
compCode, reason);
       printf("Error returned by the command server: Completion Code =
%d : Reason = %d\n",
                mqExecuteCC, mqExecuteRC);
     } /* end-if reason */
    } /* end-if ignore errors */
   /*****************************************************************/
   /* Delete the admin bag if successfully created.                 */
   /*****************************************************************/
   if (adminBag != MQHB_UNUSABLE_HBAG)
   {
      mqDeleteBag(&adminBag, &compCode, &reason);
      CheckCallResult("Delete the admin bag", compCode, reason);
   }
   /*****************************************************************/
   /* Delete the response bag if successfully created.              */
   /*****************************************************************/
   if (responseBag != MQHB_UNUSABLE_HBAG)
   {
      mqDeleteBag(&responseBag, &compCode, &reason);
      CheckCallResult("Delete the response bag", compCode, reason);
   }
} /* end mqstuff */
/*****************************************************************/
void  CheckCallResult(char *callText, MQLONG cc, MQLONG rc)
{
  if (cc == MQCC_OK) This_Queue_OK=1;
  if (reason == MQRC_OBJECT_IN_USE) This_Queue_OK=1;
  if ((cc != MQCC_OK) &&
      (reason != MQRC_UNKNOWN_OBJECT_NAME) &&
      (reason != MQRC_OBJECT_IN_USE) )
    printf("%s failed: Completion Code = %d : Reason = %d\n", callText,
cc, rc);
}
/*****************************************************************/
/* get current timestamp from the Operating System               */
/*****************************************************************/
void get_Timestamp(char* str)
{
struct tm *ptr;
time_t tm;
char str2[4];
tm = time(NULL);
ptr = localtime(&tm);
strftime(str ,100 , "%Y-",ptr);
strftime(str2 ,100 , "%m-",ptr);
strncat(str, str2,3);
strftime(str2 ,100 , "%d,",ptr);
```

```
strncat(str, str2,3);
strftime(str2 ,100 , "%H.",ptr);
strncat(str, str2,3);
strftime(str2 ,100 , "%M.",ptr);
strncat(str, str2,3);
strftime(str2 ,100 , "%S",ptr);
strncat(str, str2,2);
}
/* get current queue depth for relevant queue                    */
void get_QDepth(MQCHAR *qName)
{
  qDepth=0;
  This_Queue_OK=0;           /* assume this is a 'problem' que */
   memset(mq_timestamp,'\0',sizeof(mq_timestamp));
  strncpy(odI.ObjectName,    /* name of queue from message     */
               qName, MQ_Q_NAME_LENGTH);
  O_options = MQOO_INQUIRE   /* open to inquire attributes     */
            + MQOO_BROWSE    /* in order to browse 1st msg     */
            + MQOO_INPUT_SHARED
            + MQOO_FAIL_IF_QUIESCING;
  MQOPEN(hConn,                          /* connection handle   */
         &odI,               /* object descriptor for queue    */
         O_options,          /* open options                   */
         &hInq,              /* object handle for MQINQ        */
         &compCode,          /* completion code                */
         &reason);           /* reason code                    */
  CheckCallResult("MQOPEN Open Queue", compCode, reason);
  if (compCode == MQCC_OK) {
    Select[0] = MQIA_CURRENT_Q_DEPTH;
    MQINQ(hConn,
          hInq,               /* object handle                  */
          1,                  /* Selector count                 */
          Select,             /* Selector array                 */
          1,                  /* integer attribute count        */
          IAV,                /* integer attribute array        */
          0,                  /* character attribute count      */
          NULL,               /* character attribute array      */
          /*  note - can use NULL because charattr count is zero */
          &compCode,          /* completion code                */
          &reason);           /* reason code                    */
    CheckCallResult("MQINQ Inquire on Queue Depth", compCode, reason);
      qDepth = IAV[0];
  }
  C_options = 0;              /* no close options               */
  MQCLOSE(hConn,              /* connection handle              */
          &hInq,              /* object handle                  */
          C_options,
          &compCode,          /* completion code                */
          &reason);           /* reason code                    */
```

```
} /* end get_QDepth */
/* end of program ************************************************/
```

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

WebSphere Business Integration Message Broker – simplified functional validation: part 2

This month we continue our look at creating a test message flow.

Alternatively if you have a flow chart fetish, you can change the orientation to a top-down approach.

But enough of formatting tricks…

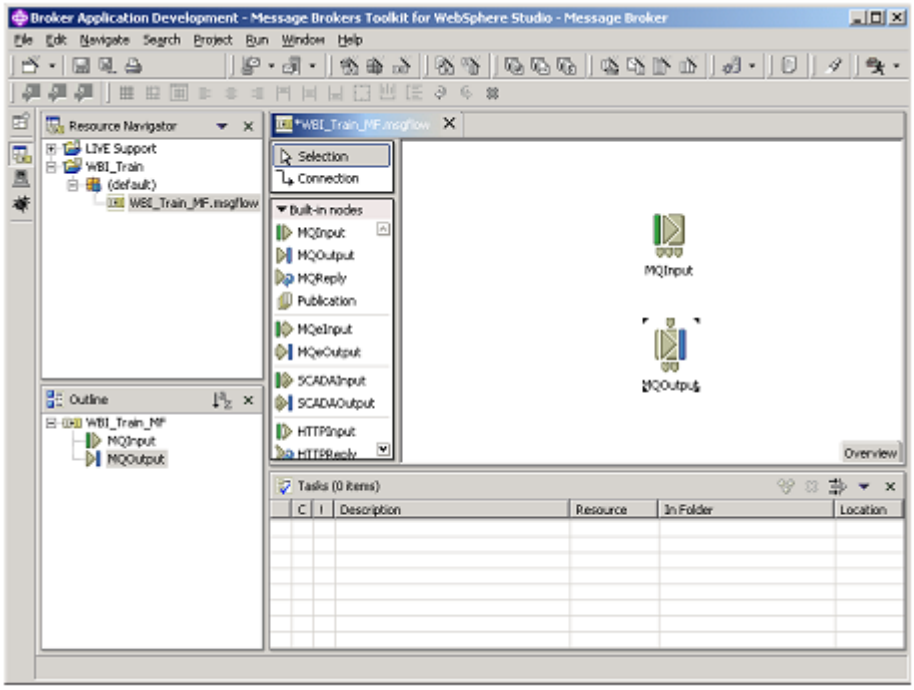

Figure 1: Top-down flowchart

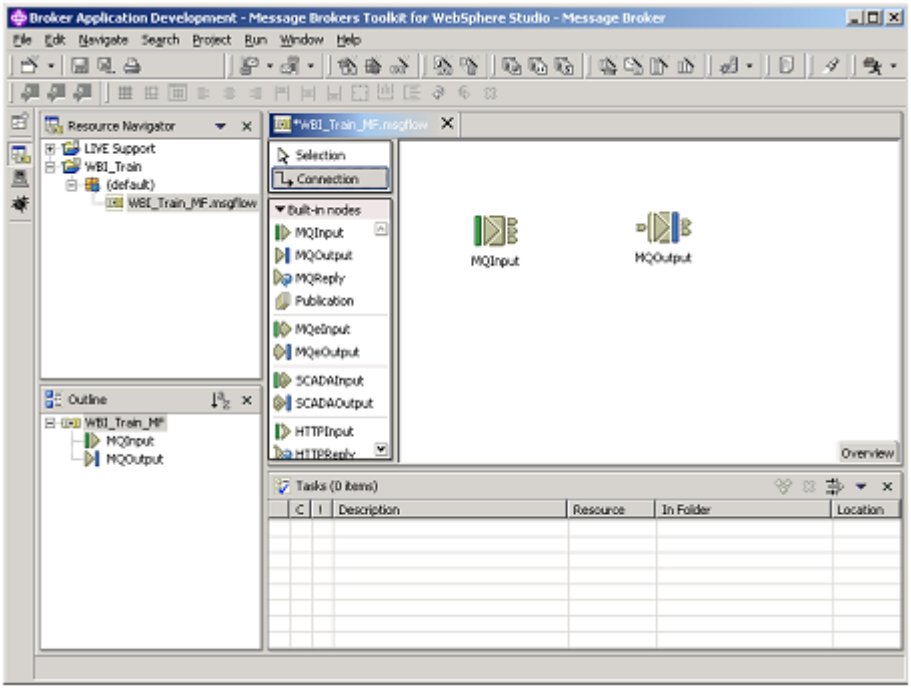

Figure 2: Connection option

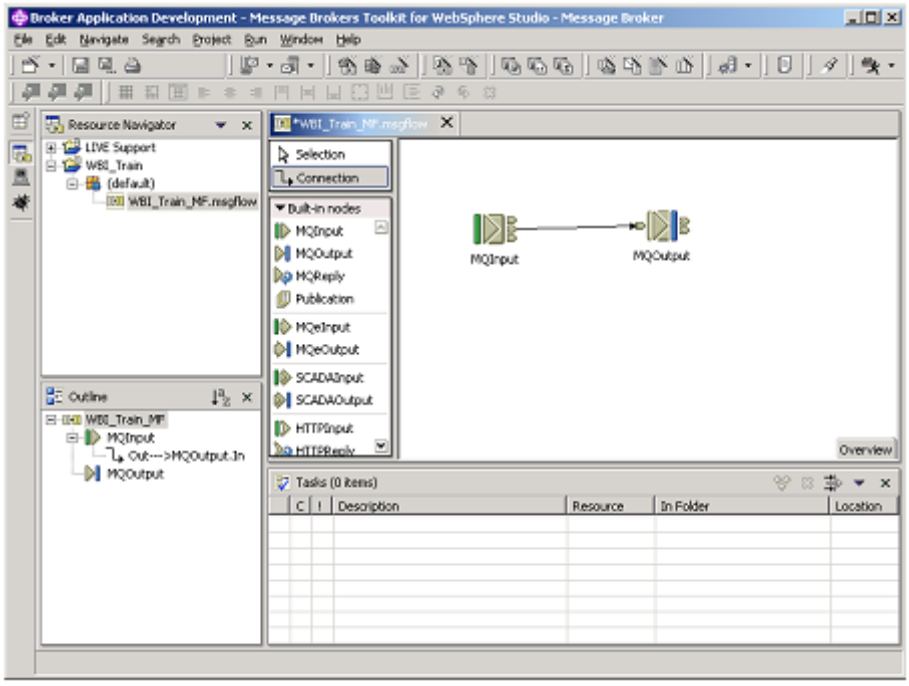

Figure 3: Moving function

Once you have the nodes in place you need to connect them. Just as you did to place the node, you need to click the **Connection** option on the editor menu – see Figure 2. When you move the cursor around the editor screen it will be in the form of a crossed circle when you cannot connect anything and it will change to a plug symbol when you reach an appropriate target and can connect.

Move the cursor to the middle tab on the MQInput icon and left-click to attach the source to the Out function of the MQInput process; then left-click the tab on the left-hand side of the MQOutput icon to have the message move from the MQInput function into the MQOutput function – see Figure 3.

Now, you need to add the logic for the two icons, so that WBIMB knows what you want to do with the messages that arrive on the first queue, and that they are to be redirected to the follow-on queue.

First, you need to change the highlighted menu option in the

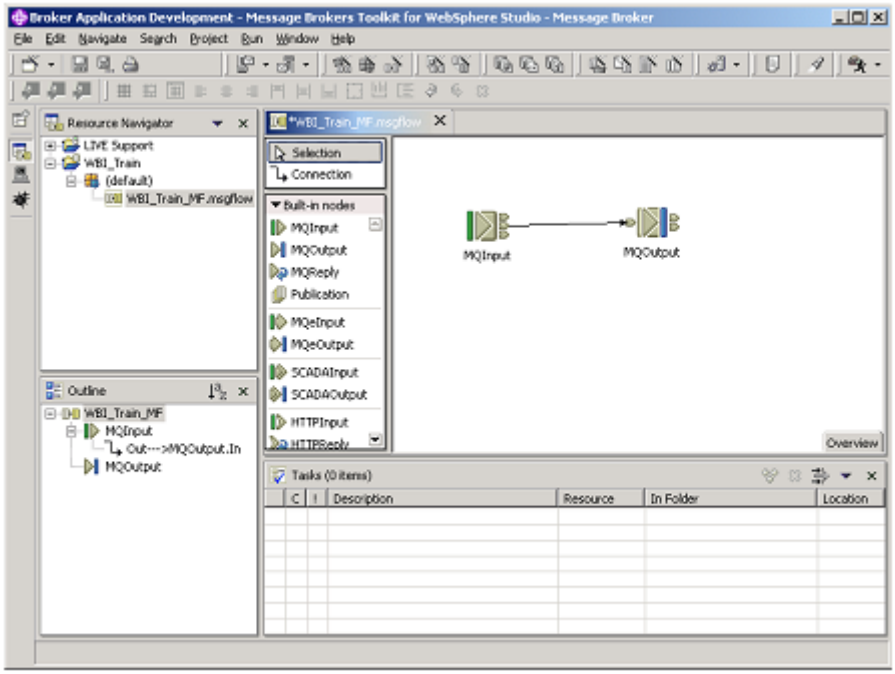

Figure 4: Selection function

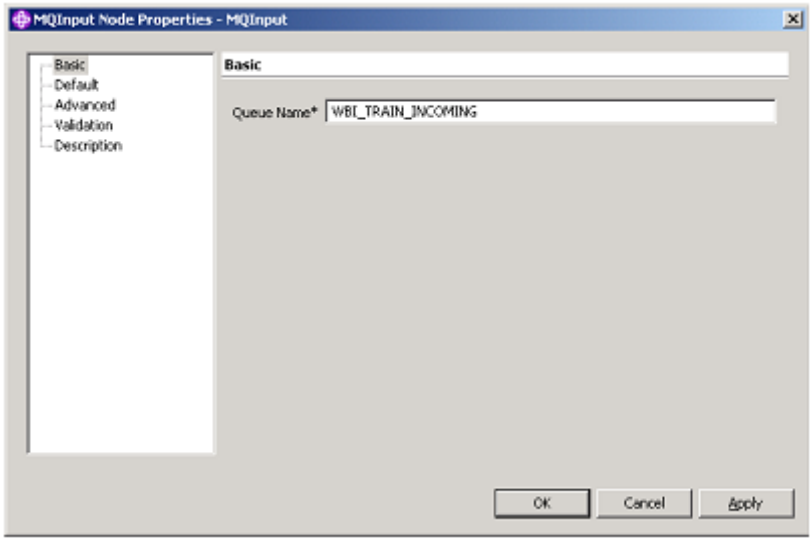

Figure 5: Selecting the queue name

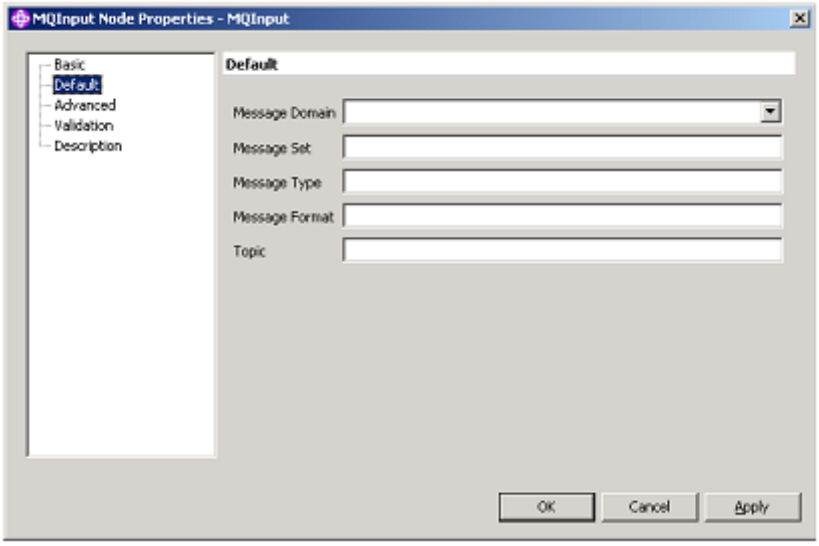

Figure 6: Selecting Default

editor back to the **Selection** function by left-clicking the menu entry – see Figure 4.

Left-click the MQInput icon and open the **Properties** dialogue. Each of the dialogue panel menu entries represents a specific function. The basic feature of a message mover is the name of the queue that incoming messages will be arriving on – see Figure 5.

The incoming messages for the MQInput function will be arriving on the WBI_TRAIN_INCOMING queue, so we add that to the **Queue Name*** input field in the dialogue panel.

If you want to change the default values that the message flow will use, highlight the **Default** menu option and make the appropriate changes as indicated – see Figure 6.

The same goes for the **Advanced** features of a message flow – see Figure 7.

The same with **Validation** – see Figure 8.

All WBI technologies from IBM provide users with the ability

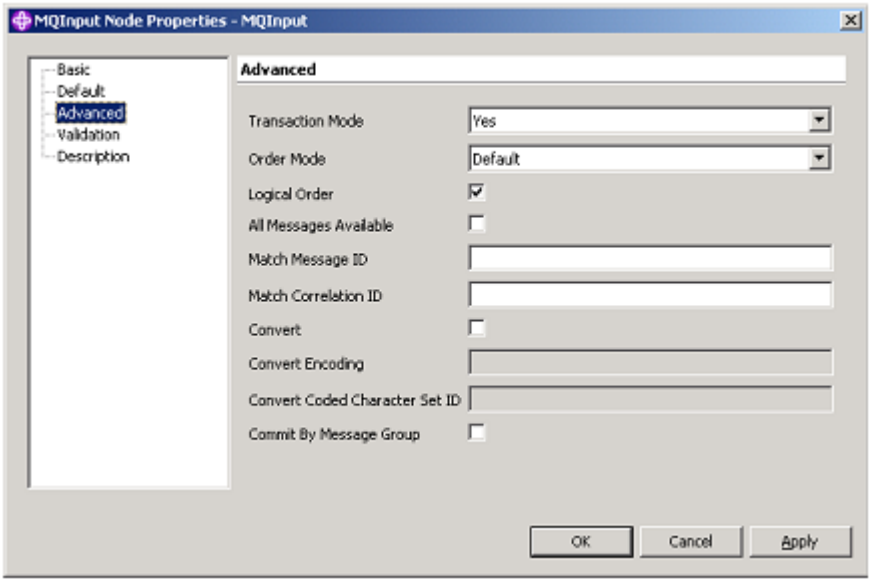

Figure 7: Advanced features

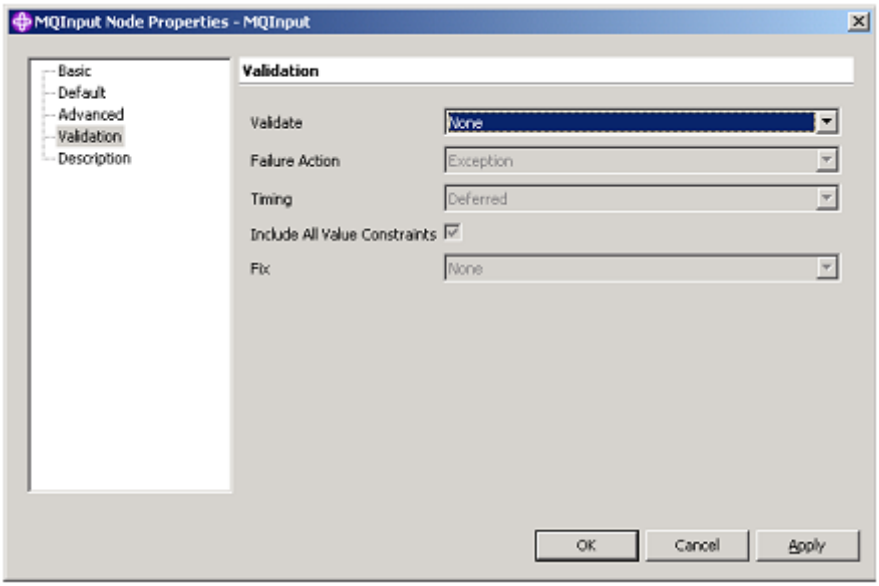

Figure 8: Validation

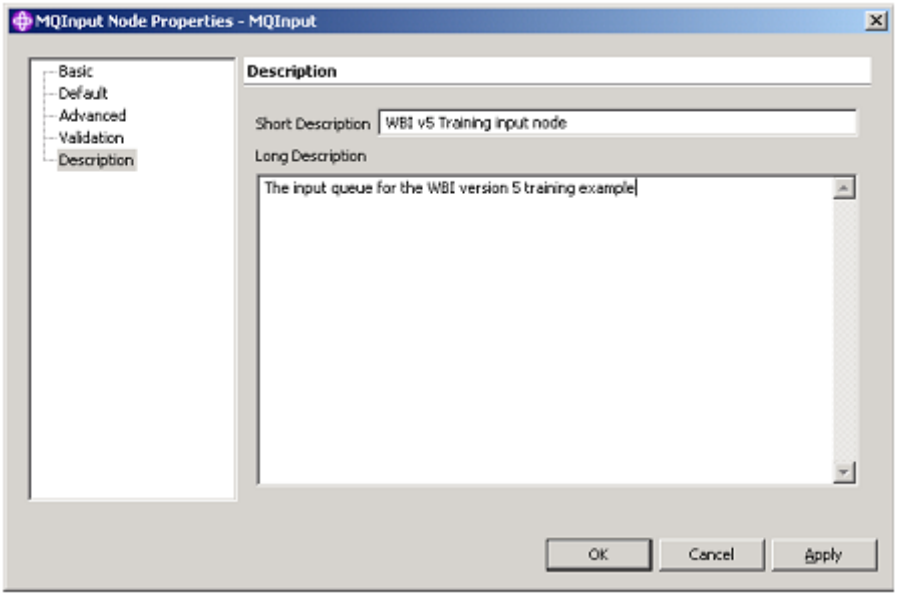

Figure 9: Adding descriptive detail

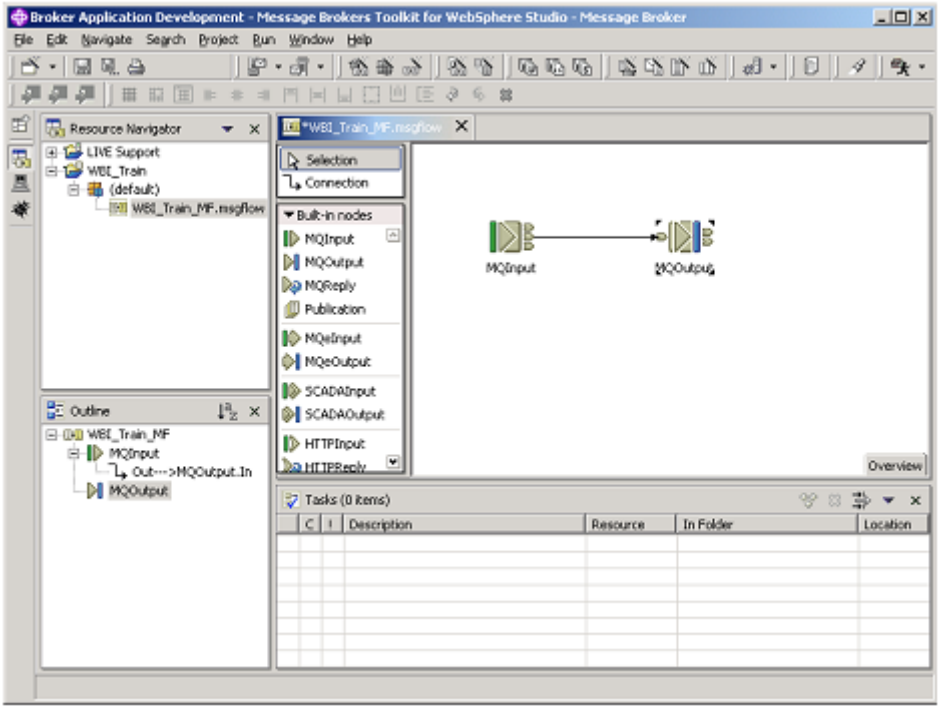

Figure 10: Finishing customization

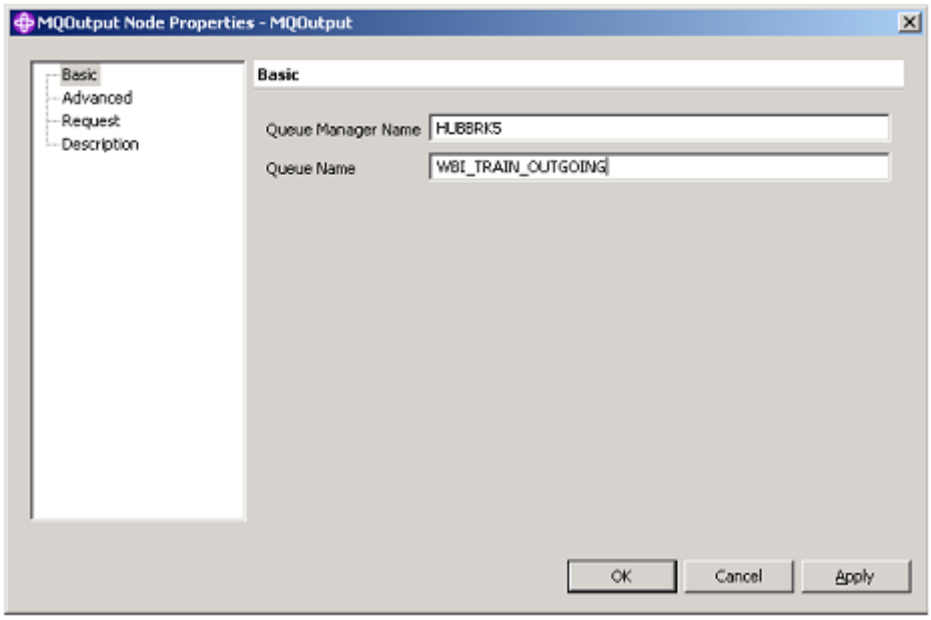

Figure 11: Customizing MQOutput node

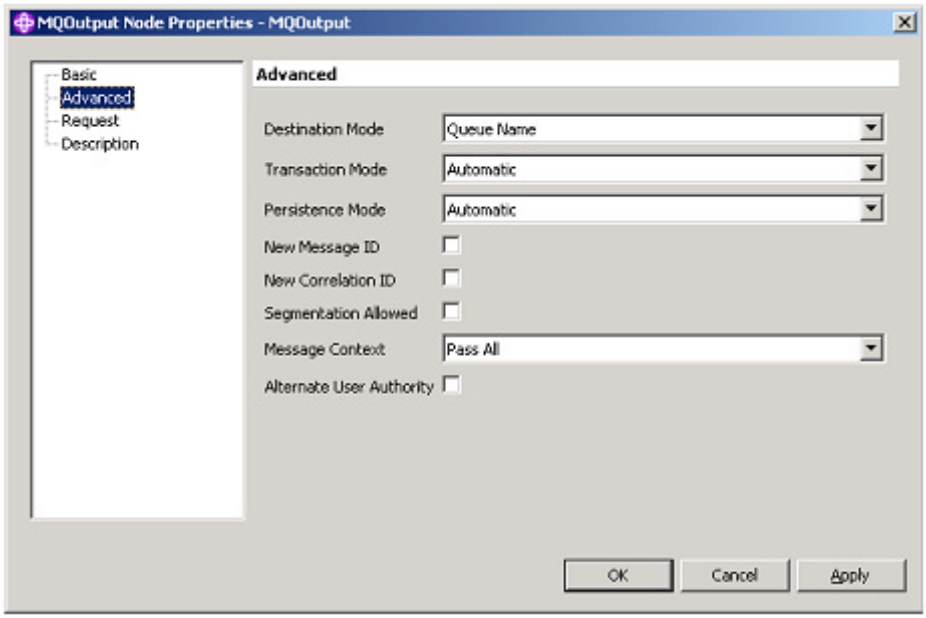

Figure 12: Customizing the Advanced option

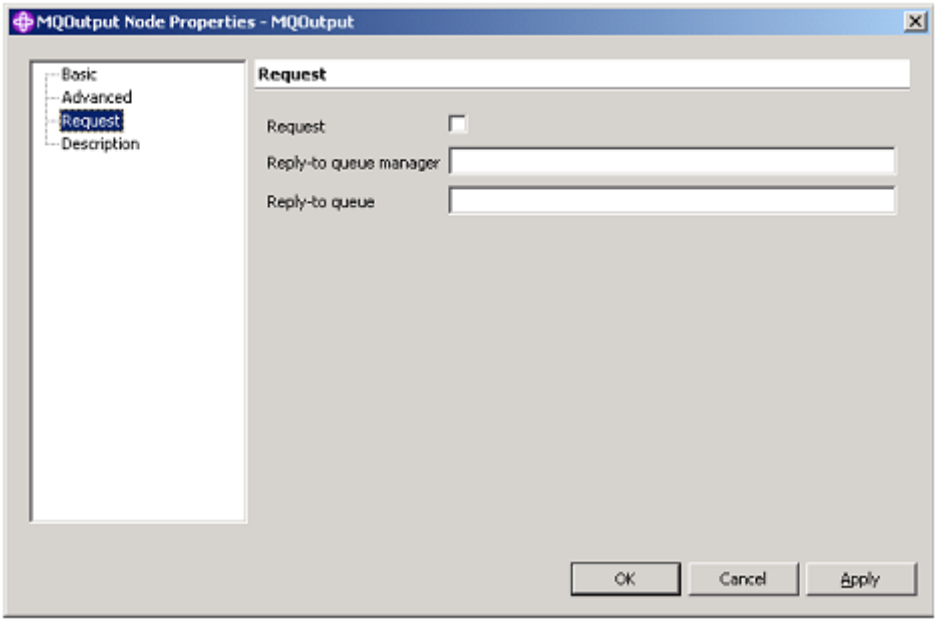

Figure 13: Customizing the Request dialog

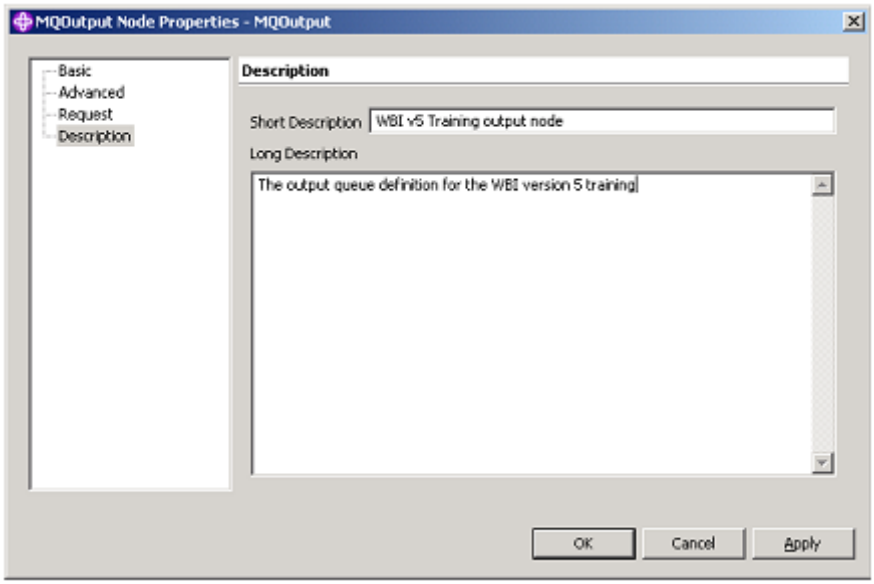

Figure 14: Customizing the Description dialog

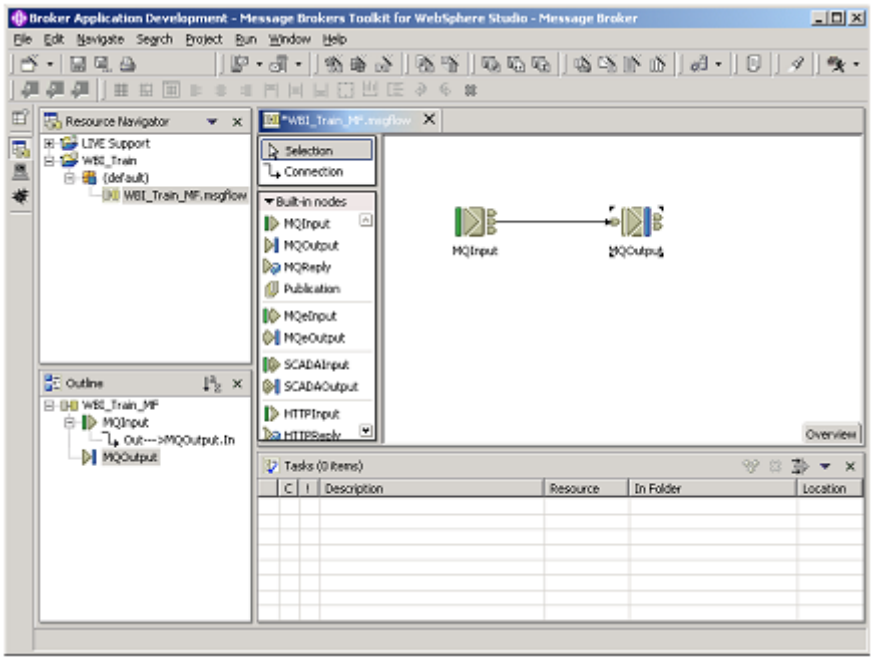

Figure 15: Saving the update

to add documentation to any object that is created. It is highly recommended that you add appropriate descriptive detail for the benefit of subsequent users – see Figure 9.

After you are finished with the details that you want to modify, press **Apply** and then **OK** to finish the customization – see Figure 10.

Now, you need to repeat the process of customizing the properties for the MQOutput node in the same fashion as above. First you will see the Basic details, including the **Queue Manager Name** and the target **Queue Name** – see Figure 11.

As before, the **Advanced** dialogue can be customized – see Figure 12.

The same applies to the **Request** dialogue – see Figure 13.

Finally the **Description** dialogue should be customized for the

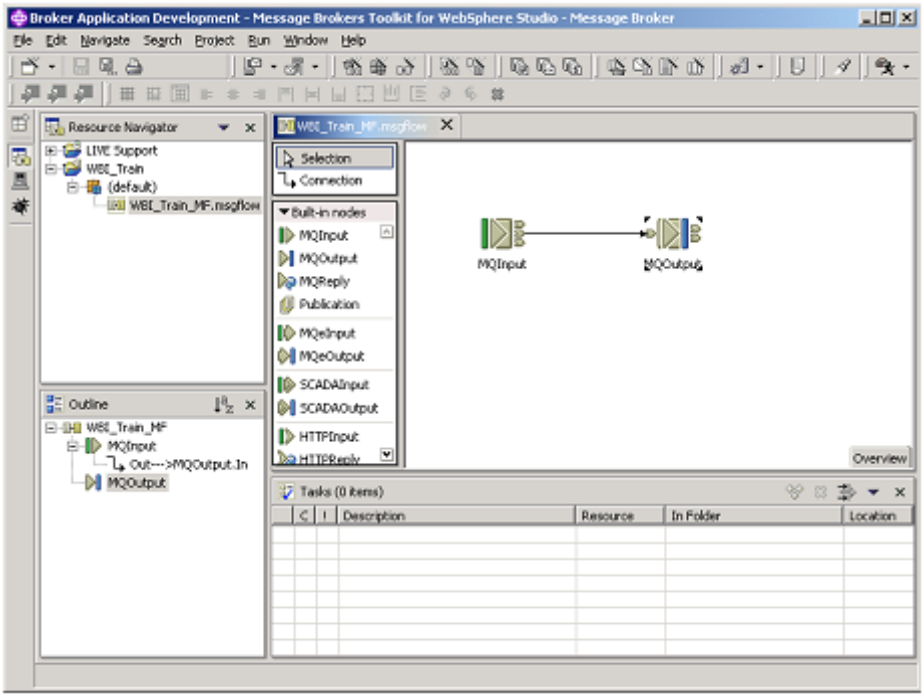

Figure 16: Saving the changes

benefit of subsequent users – see Figure 14.

As before, clicking **Apply** and **OK** will save the updates that you have made to the objects – see Figure 15.

You will notice that the WBI_Train_MF.msgflow label in the message flow editor screen has an asterisk in front of the name. This indicates that the flow has been modified and has not yet been saved. Pressing the **Save** button will flush the editor contents to disk and make sure that your changes are completely preserved – see Figure 16.

DEPLOYING THE TEST MESSAGE FLOW

In order to execute a message flow, you must have a message flow project and its subcomponent, a message flow, defined. In the outline above, a message flow project and message flow were created using the Broker ADP utility. The message flow project was stored in a file during the execution of the final

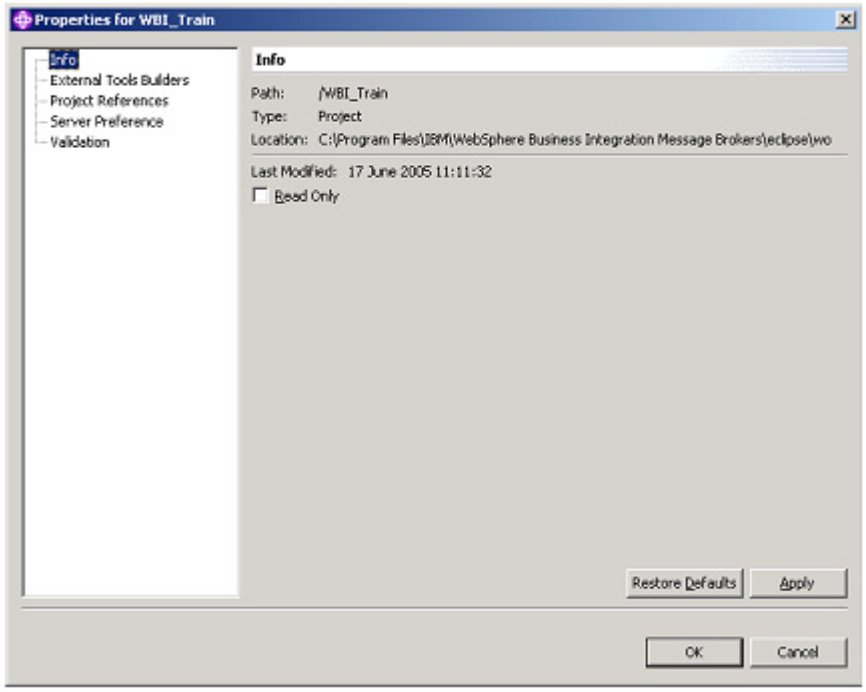

Figure 17: Destination file definition

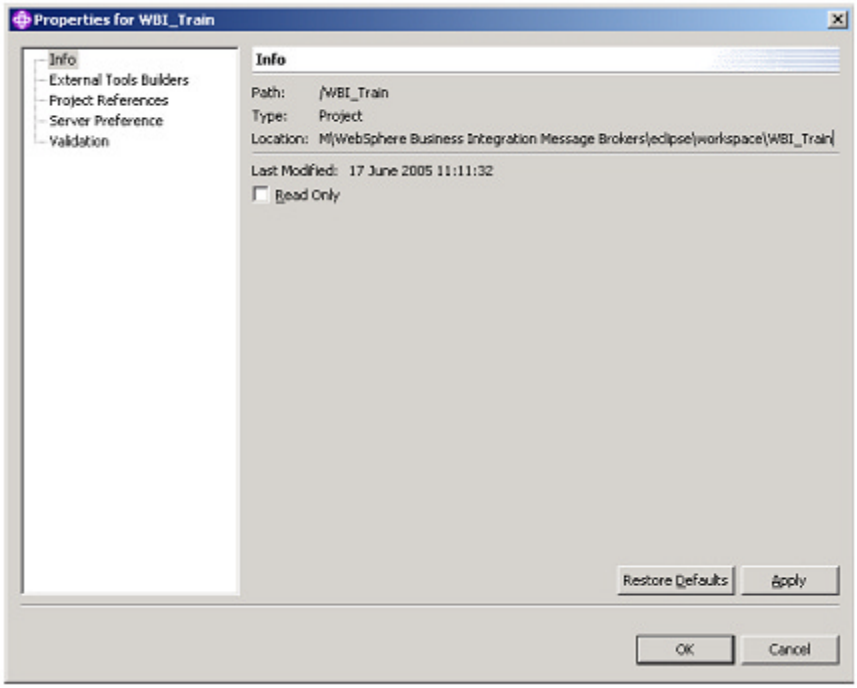

Figure 18: Destination file definition

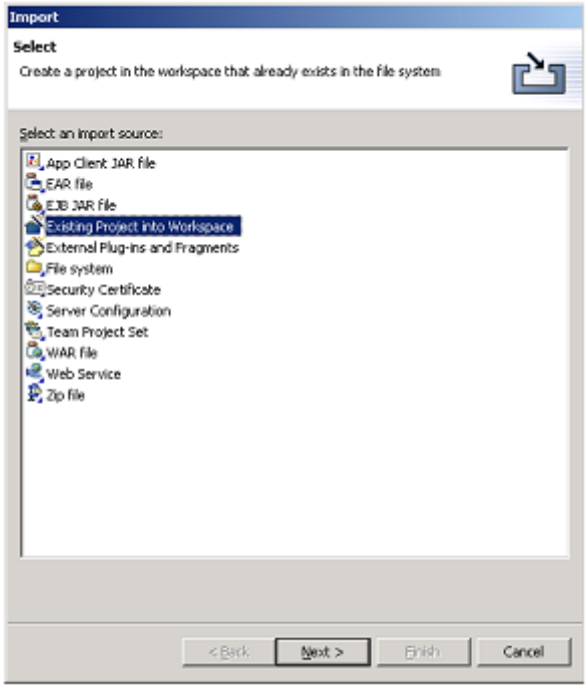

Figure 19: Dialog panel

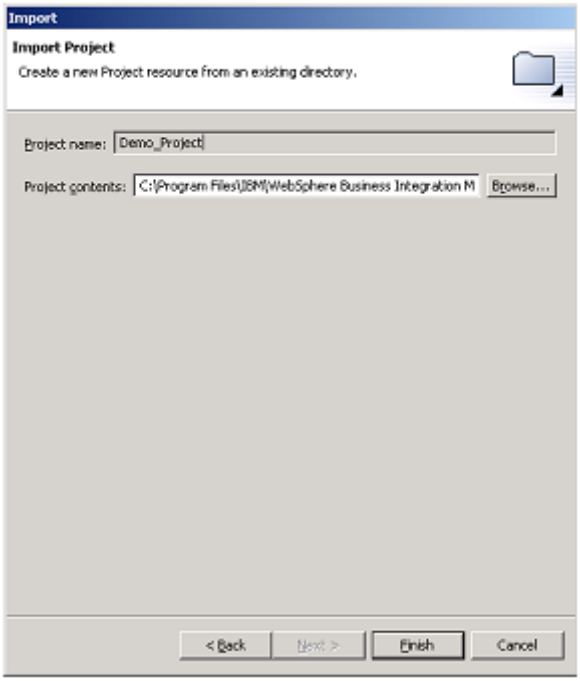

Figure 20: Existing project selection

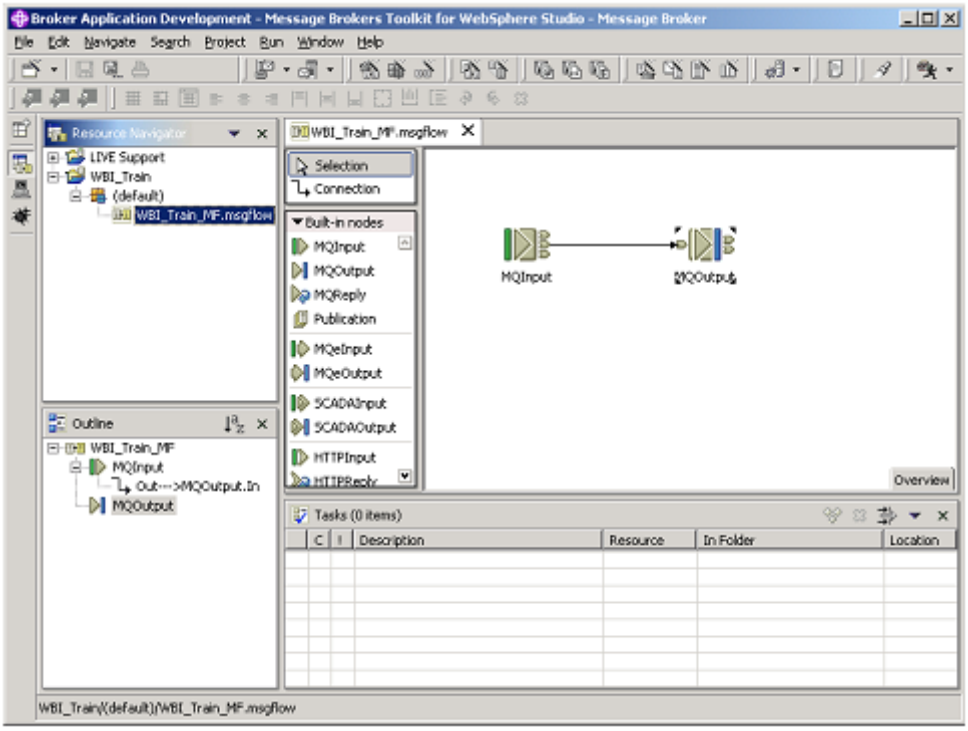

Figure 21: Highlighting Message Flow

save. In Figures 17 and 18, the complete destination file definition is shown.

At this point, I would like to point out that the alternative to creating a message flow of your own is to deploy a message flow that has been provided to you by your application or middleware development teams for execution on the Broker instance.

In the same way that you saved the message flow project above to a disk file, if you have been provided with a message flow project to deploy, you will need to import the project into your Toolkit environment in order to deploy it, by left clicking the **File/Import…** option in Message Broker Toolkit. This will start the dialog panel shown in Figure 19.

Click on the **Existing Project into Workspace** entry and press **Next** – see Figure 20.

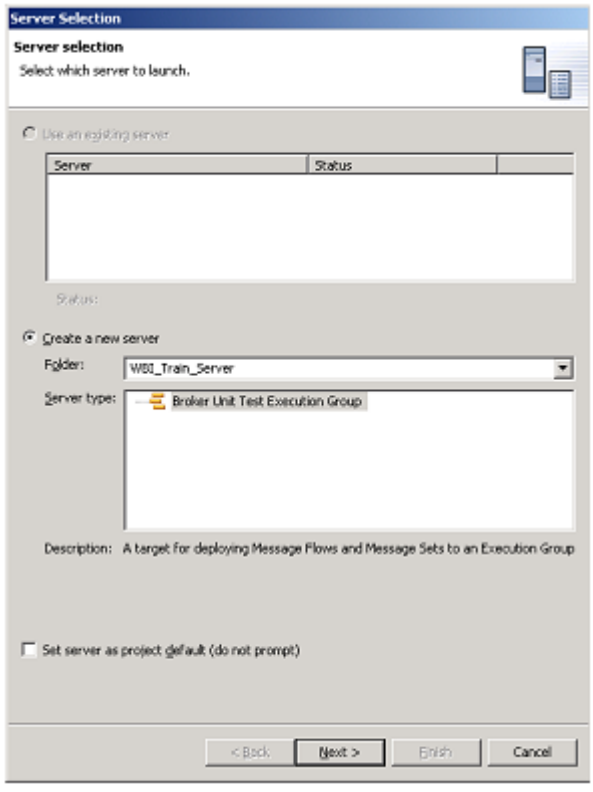

Figure 22: Run on server option

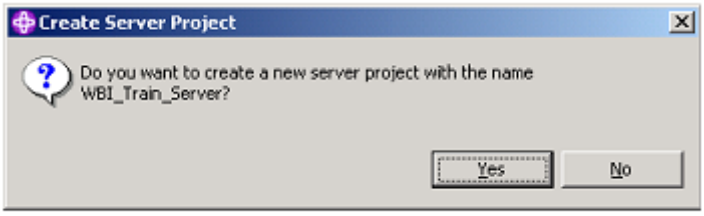

Figure 23: Create a new server

Clicking the **Browse** button will allow you to select the message flow project file from the location where you have stored it. Ensure that the project name is the same as the message flow project name, and press **Finish** to import the

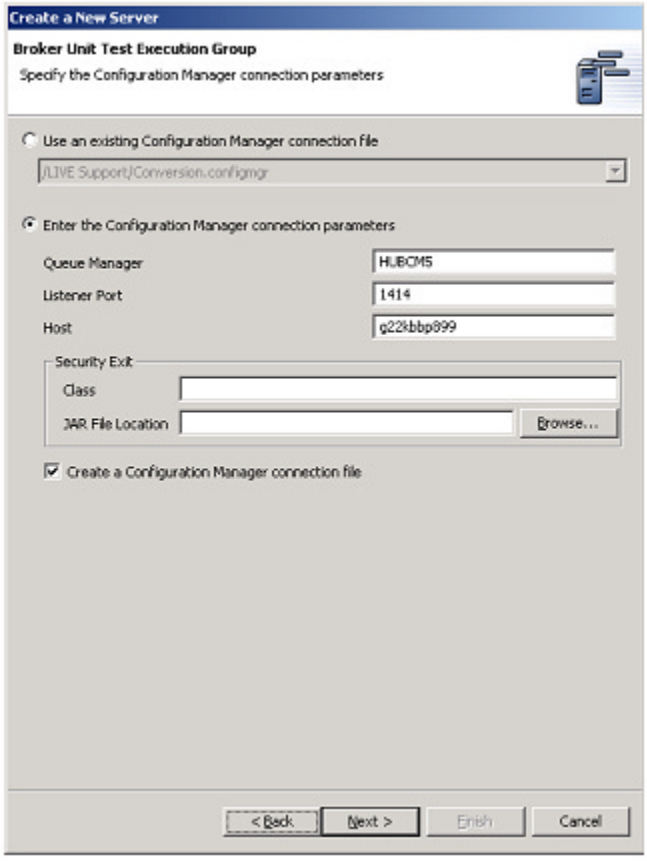

Figure 24: New server panel

contents into your **Broker Administration Navigator** pane.

Moving back to the test message flow project and its accompanying message flow, which were created above – in order to run the message flow in the WBI Message Broker instance, you must first highlight the message flow in the **Resource Navigator** dialog panel – see Figure 21.

Right-click the message flow, and select the option to **Run on Server…**, which will bring up a **Server Selection** dialog panel – see Figure 22.

At this point, since you are creating a Test message flow deployment, I would suggest that you select the **Create a new server** radio button and in the Folder entry put the name of a

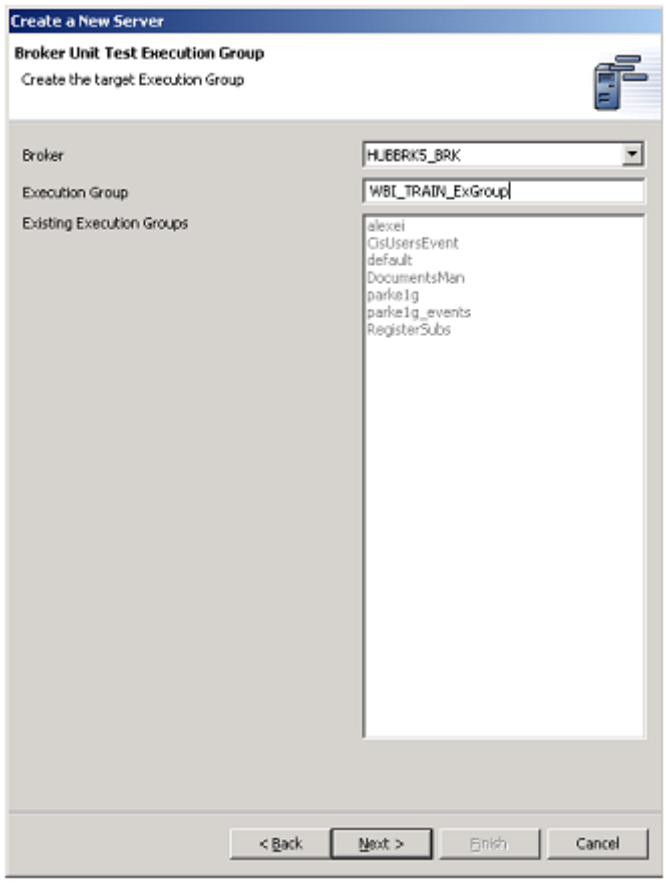

Figure 25: Modification completed

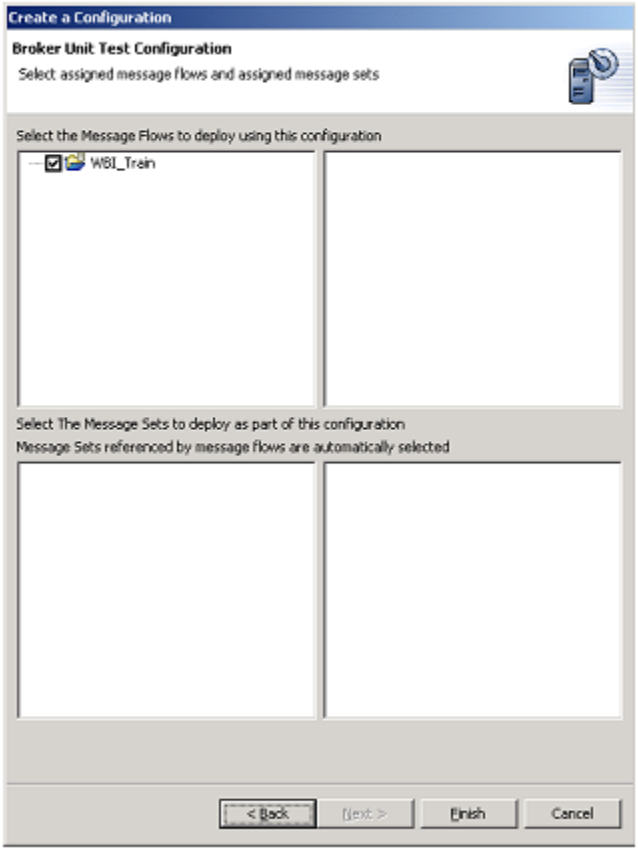

Figure 26: Execution group

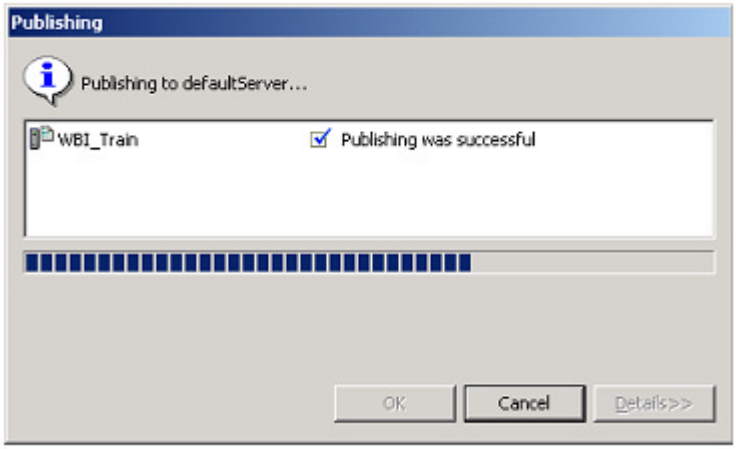

Figure 27: Processing graph

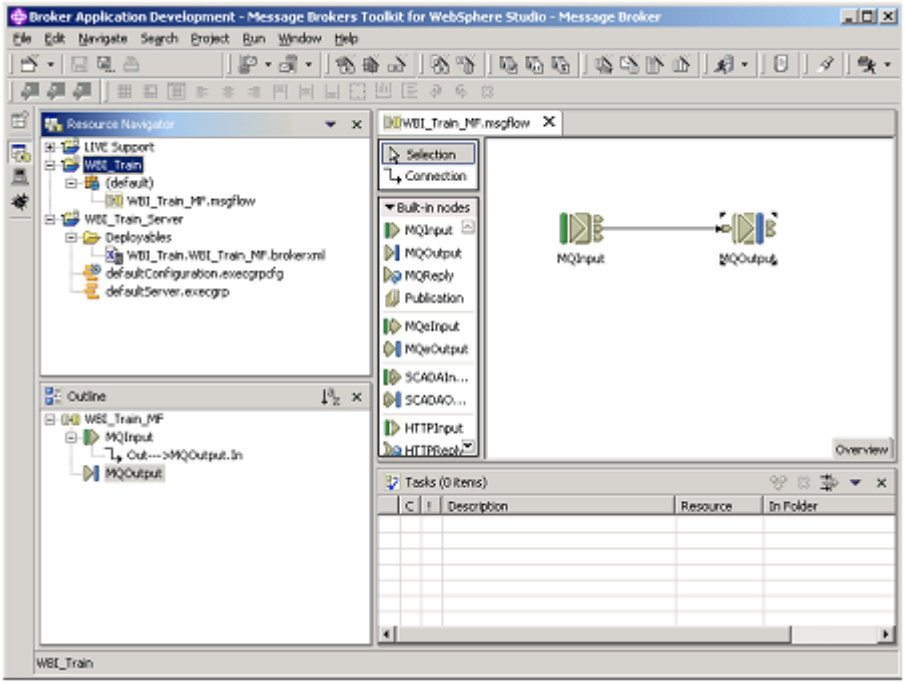

Figure 28: Server ready for testing

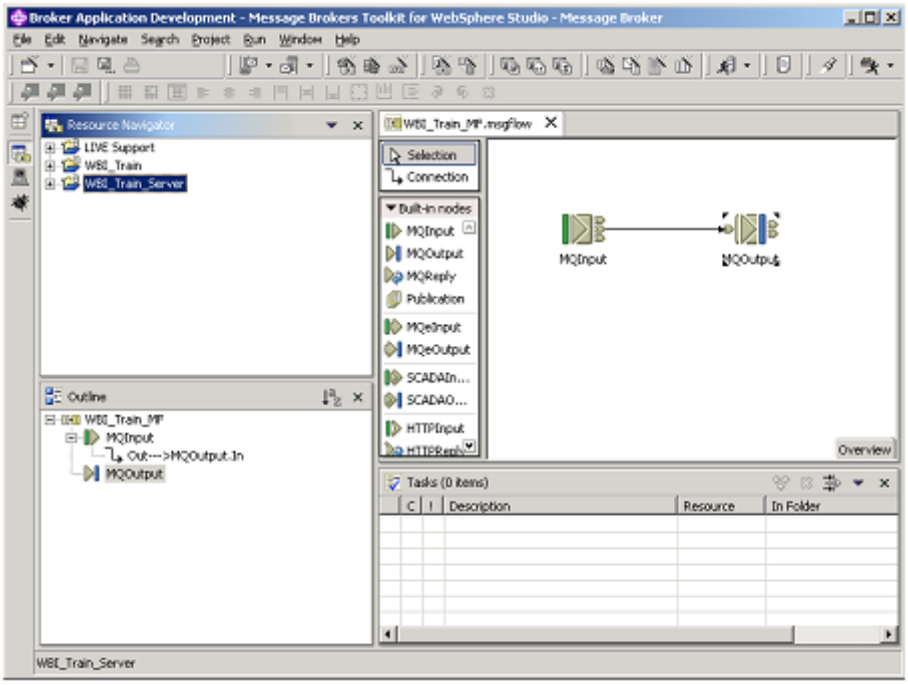

Figure 29: Highlighting WBI_Train_Server

Server group of your choice and then press **Next** – see Figure 23.

When you see the prompt press **Yes** to continue. The message flow must be defined to the Configuration Manager. The Configuration Manager validates the message flow resources and then allows you to deploy the message flow to the Message Broker.

Figure 24 shows the next screen that starts.

Fill in the configuration parameters for the WBI Configuration Manager Queue Manager, and tick the box to **Create a Configuration Manager connection file**. Once you have completed the modifications, press **Next** – see Figure 25.

You will be prompted to create an execution group in the Broker instance. Enter an appropriately descriptive name and

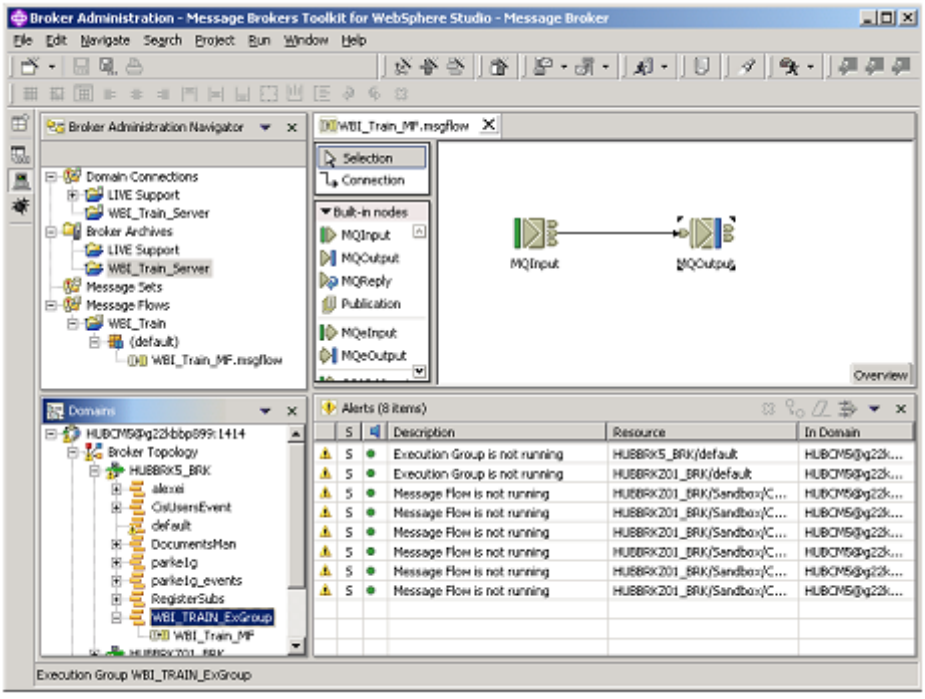

Figure 30: New components visible

press **Next** – see Figure 26.

You need to select the message flow (or flows) that are to be part of the execution group, and then press **Finish**. You will see intermediate messages come up indicating the results of processing by the Configuration Manager – see Figure 27.

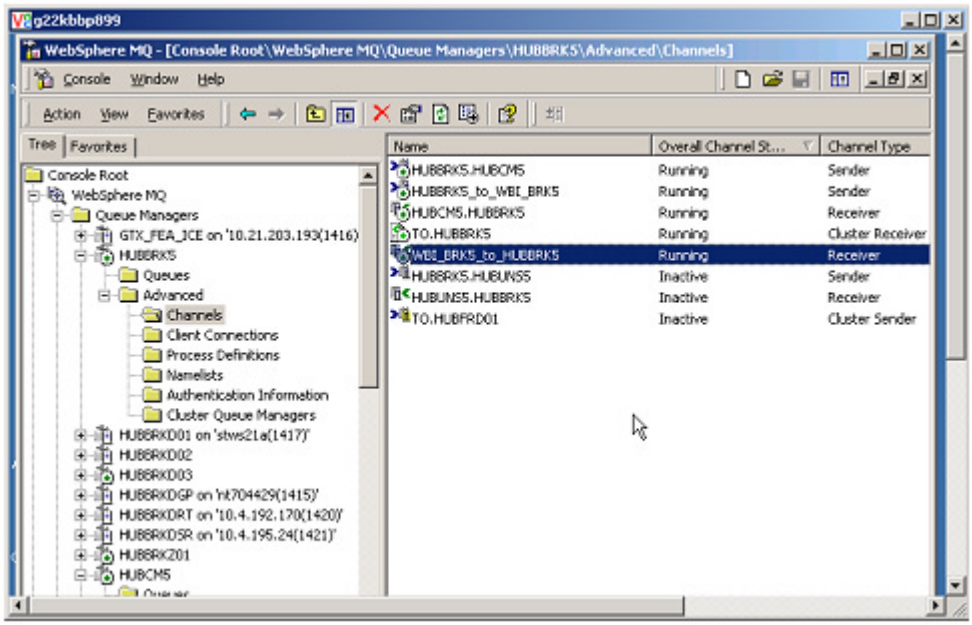

Figure 31: Ensuring that queue managers are connected

When it is finished, you will see the server deployed and ready for testing – see Figure 28.

Next, highlight the WBI_Train_Server that you have created – see Figure 29.

Right-click the server and select the drop down menu option to **Run Validation**. You will see a series of intermediate dialogue panels indicating the status of the Validation process.

By switching back to the **Broker Administration** dialogue, you will see the new components that you have added in the system – see Figure 30.

TESTING A MESSAGE FLOW

You are now ready to test your message flow. If you are using remote queue manager connections to move messages into

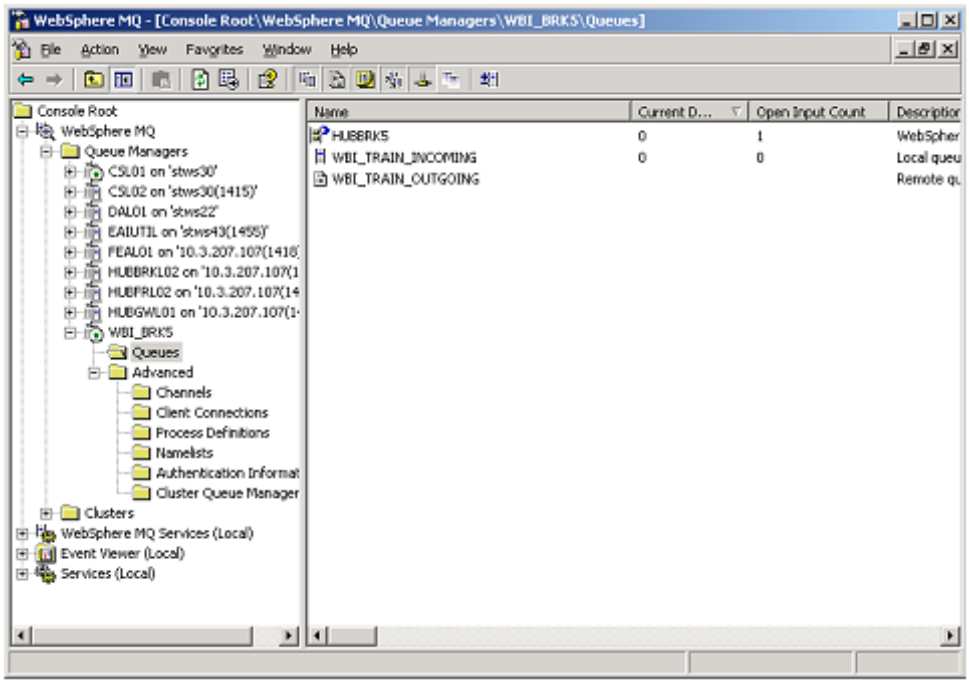

Figure 32: Linked queues

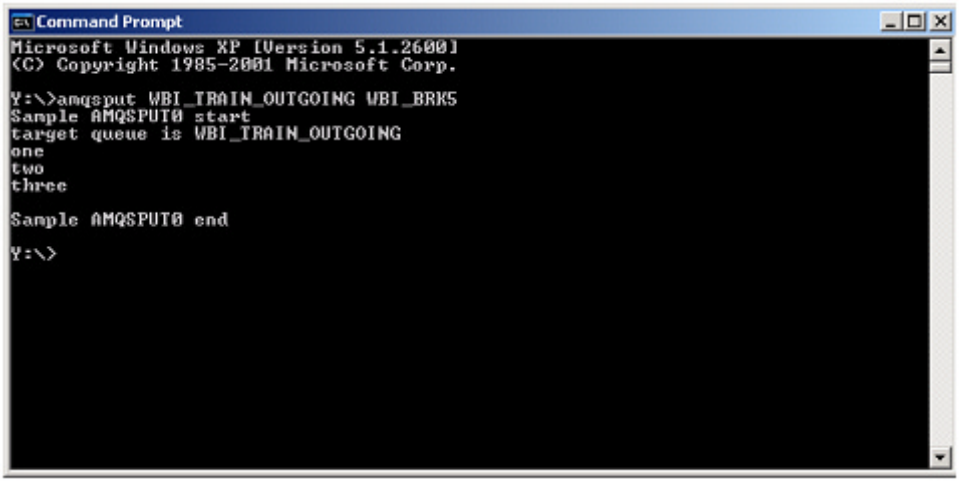

Figure 33: Forwarding messages

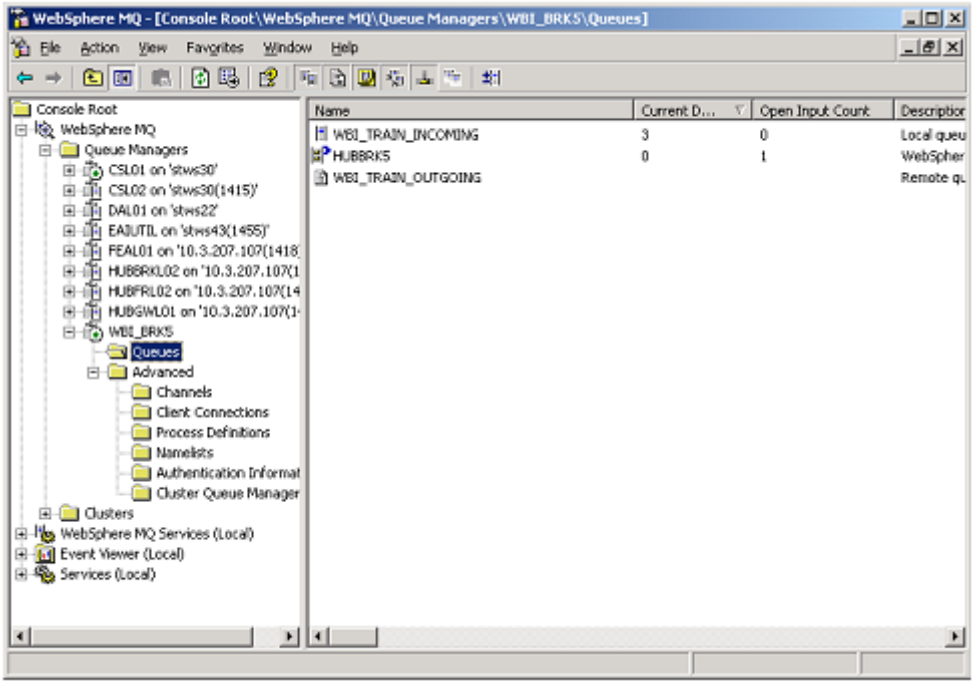

Figure 34: Messages routed back correctly

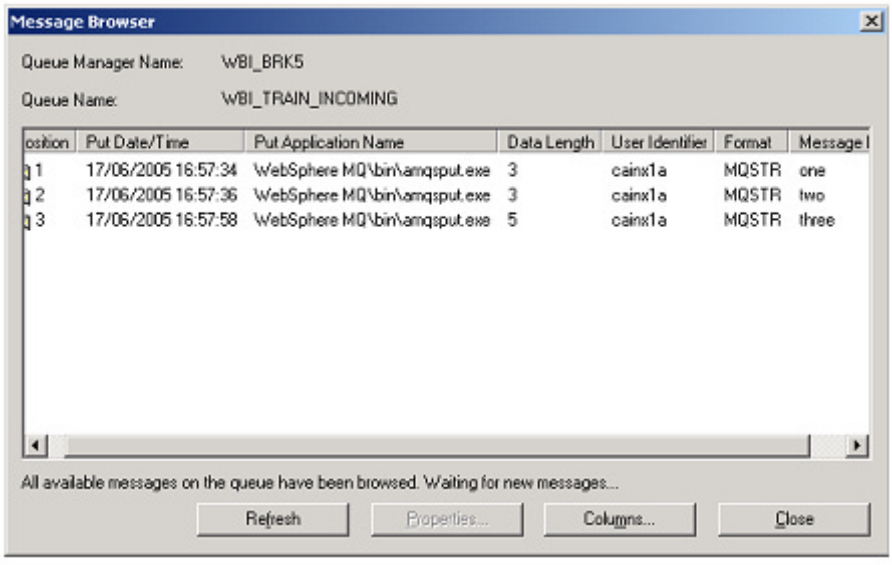

Figure 35: Correct messages on local queue

and out of the WBI Message Broker queue manager environment, make sure that your queue managers are connected – see Figure 31.

In Figure 31, both the Sender and Receiver channels between HUBBRK5 and WBI_BRK5 are connected and running. You will also need to ensure that the WMQ Queues are defined correctly on the remote queue manager for message movement. In the example below, the WBI_TRAIN_INCOMING queue is local and the WBI_TRAIN_OUTGOING queue is a remote queue pointing to the remote WBI Message Broker queue manager – see Figure 32.

You are now ready to test the new message flow. To do this, put one or more messages on the WBI_TRAIN_OUTGOING queue (if you are using a remote definition). This should send the messages to the WBI Message Broker queue manager WBI_TRAIN_INCOMING. If you are not using a remote queue manager, put the messages directly onto the local queue WBI_TRAIN_INCOMING on the WBI Message Broker queue manager. In Figure 33, we are using the remote queue manager to forward messages to the WBI Message Broker queue manager queue.

When the utility puts the three messages above onto the remote queue definition, the messages are transmitted to the Broker-based queue manager. Upon arrival on the WBI_TRAIN_INCOMING queue on the Broker queue manager, the message flow that we have defined picks up the messages and moves them onto the Broker queue manager queue WBI_TRAIN_OUTGOING, which is the WBI Message Broker remote queue definition back to our local queue manager.

If you check the WBI_TRAIN_INCOMING queue on the remote queue manager, you will find the messages have been routed back to you by the Message Broker – see Figure 34.

Browsing the local queue, we find the messages as we created them – see Figure 35.

CONCLUSIONS

When you have defined a new WBI Message Broker instance and are ready to deploy application message flow projects for your user community, it is a good idea to have at least deployed a test message flow to the instance. This ensures that any subsequent problems are not with the infrastructure itself.

With an environment that can be as complex as WebSphere Business Integration Message Broker, eliminating as many possible sources of an error is not only sensible, it is critical to your efforts to maintain a stable service to your end user community. Having a standard message flow project that you can deploy to new or existing servers is just one tool that will help you validate any WBIMB environment.

Aaron Cain
Independent Consultant
3-INGs Limited (UK)

Why not share your expertise and earn money at the same time? *MQUpdate* is looking for shell scripts, program code, JavaScript, etc, that experienced users of WebSphere MQ have written to make their life, or the lives of other users, easier. Articles can be of any length and should be e-mailed to the editor, Trevor Eddolls, at trevore@xephon.com.

Cape Clear Software has announced ESB for WebSphere, a WebSphere-specific, single-solution Enterprise Service Bus (ESB) that provides a framework for integrating internal applications and data and linking with partners using Service-Oriented Architecture (SOA) and Web services.

ESB is fully integrated and optimized for WebSphere, DB2, the WebSphere Studio family of application development tools, and the Tivoli product suite on the iSeries and other IBM platforms, the company claims. The product is designed to reduce the time, cost, and complexity of building a SOA for an enterprise's WebSphere-related applications.

For further information contact:
URL: www.capeclear.com/news/archives/2005/09/cape_clear_rele_3.shtml.

* * *

IBM has announced a new Tivoli branded Composite Application Management (ITCAM) family of solutions. These are based on its Cyanea acquisition, which had a tool for monitoring composite applications.

The ITCAM suite will mediate and monitor Web services, response time tracking, and problem tracking for WebSphere Application Server and Business Integration, CICS transaction management, MQ messaging, and IMS databases. It integrates data from a number of IBM acquisitions, including Cyanea, Candle, and Rational.

The products will track performance and response of composite applications. The first one is the Tivoli Enterprise Portal, which provides different views (workspaces) for system administrators, database administrators, and application development teams. Using Tivoli's Change Configuration Management (CCM) database as the primary source, the portals can present information from the Omegamon monitoring tools for WebSphere and MQSeries, Rational code profiling, and Tivoli event consoles. The portal provides the front end for all the data delivered by the ITCAM products.

ITCAM for WebSphere Business Integration uses the existing capabilities of the Omegamon tools to drill down on problem 'channels' in MQ. Users can view MQ message object definitions prior to deployment and at run time, regardless of whether the MQ server is based on mainframes or distributed platforms.

For further information contact:
URL: www-306.ibm.com/software/tivoli/services/ consulting/offerings/offers-composite-app-mgmt.html.

* * *

Oracle and IBM have announced a partnership to ensure that Oracle's packaged applications run natively on the majority of IBM's WebSphere-branded middleware, including its application server and portal, plus IBM's recently announced Process Server.

The plan is that there will be compatibility between WebSphere and Oracle's Project Fusion middleware.

IBM and Oracle expect to enable existing Oracle applications (Oracle JD Edwards, Oracle PeopleSoft Enterprise, and Oracle E-Business Suite) to support WebSphere and Tivoli in the areas of identity management, single sign-on, and directories.

For further information contact:
URL: www.oracle.com.