MQ update

56

February 2004

In this issue

MQ Update

Published by

Xephon Inc
PO Box 550547
Dallas, Texas 75355
USA

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

Editor

Madeleine Hudson
E-mail: MadeleineH@xephon.com

Publisher

Nicole Thomas
E-mail: nicole@xephon.com

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 1999 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.

Is my queue manager running?

"Is my queue manager running?" is a simple question but an important one nevertheless, because it forms the basis of any monitoring or health-checking tool.

While the question may be simple the answer is not so straightforward because there are several possibilities. This article explores the logic used when I worked on the development of the High Availability SupportPacs (eg MC63) for WebSphere MQ (WMQ) on distributed platforms such as Unix and Windows.

A check to see whether or not a queue manager is available needs to be an efficient and reliable process. Most health checks are periodic; for timely notification that the queue manager is down, polling needs to be reasonably frequent, eg every 30 seconds. A longer polling period will delay any recovery processing, such as failing over the queue manager to a standby machine. The test must be efficient and not consume excessive resources because it is executed regularly. It must also be accurate – an answer such as "the queue manager is probably running" does not help with availability. The test should also work with different product versions.

LOOKING FOR A PROCESS

Queue managers on distributed platforms appear as a number of cooperating processes. When you start a queue manager (**strmqm**) the command causes a new process to begin (amqzxma0), which in turn starts a number of other processes, such as those needed to control the transaction logs (eg amqhasmx, amqharmx). If any one process can be said to be 'the queue manager' it is amqzxma0, known as the Execution Controller (EC). This process initiates and monitors the various child processes.

One of the parameters of the EC is the queue manager name so that it can be spotted easily in the list of operating system processes. So, can we use the existence of this process as an indicator of a queue manager's operation? The answer is no. Testing for the existence of the process, using the following command, is efficient, but the test itself does not necessarily give a true result.

```
     ps -ef | grep amqzxmaØ | grep QMgr
```

Perhaps the queue manager is starting or stopping, which will show the EC process, but it is not available to applications. There may also have been times when processes have hung or become blocked; this leaves the EC visible but doing no useful work.

THE *QMSTATUS.INI* FILE

Everyone should know about the *qm.ini* file, which holds some of the configuration data for a queue manager. Not as many people know about the *qmstatus.ini* file because there is nothing in there that users can or should directly manipulate. This file contains data that is convenient, but not essential, for the queue manager to maintain across restarts. For example, it gives hints about how much memory was being used last time. This is useful because we do not have to waste time dynamically extending resources from 'default' values; on the assumption that this time the queue manager will handle a similar workload, the previous resource allocation is used directly.

On Windows, *ini* files are not used. Instead, they are located in the registry. The equivalent *qmstatus.ini* information can be found in HKEY_LOCAL_MACHINE\SOFTWARE\IBM\MQSeries\CurrentVersion\Status\QueueManager\<Qmgr>.

One of the stanzas in this file shows the queue manager status. With values including 'Running', 'Ended', and 'EndedImmediately', this could be used to test the queue manager. Unfortunately, once again, this would not be good enough. If the queue manager is killed directly (eg 'kill -9') there is no opportunity for the *ini* file to be updated and so it cannot be guaranteed to reflect reality. This is one of the indicators that is probably, but not necessarily, correct – not something we can rely on for a health check.

THE **DSPMQ** COMMAND

The **dspmq** command was introduced in MQSeries V5.2 as a way to list the queue managers that have been configured on the machine. It also gives the current state of those queue managers. In WMQ V5.3 the command was substantially enhanced to show more of the possible queue manager states. The states that it currently recognizes and reports on are:

- Running.
- Ended normally.
- Ended immediately.
- Ended pre-emptively.
- Ended unexpectedly.
- Starting.
- Quiescing.
- Ending immediately.
- Ending pre-emptively.
- Being deleted.
- Not available.

The **dspmq** command is a good way to find the current status. It is a reasonably reliable check on behaviour. It is a fast and efficient program to run and there are enough checks being done inside it that it will reflect reality even when the queue manager has been forced down. (At least it will accurately show whether there is a running system – the various flavours of 'ending' and 'ended' might not be truly distinguished but that level of knowledge is not needed for a health monitor.) When we developed the HA SupportPacs this command was not available and so we couldn't use it. If you want to check the behaviour on anything other than a WMQ V5.3 system you cannot use **dspmq**.

The output from the command is dependent on the environment. For a reliable script you should always force a known language environment. This is necessary so you can parse the responses, knowing which language the strings are displayed in:

```
     LANG=C
     export LANG
     dspmq -m <qmgr> |\
        grep -e "Running" -e "Starting" >/dev/null 2>&1
     if [ $? -eq Ø ]
     then
  …       qmgr is running or on its way up
     else
        … anything else is NOT available
     fi
```

USING **RUNMQSC**

The **runmqsc** command **PING QMGR** tests whether the queue manager is running. This turned out to be the best way of checking availability. Not only does the **PING** have to succeed but starting a new instance of **runmqsc** for each poll on the health of the queue manager means that the MQCONN must also succeed.

Because MQCONN is executed on each test the check works even during a queue manager restart – the MQCONN is simply blocked and delayed while recovery takes place; it does not fail. The test might, therefore, take a little longer during restart but it does not falsely report a queue manager failure (which could in turn lead to an unnecessary failover and recovery cycle). This operation works with every version of MQSeries and WMQ and was our chosen mechanism.

```
     echo "PING QMGR" | runmqsc <qmgr>
```

Testing the return code from this operation will show whether the queue manager is alive.

When issuing a command such as this it is a good idea to put a timeout around it so that a failure can be returned in the case of the command hanging completely. The timeout should be long enough to cope with a queue manager going through restart and recovery but not so long that it inhibits the failover. Some of the HA products will have a timeout of their own; if the health check does not return in time they will assume a failure.

I have seen versions of this where **runmqsc** is left permanently running and the **PING** command is regularly issued to it. This is not a good test because it does not get the MQCONN process exercised during each poll. A **PING** shows not only that the queue manager processes are alive but also that the queue manager is processing real work – the only true test of availability.

TESTING CHANNELS AND ROUNDTRIPS

More sophisticated health checks can be devised. A common one is to test that two queue managers and their connecting channels are available. This can be done quite simply by the first queue manager having a remote queue definition that points to the partner machine, where the destination is in turn another remote definition pointing back to the original machine. This can make it easy to measure not just queue manager availability but also response time across the network. The polling application puts a message to the remote queue and then waits for the same message to appear on its reply queue. Because all the routing is handled by the queue managers there is no need to have a servicing application on the partner machine.

There are some drawbacks to this kind of testing. One is scalability – you will perhaps need many pair-wise definitions to test the health of all the intercommunicating queue managers and channels. But more importantly, these tests do not give any clue as to which component has failed. Not receiving a response message could indicate a local channel problem, a network failure, or a failure of the partner queue manager. This does not help with the decision of whether or not to failover a queue manager to a standby machine and is something that is better left to more complex system management tools.

Measuring response times and testing the availability of roundtrip configurations are important, but for the original question of whether the local queue manager is running this is an area that should be avoided.

SUMMARY

In this article I have explained why looking for processes is not a sufficient check for queue manager availability. Instead, use **PING QMGR** as a single-shot command into **runmqsc**. This is an efficient and accurate test that works with all versions of WMQ. If you are dealing only with WMQ V5.3 systems, **dspmq** is an alternative approach.

Mark E Taylor
IBM Hursley (UK)

Creating a channel-based IP wrapper

A WebSphere MQ (WMQ) queue manager that has been set up using a default configuration runs with a very weak security profile. One way of improving the security is to install and configure a TCP wrapper such as *tcpd*. This wrapper checks the address of a requesting queue manager or WMQ client for each incoming channel request. *tcpd* compares this address with the contents of two lists of allowed and denied addresses. If allowed, the wrapper starts the program amqcrsta as the receiving end of the channel. The address check of *tcpd* is independent from a channel name or type.

This article explains how to create a channel-based IP wrapper. This wrapper works in a similar way to *tcpd* but is developed as a security exit. The advantage of this solution is that permissions are validated that are specific to a channel. In contrast to some other security exit solutions this validation happens only on the receiving side of a channel. No communication with the sending end (ie user or password exchange) is necessary.

HOW *TCPD* WORKS

The TCP wrapper *tcpd* (which is available free of charge) controls the startup of applications through the demon inetd. It uses two control files, */etc/hosts.allow* and */etc/hosts.deny*, to enable or disable specific IP addresses or host names to run an application. Add the following entries to the files */etc/services* and */etc/inetd.conf* to start a WMQ channel with inetd on AIX (on Sun Solaris and other Unix systems replace usr with opt):

- /etc/services:

```
WMQport    1414/tcp       # Channel listener for queue manager QM1
```

- /etc/inetd.conf:

```
WMQport stream TCP nowait mqm /usr/mqm/bin/amqcrsta amqcrsta -m
QM1
```

To set up the wrapper for WMQ, assuming *tcpd* is installed in */usr/local/bin*, the file */etc/inetd.conf* has to be modified as described below:

- /etc/inetd.conf with tcpd:

```
WMQport stream TCP nowait mqm /usr/local/bin/tcpd /usr/mqm/bin/
amqcrsta -m QM1
```

Now reconfigure the inetd by executing the following command:

- On AIX:

```
refresh -s inetd
```

- On Sun Solaris and other Unix systems:

```
kill -HUP $(ps -ef | grep inetd | grep -v grep | awk '{print $2}')
```

To use *tcpd* with WMQ the binary */usr/local/bin/tcpd* must be executable and the control files */etc/hosts.allow* and */etc/hosts.deny* have to be readable by the group mqm.

LIMITATIONS OF *TCPD*

The wrapper *tcpd* works very well in many situations; nevertheless, there are some limitations, which may restrict or prevent its use:

- You have to install additional software (the TCP wrapper).
- Root privileges are required to set up or modify the *tcpd* configuration.

- It is not possible to set up different permissions on different channels.
- This solution is available only on systems using inetd. It does not work with the listener **runmqlsr**.

THE CHANNEL-BASED IP WRAPPER

Purpose of a channel-based IP wrapper

In large WMQ networks the administration of queue manager configurations can be performed using graphical interfaces such as MQExplorer or MQMON (IBM SupportPac MO71). These solutions require WMQ administrator local user accounts on the machines running a WMQ queue manager. Additionally, MQExplorer requires an administration channel called SYSTEM.ADMIN.SVRCONN. If WMQ administrator local user accounts are not available, or they do not belong to the group mqm, it may be necessary to alter the attribute MCAUSER to mqm. Unfortunately this would open the channel for every user and provide them with WMQ administrator privileges.

Free Java tools, which are available on the Web, provide users with full administration rights, even without a local user-ID or MCAUSER set to mqm or a channel SYSTEM.ADMIN.SVRCONN. A TCP wrapper can reject unauthorized access by filtering allowed IP addresses but, to install, configure, and administer a tool such as *tcpd*, root privileges are required. Additionally, *tcpd* checks access to the process amqcrsta**,** not to a specific channel. Another difficulty may be that, because of security concerns, administrators might want to disable inetd completely and use **runmqlsr** instead.

It would be much easier to have a solution that could be set up by a WMQ administrator. The solution described in this article is independent of such restrictions. It can be seen as a part of the WMQ configuration and carried out by WMQ administrators.

How the channel-based IP wrapper works

The solution described is developed as a security exit. As with *tcpd*, this exit checks the requesting IP address or host name and compares it with a list of allowed addresses. In contrast to *tcpd*, no deny list is used so unknown or not allowed addresses are always denied. The lists of allowed IP addresses are set up as namelists. The namelist, which is used for a channel, is defined as security exit data. This mechanism allows individual address lists to be set up for each channel. It is also possible to configure a couple of channels (eg an online and a batch channel, a converting and a non-converting channel) in the same way by using the same namelist.

If an address is not found in the 'allow-list' the request will be revoked. This also happens when no namelist is configured or the namelist does not exist. Revoked channel requests will be reported in the log file. In report-only mode every request is reported but nothing will be really revoked. This mode is useful to test the configuration of the list simply to report channel requests and start-ups. If a log entry cannot be written (e g because the disk is full) the functionality of the wrapper is not affected.

Building the library

The following script sample creates the binary – on non-DCE platforms – from the file *wrapper.c* for AIX and Sun Solaris systems. I assume the GNU compiler gcc is installed in */usr/local/bin*.

```
os=$(uname -s)
file=wrapper
case $os in
 AIX)
   CC="/usr/local/bin/gcc"
   LD="/usr/bin/ld"
   CC_FLAGS="-c -I/usr/mqm/inc"
   LD_FLAGS="-bE:$file.exp -H512 -T512 -e MQStart -bM:SRE"
   LD_LIBS="-lmqm_r -lpthreads_compat -lpthreads -lc_r"
   ;;
 SunOS)
   CC="/usr/local/bin/gcc"
   LD="/usr/ucb/ld"
   CC_FLAGS="-c -I/opt/mqm/inc"
   LD_FLAGS="-G"
   LD_LIBS="-lmqm -lthread -lsocket -lc -lnsl -ldl"
   ;;
```

```
     *)
       echo "Operating system not supported!"
       return 1
       ;;
   esac
   $CC $CC_FLAGS -o $file.o $file.c
   if [ $? -eq Ø ]
   then
    $LD -o $file $file.o $LD_FLAGS $LD_LIBS
     fi
```

For AIX you need the file *wrapper.exp*, which is shown here:

```
#!
checkAccess
reportAccess
  MQStart
```

How to build the software on other platforms is described in the *WebSphere MQ Application Programming Guide*. Feel free to extend or modify the script sample above.

Installation of the channel-based IP wrapper

Install the wrapper by copying the created binary wrapper to the directory */var/mqm/exits*. The program must be executable for user and group mqm. Create a directory */var/mqm/errors/exits* and make it read/write for user and group mqm. The wrapper creates its log file within this directory.

Configure channel security

First disable all default channel *SYSTEM.DEF.**and *SYSTEM.AUTO.** of type SVRCONN, RCVR, SVR, and CLUSRCVR, by setting the attribute MCAUSER to ‘nobody’. The newly created channel will then be disabled by default because it is assumed that nobody is a non-existing user or at least a user with no permission in WMQ:

```
ALTER CHANNEL(SYSTEM....) CHLTYPE(...) +
        MCAUSER('nobody')
```

To configure the channel-based IP wrapper alter the attributes SCYEXIT, SCYDATA, and MCAUSER. Two different functions are valid. Calling the function ‘reportAccess’ runs the wrapper in report-only mode:

```
...
SCYEXIT('wrapper(reportAccess)')
SCYDATA(MYCHANNEL.ALLOWED.IP)
MCAUSER('mqm')
  ...
```

In report-only mode every channel request will be logged but no further action will take place. In the log file you will see a message asking if the channel request should be accepted or revoked by the exit. The attribute SCYDATA must contain the name of a namelist. This namelist contains a number of IP addresses or host names that are allowed to start this channel.

```
NAMELIST(MYCHANNEL.ALLOWED.IP)
...
NAMES(10.10.100.1
  ,10.10.100.2
  ...)
  ...
```

A namelist can be used from more than one channel so you can easily allow the same IP addresses for more than one channel (eg batch and online channel). It is also possible to activate the wrapper on one channel (function checkAccess) and run in report-only mode on another channel(function reportAccess) even when they use the same namelist.

To activate the channel-based IP wrapper alter the exit function to checkAccess:

```
  SCYEXIT('wrapper(checkAccess)')
```

From this point every new channel request will be revoked if the requesting address is not found in the configured namelist. Only revoked requests will be reported in this mode.

SAMPLE CONFIGURATION

This example describes how to configure the administration channel SYSTEM.ADMIN.SVRCONN (eg for use with MQExplorer) and how to activate the channel-based IP wrapper. It assumes that you have disabled the default channel, as described above.

Creating an administration channel

Create the administration channel SYSTEM.ADMIN.SVRCONN (eg for use with MQExplorer) with **runmqsc**:

```
     DEFINE CHANNEL(SYSTEM.ADMIN.SVRCONN) CHLTYPE(SVRCONN)
```

The default channel has been disabled so the administration channel has now set the attribute MCAUSER to nobody and is not usable at the moment.

Set up the wrapper

The following commands configure the wrapper for the administration channel. Now create the namelist with the allowed IP addresses:

```
   DEFINE NAMELIST(ADMINCHANNEL.ALLOWED.IP) +
           NAMES('1Ø.1Ø.1ØØ.1','1Ø.1Ø.1ØØ.2')
```

Now set up the wrapper in report-only mode:

```
   ALTER CHANNEL(SYSTEM.ADMIN.SVRCONN) CHLTYPE(SVRCONN) +
      MCAUSER('mqm') +
           SCYEXIT('wrapper(reportAccess)') +
           SCYDATA(ADMINCHANNEL.ALLOWED.IP)
```

From this point every user can open the administration channel. Every channel request will be reported but nothing will be revoked. This configuration is useful on development systems where developers can modify the WMQ configuration. This mode is also of use during the implementation of live systems to identify the required IP addresses to be enabled for the channel. Therefore, run the channel in report-only mode for a while. Analyse the log file and add the requesting IP addresses to the namelist (assuming that you have – of course – double-checked the addresses!).

On live systems the administration channel should not be accessible by any user except the WMQ administrators. When the appropriate WQM administrators have local user accounts you should clear the attribute MCAUSER:

```
   ALTER CHANNEL(SYSTEM.ADMIN.SVRCONN) CHLTYPE(SVRCONN) +
           MCAUSER(' ')
```

This works well for tools such as MQExplorer or MQMON. But, as mentioned above, with freely available Java tools it is still possible to get administrative access without having a user-ID. This behaviour results from the way Java submits the client user-IDs. Activate the channel-based IP wrapper by modifying the exit function:

```
ALTER CHANNEL(SYSTEM.ADMIN.SVRCONN) CHLTYPE(SVRCONN) +
        SCYEXIT('wrapper(checkAccess)')
```

Afterwards, only channel requests from the addresses 10.10.100.1 and 10.10.100.2 are accepted. The wrapper revokes requests from any other addresses. Now set up further channel and namelists for client applications and queue manager connections.

Examples

```
DEFINE CHANNEL(APPL.SVRCONN) CHLTYPE(SVRCONN) +
   MCAUSER('appluser') +
   SCYEXIT('wrapper(reportAccess)') +
        SCYDATA(' ')

DEFINE CHANNEL(QM2.TO.QM1) CHLTYPE(RCVR) +
   MCAUSER(' ') +
   SCYEXIT('wrapper(checkAccess)') +
        SCYDATA(QM2.ALLOWED.IP)

DEFINE NAMELIST(QM2.ALLOWED.IP) +
        NAMES('1Ø.3Ø.1ØØ.1','1Ø.3Ø.1ØØ.2')
```

In the examples above, the local queue manager is named QM1. It will accept and report every channel request for WMQ clients using the channel APPL.SVRCONN. I assume the user appluser is enabled for some application queues by the command **setmqaut**. Requests to start the receiver channel QM2.TO.QM1 are accepted only from the addresses 10.30.100.1 and 10.30.100.2. Modify the namelist QM2.ALLOWED.IP to allow or deny IP addresses to start this channel.

THE CODE

The channel-based IP wrapper is developed in the form of a C library. This library is called 'Wrapper' and has two defined entry points, reportAccess and checkAccess. Both of them call the same function, compareAddress, which contains the whole wrapper logic. Another function, logTS, is used to create the message prefix for log file entries.

Exit entry points

Before the exit entries reportAccess and checkAccess can be called from WMQ they have to be defined in the attribute SCYEXIT in the form libraryName(entryPoint). The entry functions simply pass all exit arguments to a function called compareAddress. An additional argument is passed to this function, which defines the mode of the exit (SEC_REPORT_ONLY or SEC_CHECK_PERM for read-only or active mode respectively).

Exit function

The main function, compareAddress, checks the Exit-ID provided with the exit parameters. Values other than MQXT_CHANNEL_SEC_EXIT are not accepted. Those channel requests will be closed immediately. If the Exit-ID is of type MQXT_CHANNEL_SEC_EXIT, the ExitReason is checked.

The wrapper functionality is activated when the ExitReason is MQXR_INIT. If the channel runs in report-only mode or an address check has been successful, the ExitResponse will be MQXCC_OK, which means a channel start-up is allowed. Otherwise the ExitResponse is set to MQXCC_CLOSE_CHANNEL and the channel start-up is revoked.

For the ExitReasons MQXR_INIT_SEC, MQXR_SEC_MSG, and MQXR_TERM, the function always returns the ExitResponse MQXCC_OK. Other ExitReasons will set the ExitResponse to MQXCC_CLOSE_CHANNEL.

The function compareAddress is declared as static. This means it is not visible from functions outside this module. Only the exit entry points are able to call this function.

Timestamp function

The function logTS writes a message prefix to the log file entry. The format of the prefix is defined by the C function asctime and is followed by the name of the queue manager. Sample log file entries may look as follows:

```
<Wed Nov 5 Ø9:Ø8:46 2ØØ3; QM1> *** Channel APPL.SVRCONN runs in
report mode ***
<Wed Nov 5 Ø9:Ø8:46 2ØØ3; QM1> SCYDATA of channel APPL.SVRCONN
contains not a namelist
  <Wed Nov  5 Ø9:Ø8:46 2ØØ3; QM1> Access to channel " APPL.SVRCONN"
  from address "1Ø.2Ø.1ØØ.1" would be revoked by exit
```

Global parameters

Two parameters are defined as global to the module. This is to give the function logTS access to the file handle and to the name of the queue manager.

Macros

To simplify writing to the log file the macros error_log and error_exit are defined. The first parameter of these macros is a message format string. The second parameter is an integer value (a reason or return code). If this integer value is unequal to zero and the file handle is not NULL, a log entry will be written. In case of the function error_exit the channel program ends and returns this integer value.

WRAPPER.C

```
/* July 21 2ØØ3                                               */
/* Hubert Kleinmanns                                          */
/* Senior Consultant                                          */
/* Exit Purpose: This exit is intended to prohibit unauthorized    */
/* access to a channel. It checks the requesting IP address against */
/* the contents of a namelist. The name of the namelist is read    */
/* from the exit data.                                        */
/* Secure a channel:                                          */
/*  1) Set the inbound channel scyexit and scydata with:      */
/*    DEFINE CHANNEL(channel_name) CHLTYPE(channel_type) +    */
/*       SCYEXIT('wrapper(checkAccess)') +                    */
/*       SCYDATA(name_list)                                   */
/*  2) Define the name_list specified in step (1) with:       */
/*    DEFINE NAMELIST(name_list) +                            */
/*       NAMES(ip_address1[,ip_adress2[,...]])                */
/* Setup in report mode:                                      */
/*  This exit may be run in report mode, so the IP addresses,       */
/*  which try to connect to the channel, will be logged, but        */
/*  connections will not be revoked. To enable report mode          */
/*  use another entry for the exit:                                 */
/*    DEFINE CHANNEL(channel_name) CHLTYPE(channel_type) +          */
/*  --->   SCYEXIT('wrapper(reportAccess)') +   <---                */
/*       SCYDATA(name_list)                                         */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
#ifdef AIX
#include <langinfo.h>
#endif /* AIX */
#ifndef WIN32
#include <unistd.h>
#endif /* WIN32 */
#ifdef WIN32
#include <ctype.h>
#include <windows.h>
#include <winbase.h>
#endif /* WIN32 */
#include <time.h>
#include <sys/timeb.h>
/* includes for MQI */
#include <cmqc.h>
#include <cmqxc.h>
#ifdef WIN32
 #define LOG_PATH "C:\\TEMP\\"
#else /* WIN32 */
 #define LOG_PATH "/var/mqm/errors/exits/"
#endif /* WIN32 */
#define LOG_SUFFIX "_wrapper.log"
#define LOG_FILE_LENGTH 256   /* Must be more than length of LOG_PATH
and LOG_SUFFIX */
#define SEC_REPORT_ONLY 'r'
#define SEC_CHECK_PERM 'c'
/* Macros, which write out a log entry, when the second parameter is
not Ø */
#define error_log(x,y)
if(((y)!=Ø)&&(fp!=NULL)){logTS();fprintf(fp,(x),(y));}
/* ... and ends the program ... */
#define error_exit(x,y)
if(((y)!=Ø)&&(fp!=NULL)){logTS();fprintf(fp,(x),(y));exit((y));}
/* Declare function prototypes. */
void MQENTRY checkAccess (PMQVOID, PMQVOID, PMQLONG, PMQLONG, PMQVOID,
PMQLONG,
  PMQPTR);
void MQENTRY reportAccess (PMQVOID, PMQVOID, PMQLONG, PMQLONG, PMQVOID,
PMQLONG,
  PMQPTR);
static void logTS (void);
static void compareAddress (PMQVOID, PMQVOID, PMQLONG, PMQLONG,
PMQVOID, PMQLONG,
  PMQPTR, char);
/* Global parameters only visible to functions in this file.  */
static FILE *fp;
static char queueManagerName[MQ_Q_MGR_NAME_LENGTH+1];
/* Exit entry definitions                                     */
/* Dummy entry point.                                         */
void MQStart() {;}
/* Entry point for exit in active mode.                       */
void MQENTRY checkAccess (
  PMQVOID pChannelExitParms,  /* Channel exit parameter block */
  PMQVOID pChannelDefinition, /* Channel definition           */
  PMQLONG pDataLength,        /* Length of data               */
  PMQLONG pAgentBufferLength, /* Length of agent buffer       */
  PMQVOID pAgentBuffer,       /* Agent buffer                 */
  PMQLONG pExitBufferLength,  /* Length of exit buffer        */
  PMQPTR  pExitBufferAddr)    /* Address of exit buffer       */
{
  /* Run exit in active mode  */
  compareAddress (pChannelExitParms,
   pChannelDefinition,
   pDataLength,
   pAgentBufferLength,
   pAgentBuffer,
   pExitBufferLength,
   pExitBufferAddr,
   SEC_CHECK_PERM);
}
/* Entry point for exit in report-only mode.                     */
void MQENTRY reportAccess (
  PMQVOID pChannelExitParms,  /* Channel exit parameter block    */
  PMQVOID pChannelDefinition, /* Channel definition              */
  PMQLONG pDataLength,        /* Length of data                  */
  PMQLONG pAgentBufferLength, /* Length of agent buffer          */
  PMQVOID pAgentBuffer,       /* Agent buffer                    */
  PMQLONG pExitBufferLength,  /* Length of exit buffer           */
  PMQPTR  pExitBufferAddr)    /* Address of exit buffer          */
{
  /* Run exit in report-only mode */
  compareAddress (pChannelExitParms,
   pChannelDefinition,
   pDataLength,
   pAgentBufferLength,
   pAgentBuffer,
   pExitBufferLength,
   pExitBufferAddr,
   SEC_REPORT_ONLY);
```

```
}
/* Main exit function, which is used by the exit entries
'checkAccess'*/
/* and 'reportAccess'.                                          */
static void compareAddress (
  PMQVOID pChannelExitParms,  /* Channel exit parameter block    */
  PMQVOID pChannelDefinition, /* Channel definition              */
  PMQLONG pDataLength,        /* Length of data                  */
  PMQLONG pAgentBufferLength, /* Length of agent buffer          */
  PMQVOID pAgentBuffer,       /* Agent buffer                    */
  PMQLONG pExitBufferLength,  /* Length of exit buffer           */
  PMQPTR  pExitBufferAddr,    /* Address of exit buffer          */
  char   controlFlag)         /* Flag to control the exit mode   */
{
  PMQCXP pExitParms = (PMQCXP) pChannelExitParms;
  PMQCD pChDef = (PMQCD) pChannelDefinition;
  static MQHCONN Hcon;        /* connection handle               */
  static logEnable = Ø;
  MQHOBJ HobjInqNL;           /* object handle                   */
  MQLONG CompCode;            /* completion code                 */
  MQLONG Reason;              /* reason code                     */
  MQOD InqObjDesc = {MQOD_DEFAULT};
  MQLONG openOptions, select[1], nameLen, count;
  MQLONG iAttrArr[1];
  PMQCHAR pNames, pNameStart, pFirstBlank;
  MQCHAR channelName[MQ_CHANNEL_NAME_LENGTH+1] = "";
  MQLONG channelLength;
  MQCHAR connectionName[sizeof(pChDef->ConnectionName)];
  MQLONG connectionLength;
  char logFileName[LOG_FILE_LENGTH + MQ_Q_MGR_NAME_LENGTH + 1];
  /* Mark channel, to be closed by default. */
  pExitParms->ExitResponse = MQXCC_CLOSE_CHANNEL;
  strncpy (queueManagerName, pChDef->QMgrName, MQ_Q_MGR_NAME_LENGTH);
  queueManagerName[MQ_Q_MGR_NAME_LENGTH] = ' ';
  pFirstBlank = strstr (queueManagerName, " ");
  *pFirstBlank = Ø;
  sprintf (logFileName, "%s%s%s", LOG_PATH, queueManagerName,
   LOG_SUFFIX);
  fp = fopen (logFileName, "a");
  switch (pExitParms->ExitId)
  {
   /* Only call as security exit is accepted       */
   case MQXT_CHANNEL_SEC_EXIT:
     switch (pExitParms->ExitReason)
     {
      /* Exit initialization, connect to the queue manager
        and store the handle in a static variable. */
      case MQXR_INIT:
        /* Print out, when in report-only mode.    */
        if ((controlFlag == SEC_REPORT_ONLY) && (fp != NULL))
        {
         logTS ();
         fprintf (fp, "*** Channel %s runs in report-only mode ***\n",
           pChDef->ChannelName);
        }
        /* Try to connect to the queue manager     */
        MQCONN (pChDef->QMgrName, /* queue manager */
         &Hcon, /* connection handle               */
         &CompCode, /* completion code             */
         &Reason); /* reason code                  */
        /* End program, if connection is not possible. */
        if ((Reason != MQRC_NONE) && (Reason !=
MQRC_ALREADY_CONNECTED))
         error_exit ("MQCONN ended with reason %ld\n", Reason);
     /* Read the security exit data and interpret it as a namelist. */
        InqObjDesc.ObjectType = MQOT_NAMELIST;
        strncpy (InqObjDesc.ObjectName, pChDef->SecurityUserData,
         sizeof (InqObjDesc.ObjectName));
        pNameStart = InqObjDesc.ObjectName;
        pFirstBlank = strstr (InqObjDesc.ObjectName, " ");
        /* The security exit data is not empty */
        if (pFirstBlank-pNameStart > Ø)
        {
         /* Inquire the namelist. */
         openOptions = MQOO_INQUIRE;
         MQOPEN (Hcon,  /* connection handle            */
           &InqObjDesc, /* object descriptor for queue   */
           openOptions, /* open options                 */
           &HobjInqNL,  /* object handle                */
           &CompCode,   /* MQOPEN completion code       */
           &Reason);    /* reason code                  */
         /* Inquiry was NOT successful.                 */
         if (Reason != Ø)
         {
           /* Mark channel, to be closed.               */
           pExitParms->ExitResponse = MQXCC_CLOSE_CHANNEL;
           if ((Reason == MQRC_UNKNOWN_OBJECT_NAME) && (fp != NULL))
           {
            logTS ();
            fprintf (fp, "Namelist \"%.*s\" is not known\n",
              pFirstBlank-pNameStart, InqObjDesc.ObjectName);
           }
           if (controlFlag == SEC_CHECK_PERM)
           {
            error_exit ("MQOPEN NL ended with reason %ld\n",
              Reason);
           }
         }
         /* Inquiry was successful. */
         else
```

```
{
  /* Inquire the NAMCOUNT attribute.                 */
  select[Ø] = MQIA_NAME_COUNT;/* attribute selectors */
  MQINQ (Hcon, /* connection handle                  */
   HobjInqNL, /* object handle                       */
   1L, /* Selector count                             */
   select, /* Selector array                         */
   1L, /* integer attribute count                    */
   iAttrArr, /* integer attribute array              */
   Ø, /* character attribute count                   */
   NULL, /* character attribute array                */
   &CompCode, /* completion code                     */
   &Reason); /* reason code                          */
  if (controlFlag == SEC_CHECK_PERM)
   error_exit ("MQINQ NL ended with reason %ld\n", Reason);
  /* Allocate memory for the NAMES attribute.        */
  pNames = (char *) malloc (iAttrArr[Ø] *
   MQ_NAMELIST_NAME_LENGTH);
  /* Inquire the NAMES attribute.                    */
  select[Ø] = MQCA_NAMES;/* attribute selectors      */
  MQINQ (Hcon, /* connection handle                  */
   HobjInqNL, /* object handle                       */
   1L, /* Selector count                             */
   select, /* Selector array                         */
   ØL, /* integer attribute count                    */
   iAttrArr, /* integer attribute array       */
   iAttrArr[Ø]*MQ_NAMELIST_NAME_LENGTH,
     /* character attribute count             */
   pNames,    /* character attribute array    */
   &CompCode, /* completion code              */
   &Reason);  /* reason code                  */
  if (controlFlag == SEC_CHECK_PERM)
   error_exit ("MQINQ NL ended with reason %ld\n", Reason);
  /* Compare the connection name to the defined addresses. */
  for (count = Ø; count < iAttrArr[Ø]; count++)
  {
   pNameStart = &pNames[count*MQ_NAMELIST_NAME_LENGTH];
   pFirstBlank = strstr (pNameStart, " ");
   nameLen = pFirstBlank - pNameStart;
   if (strncmp (pNameStart, pChDef->ConnectionName,
     nameLen) == Ø)
   {
     /* Mark channel, Not to be closed.       */
     pExitParms->ExitResponse = MQXCC_OK;
   }
  }
  /* Free the memory for the NAMES attribute. */
  free (pNames);
  /* Close the namelist.                     */
  MQCLOSE (Hcon, /* connection handle         */
    &HobjInqNL,
    MQCO_NONE,
    &CompCode,
    &Reason);
   if (controlFlag == SEC_CHECK_PERM)
    error_log ("MQCLOSE NL ended with reason %ld\n",
      Reason);
 }
}
/* The security exit data is empty */
else
{
 /* Mark channel, to be closed. */
 pExitParms->ExitResponse = MQXCC_CLOSE_CHANNEL;
 if (fp != NULL)
 {
   logTS ();
   fprintf (fp,
    "SCYDATA of channel %s contains not a namelist\n",
    pChDef->ChannelName);
 }
 if (controlFlag == SEC_CHECK_PERM)
   error_exit ("MQOPEN NL ended with reason %ld\n", Reason);
}
if ((pExitBufferAddr != NULL) && (*pExitBufferAddr != NULL))
{
 free(*pExitBufferAddr);
 *pExitBufferAddr = NULL;
 *pExitBufferLength = Ø;
}
pNameStart = pChDef->ChannelName;
pFirstBlank = strstr (pNameStart, " ");
if (pFirstBlank > pNameStart)
 channelLength = pFirstBlank-pNameStart;
else
 channelLength = strlen (pChDef->ChannelName);
strncpy (channelName, pNameStart, channelLength);
pNameStart = pChDef->ConnectionName;
pFirstBlank = strstr (pNameStart, " ");
connectionLength = pFirstBlank-pNameStart;
strncpy (connectionName, pNameStart, connectionLength);
/* Channel is marked to be closed.       */
if (pExitParms->ExitResponse != MQXCC_OK)
{
 if (fp != NULL)
 {
   /* Channel access is revoked by exit. */
   if (controlFlag == SEC_CHECK_PERM)
   {
    logTS ();
```

```
            fprintf (fp, "Access to channel \"%.*s\" from address
\"%.*s\" is revoked by exit\n",
              channelLength, channelName,
              connectionLength, connectionName);
           }
           /* Channel access is NOT revoked in report-only mode. */
           else
           {
            logTS ();
            fprintf (fp, "Access to channel \"%.*s\" from address
                \"%.*s\" would be revoked by exit\n",
              channelLength, channelName,
              connectionLength, connectionName);
           }
         }
         /* Mark channel, Not to be closed in report-only mode.  */
         if (controlFlag == SEC_REPORT_ONLY)
           pExitParms->ExitResponse = MQXCC_OK;
        }
        else
        {
         /* Report allowed access in report-only mode.            */
         if ((controlFlag == SEC_REPORT_ONLY) && (fp != NULL))
         {
           logTS ();
           fprintf (fp, "Access to channel \"%.*s\" from address
               \"%.*s\" would be allowed\n",
            channelLength, channelName,
            connectionLength, connectionName);
         }
        }
        break;
      case MQXR_INIT_SEC:
        /* Mark channel, Not to be closed. */
        pExitParms->ExitResponse = MQXCC_OK;
        break;
      case MQXR_SEC_MSG:
        /* Mark channel, Not to be closed. */
        pExitParms->ExitResponse = MQXCC_OK;
        break;
      case MQXR_TERM:
        /* Mark channel, Not to be closed. */
        pExitParms->ExitResponse = MQXCC_OK;
        break;
      default:
        /* Mark channel, to be closed.     */
        pExitParms->ExitResponse = MQXCC_CLOSE_CHANNEL;
     }
     break;
   default:
     error_log ("Invalid exit ID %ld\n", pExitParms->ExitId);
     /* Mark channel, to be closed.        */
     pExitParms->ExitResponse = MQXCC_CLOSE_CHANNEL;
     break;
  }
  if (fp != NULL)
   fclose (fp);
}
/* Function, which prints out a time stamp and the name        */
/* of the queue manager.                                       */
static void logTS (void)
{
  struct tm *pTime;
  time_t localTime;
  char timeString[1ØØ];
  int len;
  if (fp != NULL)
  {
   localTime = time (NULL);
   pTime = localtime (&localTime);
   strcpy (timeString, asctime (pTime));
   len = strlen (timeString);
   timeString[len - 1] = Ø;
   fprintf (fp, "<%s; %s> ", timeString, queueManagerName);
  }
}
```

Hubert Kleinmanns
N-Tuition Business Solutions (Germany)

Triggering WMQ Workflow

INTRODUCTION

This article presents two scenarios where Java classes were written to interact with MQ Series, achieve legacy database integration, and trigger WMQ Workflow.

SCENARIO ONE

A CRM (customer relationship management) order management system used a view of the legacy database to provide a single view of a customer’s details. In special circumstances there was sometimes a requirement to update customer information directly from the order management system. WMQ was the messaging system in use so we implemented the solution using a Java program and interfaced it with a Siebel CRM system.

LEGACYUPDATELOG.JAVA

```
/* Program name: LegacyUpdateLog                                      */
/* Description: This java program shows MQ-based Java and DB2 in     */
/*              a two-phase commit                                    */
/* Function:                                                          */
/*  This program accepts a  database update statement from the CRM   */
/*  application.  This statement is then written to an MQSeries      */
/*  queue and executed against a database in the same unit of work.  */
/*  The user is then asked whether this unit of work should be       */
/*  committed or backed out.                                         */
/*  This program is called as follows:                               */
/*     java LegacyUpdateLog -q ... -m ... -d ...                     */
/*  where                                                             */
/*   -q is the queue where messages will be put                      */
/*   -m is the queue manager (if not specified default qmgr is used) */
/*   -d is the database that will be used                            */
/*  and possible input would be:                                      */
/*    update table set column1 = 'value1' where column2 = 'value2'   */
/* The output of this program can be verified by running:            */
/*     amqsbcg <queue name> <qmgr name>                              */
/* If the work was committed, the db command will be on the queue.   */
/* If the work was backed out, the queue will be empty.              */
/* The database can also be queried to confirm whether or not the    */
/* database update was committed or rolled back.                     */
/* In order to use this program, the following must be done:         */
/* - An application database must be created using DB2. This        */
/*   database will be one of the resources in the two phase commit.  */
/* - The MQSeries queue manager must be updated to recognize the     */
/*   database as a resource. This can be done by using the WMQ       */
/*   Services to look at the queue manager. The properties of the    */
/*   queue manager are then selected and updated for this database:  */
/*     Name: any name you wish to use                                */
/*     SwitcFile: <mq install>\java\lib\jdbc\jdbcdb2.dll             */
/*     XAOpenString: database name, userid, password                 */
/*     ThreadOfControl: PROCESS                                      */
/* Note 1: There are several changes to the way things work with     */
/* java. Details are supplied in the Using Java manual Chapter 7.    */
/* Note 2: The MQ System Administration manual gives additional      */
/* information on using MQSeries as a transaction manager.           */
/* Note 3: This program is designed to work with update statements   */
/* only. If you enter select statements they will throw an exception.*/
import com.ibm.mq.*;            // Include the MQ package
import java.io.*;
import java.lang.*;
import javax.sql.*;
import java.sql.*;
public class LegacyUpdateLog {
   private MQQueueManager qMgr;
   private String         qmgrName;
   private String         queueName;
   private String         dbName;
   private Connection     jdbcConn;
   public static void main (String args[]) {
        LegacyUpdateLog   mySample = new LegacyUpdateLog(args);
        mySample.start();
   }
   public LegacyUpdateLog(String[] args) {
        /* Get the command-line arguments */
        for( int i=Ø; i<args.length; i++ ) {
            String arg = args[i].toLowerCase();
            if( arg.equals("-m") ) {
                if ( i+1<args.length ) {
                   qmgrName = args[++i];
                } else {
                   System.out.println("didn't specify queue manager,
exiting");
                   System.exit(-1);
                }
            } else if( arg.equals("-q") ) {
                if ( i+1<args.length ) {
                   queueName = args[++i];
                } else {
                   System.out.println("didn't specify queue, exiting");
                   System.exit(-1);
                }
            } else if( arg.equals("-d") ) {
                if ( i+1<args.length ) {
                   dbName = args[++i];
                } else {
                   System.out.println("didn't specify datbase name,
exiting");
                   System.exit(-1);
                }
            } else {
              System.out.println( "Unknown argument: " + arg );
            }
        }
        /* Check that all arguments were entered. */
        if ( (queueName==null)
             || (dbName==null) ) {
```

```
          System.out.println("java LegacyUpdatelog -q ... -m ... -d
...");
          System.out.println("where -q is the queue");
          System.out.println("      -m is the queue manager");
          System.out.println("      -d is the database name");
          System.exit(-1);
       }
    }
    /* Put any Specific Initializations Here                  */
    public void init() {
    }
    public void start() {
       try {
          System.out.println("LegacyUpdatelog started...");
          /* Create a queue manager object and access the queue */
          /* that will be used for the putting of messages.     */
          qMgr = new MQQueueManager(qmgrName);
          int openOptions = MQC.MQOO_OUTPUT;
          MQQueue myQueue = qMgr.accessQueue(queueName, openOptions,
                                             null, null, null);
          /* Create a DB2 XA DataSource that we will use as the */
          /* place to perform database updates.                 */
          COM.ibm.db2.jdbc.DB2XADataSource myDataSource =
              new COM.ibm.db2.jdbc.DB2XADataSource();
          myDataSource.setDatabaseName(dbName);
          jdbcConn = qMgr.getJDBCConnection(myDataSource);
          /* Set up a reader to get the user input */
          InputStreamReader isr = new InputStreamReader(System.in);
          BufferedReader br     = new BufferedReader(isr);
          String runShow;
          System.out.println("LegacyUpdatelog ready for db command");
          /* As long as the applications sends data, */
          /* process it...                           */
          do {
             runShow = br.readLine();
             /* See if the user entered anything */
             if (runShow.length() > Ø) {
                qMgr.begin();
                /* Set up a new message with a format of string and */
                /* write the user input to it.                      */
                MQMessage myMessage = new MQMessage();
                myMessage.writeString(runShow);
                myMessage.format = MQC.MQFMT_STRING;
                MQPutMessageOptions pmo = new MQPutMessageOptions();
                pmo.options = pmo.options | MQC.MQPMO_SYNCPOINT;
                myQueue.put(myMessage, pmo);
                boolean   validStatement = true;
                Statement stmt = jdbcConn.createStatement();
                try {
                   int rowsUpdated = stmt.executeUpdate(runShow);
```

```
                  System.out.println("Rows updated: " + rowsUpdated);
               } catch (java.lang.Exception ex) {
                  validStatement = false;
                  System.out.println("Java exception: " + ex);
                  System.out.println("LegacyUpdatelog is designed to
                     work only with update statements.\n");
               }
               stmt.close();
         /* Ask if the db update, message put should be committed or  */
         /* backed out (if db command was valid).  If the command     */
         /* wasn't valid, we'll backout the qmgr update.              */
               if (validStatement) {
              System.out.println("Enter C to Commit or R to rollback");
                  runShow = br.readLine();
                  if (   (runShow.indexOf("c") >= Ø)
                      || (runShow.indexOf("C") >= Ø) ) {
                     qMgr.commit();
                  } else {
                     qMgr.backout();
                  }
               } else {
                  qMgr.backout();
               }
            }
            System.out.println("LegacyUpdatelog ready for db command");
          } while (runShow.length() > Ø) ;
          /* Before the program ends, we need to close all of our    */
          /* connections.                                            */
          myQueue.close();
          jdbcConn.close();
          qMgr.disconnect();
       }
       catch (MQException ex) {
          System.out.println("An MQ error occurred: " +
ex.completionCode + " " +
                            ex.reasonCode);
       }
       catch (java.io.IOException ex) {
          System.out.println("Java.io exception: " + ex);
       }
       catch (java.lang.Exception ex) {
          System.out.println("Java exception: " + ex);
       }
       System.out.println("LegacyUpdateLog finished...");
    }
}
```

SCENARIO TWO

A customized collection system tracks the accounts receivables and validates the payments against the legacy database. A Java program had to be written to interface it with XML and WMQ, in order to initiate a WMQ Workflow process. This program, included with the collection system, starts the MQ Workflow process.

STARTMQWORKFLOW.JAVA

```
// This program is used to start the sample MQ Workflow process using
// XML. Also, this process is started as a particular user that is not
// the WinNT logged on user. The user identifier is set to ADMIN
// (This user must be defined in the MQ Workflow runtime database)
// XML messages for MQSeries Workflow are put into the queue
// EXEXMLINPUTQ. The queue manager is set to FMCQM in this code.
// Process name = StartWorkflowRequest
// InputContainer = PersonInfo
// container members = FirstName,LastName,TaxID  are string types
// You will need to compile this using the command:
//     javac StartWorkflowRequest.java
// To run it, make sure the MQSeries Workflow server is started.
// Make sure you have imported and translated fmccred.fdl
// Type:   java StartWorkflowRequest
// A window/dialog is displayed that prompts you for the First
// Name,Last Name and TaxID. Press the pushbutton to start the process.
// You will get no response back.  Must login to client to verify
process start.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.text.*;
import javax.swing.event.*;
import com.ibm.mq.*;       //MQSeries java classes
public class StartRequestWorkflow extends JFrame implements
ActionListener, DocumentListener
{
   // variables used
   private JLabel fnameLabel= null;
   private JTextField fnameText = null;;
   private JLabel lnameLabel= null;
   private JTextField lnameText = null;
   private JButton startButton   = null;
   private JButton cancelButton = null;
   private Document myDocTextField = null;
   private String firstName     = null;
   private String lastName      = null;
   private MQQueueManager  wfqmgr = null;
   private MQQueue         xmlinputq = null;
   private String procTempName  = null;
   private String procInstName  = null;
   private String contName      = null;
   private String xmlRequestMessage = null;
   public StartWorkflowRequest()
   {
      // put title on window
      super( "Workflow Request Application" );
      //setSize(500,450);
      setLocation(300,300);       // center of display
      addWindowListener( new WindowAdapter( ) {
         public void windowClosing( WindowEvent e ) {
            System.exit( 0 );
         }
      } );
      JPanel rootPanel = new JPanel();
      rootPanel.setLayout(new BorderLayout());
      JPanel fnamePanel = new JPanel();
      fnamePanel.setLayout(new FlowLayout());
      fnameLabel = new JLabel("First Name: ");
      fnameText = new JTextField(25);
      fnamePanel.add(fnameLabel);
      fnamePanel.add(fnameText);
      JPanel lnamePanel = new JPanel();
      lnamePanel.setLayout(new FlowLayout());
      lnameLabel= new JLabel("Last Name:  ");
      lnameText = new JTextField(25);
      lnamePanel.add(lnameLabel);
      lnamePanel.add(lnameText);
      JPanel taxidPanel = new JPanel();
      taxidPanel.setLayout(new FlowLayout());
      taxidPanel = new JLabel("Tax ID:  ");
      taxidPanel = new JTextField(25);
      taxidPanel.add(taxidLabel);
      taxidPanel.add(taxidText);
      myDocTextField = lnameText.getDocument();
      myDocTextField.addDocumentListener(this);
      JPanel buttonPanel = new JPanel();
      buttonPanel.setLayout(new FlowLayout());
      startButton = new JButton("Start Workflow Request Process");
      cancelButton = new JButton("Cancel");
      buttonPanel.add(startButton);
      startButton.addActionListener(this);
      startButton.setEnabled(false);
      buttonPanel.add(cancelButton);
      cancelButton.addActionListener(this);
      rootPanel.add("North",fnamePanel);
      rootPanel.add("Center",lnamePanel);
      rootPanel.add("South", buttonPanel);
      getContentPane( ).setLayout(new BorderLayout( ));
      getContentPane( ).add("Center", rootPanel);
      pack();
```

```
      setVisible(true);
   }
   public void insertUpdate(DocumentEvent evtDoc)
   {
       if (myDocTextField.getLength() > Ø)
          startButton.setEnabled(true);
   }
   public void removeUpdate(DocumentEvent evtDoc)
   {
        if (myDocTextField.getLength() > Ø)
          startButton.setEnabled(true);
       else
          startButton.setEnabled(false);
   }
   public void changedUpdate(DocumentEvent evtDoc)
   {
      // nothing here
   }
   public void actionPerformed( ActionEvent e )
   {
    // See if the start button was pressed
    if ( e.getActionCommand( ) == "Start Workflow Request Process" ) {
       // get the data from the input fields
       firstName=fnameText.getText();
       lastName=lnameText.getText();
       taxid = taxidText.getText();
       System.out.println("Firstname = '" + firstName + "'");
       System.out.println("Lastname = '" + lastName + "'");
       System.out.println("Tax ID = '" + taxID + "'");
       // Connect to queue manager and open the queue
       try
       {
          wfqmgr = new MQQueueManager( "FMCQM" );
          // SET_IDENTITY_CONTEXT is used to change the useridentifier
          int openOptions = MQC.MQOO_OUTPUT |
MQC.MQOO_SET_IDENTITY_CONTEXT;
          xmlinputq = wfqmgr.accessQueue("EXEXMLINPUTQ", openOptions,
null, null, null);
       }
       catch (MQException mqException)
       {
          System.out.println("MQException has been thrown on connect or
open");
          System.exit(99);
       }
       // Build the XML message that will start the process
       procTempName="WorkflowRequest";
       procInstName = lastName;
       contName = "PersonInfo";
       xmlRequestMessage=
              "<?xml version=\"1.Ø\" standalone=\"yes\"?>" +
              "\n <WfMessage>" +
              "\n <WfMessageHeader>" +
              "\n <ResponseRequired>No</ResponseRequired>" +
              "\n </WfMessageHeader>" +
              "\n <ProcessTemplateCreateAndStartInstance>" +
              "\n    <ProcTemplName>" + procTempName + "</
ProcTemplName>" +
              "\n    <ProcInstName>" + procInstName + "</ProcInstName>"
+
              "\n    <ProcInstInputData>" +
              "\n    <" + contName + ">" +
              "\n       <FirstName>" + firstName + "</FirstName>" +
                        "<LastName>" + lastName + "</LastName>" +
                        "<TaxID>" + TaxID + "</TaxId>" +
                     "</" + contName + ">"+
              "\n    </ProcInstInputData>" +
              "\n </ProcessTemplateCreateAndStartInstance>" +
              "\n </WfMessage>\n\n\n";
       try
       {
          // Set user identifier and write the message to the buffer
          MQMessage msg = new MQMessage();
          msg.userId = "ADMIN"; // use a userid other than WinNT userid
          msg.writeString(xmlRequestMessage);
          // Specify put options and put the message on the queue
          MQPutMessageOptions pmo = new MQPutMessageOptions();
          pmo.options = MQC.MQPMO_SET_IDENTITY_CONTEXT;   // to use
non-WinNT userid
          xmlinputq.put(msg, pmo);
          System.out.println("Have put xml message on queue.");
       } // end try
       catch(MQException mqexception)
       {
           System.out.println("MQException thrown on put of message.");
       }
       catch (java.io.IOException exception)
       {
           System.out.println("An error occurred writing string into
the message buffer. Exception = " + exception);
           this.cleanup();
           this.endApp();
       }
         // close queue and disconnect
       this.cleanup();
       // End application
       this.endApp();
    } // endif Start button pressed
    // See if the cancel button was pressed
    if ( e.getActionCommand( ) == "Cancel" ) {
       this.endApp();
    }
   }  //end actionPerformed
   // Close the queue and disconnect from qmgr
   public void cleanup()
   {
      System.out.println("Closing queue and disconnect from qmgr.");
      try
      {
         xmlinputq.close();
      }
      catch(MQException mqexecption)
      {
          System.out.println("MQException has been thrown on close of
queue.");
      }
      try
      {
         wfqmgr.disconnect();
      }
      catch(MQException mqexception)
      {
         System.out.println("MQException has been thrown on disconnect
from qmgr.");
      }
   } // end ActionPerformed
   // End the application
   public void endApp()
   {
      System.out.println("Leaving this application.");
      System.exit(Ø);
   }
   // Main
   public static void main( String[] args )
   {
       StartWorkflowRequest startworkflowRequest = new
StartWorkflowRequest( );
   }
}  // end of StartWorkflowRequest class
```

Vikas Baruah
American Management Systems (USA)

BAR file deploy in Message Broker V5.0 Toolkit

The BAR (Broker Archive) file deploy is one of the newest and most important pieces of functionality in WebSphere Business Integration Message Broker V5.0 Toolkit (hereafter referred to as Message Broker V5.0). It is a radical change from the deploy method in V2.1.

The BAR file is a unit of deployment to a broker. It can be seen as a zip file containing message flows and message sets, thus forming the unit to be deployed to a broker. This also means that the assignments pane in V2.1 does not exist in V5.0 in a perspective of its own. The BAR is a fresh functional addition to the Message Broker V5.0 Toolkit.

The most functional use of a BAR file probably lies in its portability. A user can create a number of message flows and message sets, add them to a BAR file, and then export the BAR file. This means that the BAR file can be used in different workspaces without the need to export the actual message flows and message sets. The message flows and message sets are compiled and stored in the BAR file.

The following sections describe how to create a BAR file and the source, add message flows and message sets to a BAR file, and ways of deploying a BAR file to a broker with other important functional aspects.

I use 'BARMessageFlow' and 'BARMessageSet' as a message flow and message set respectively, with the assumption that they exist in the workspace for illustrative purposes.

REQUIREMENTS

The following set-up is required before proceeding:

- A configuration manager with its queue manager and listener running.

- A broker with its queue manager and listener running. (It's easier if the configuration manager and broker share the same queue manager and listener.)
- A domain connection to the configuration manager and a broker to deploy to in the toolkit.

CREATE A BAR FILE

A BAR can be created using the file menu or the toolbar button in the Broker Administration perspective, as shown in Figure 1.

- The BAR file has to be part of a server project, so create a server project from the file menu: File->New->Project. Select Server in the left-hand pane and Server Project in the right pane. Click Next.
- Give the project a name, eg 'BARProject'. Make sure the Use Default checkbox is checked and click Finish. The project is created and the Server perspective is opened. Close the Server perspective.
- In the Broker Administration perspective click the Create a New Broker Archive File toolbar button.
- In the New Message Broker Archive window select the server project just created and enter a name for the BAR file to be created, for example 'BARFile1' and click Finish.
- An editor opens in the workspace, showing the name of the BAR file just created with a *.bar* extension. In the Broker

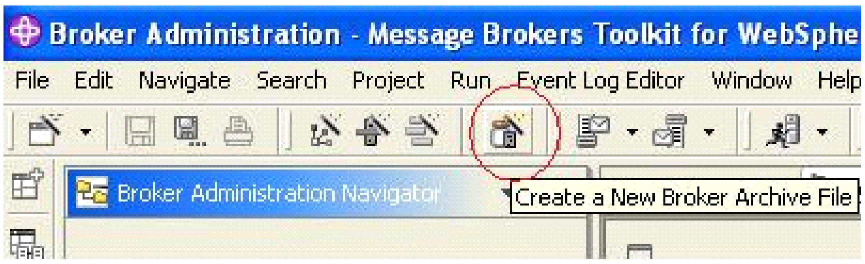

Broker Administration - Message Brokers Toolkit for WebSphe
File Edit Navigate Search Project Run Event Log Editor Window Help
Broker Administration Navigator
Create a New Broker Archive File

Figure 1: Creating a BAR file

Administration Navigator pane of the Broker Administration perspective, the BAR file is listed under the Broker Archives folder as a child of the Server project it belongs to (see Figure 2).

(Note: assuming defaults were used, the BAR file is created and stored under *<WBIMBv5.0InstallDirectory>/eclipse/workspace/<ProjectName>* on the system.)

ADD MESSAGE FLOWS AND MESSAGE SETS TO A BAR FILE

After the BAR file is created the message flows and message sets to be deployed have to be added to the BAR file.

- Click the Add button on the BAR file editor pane, as shown in Figure 3.
- In the Add to Broker Archive window that opens, select the deployable resources and check the Include message flow/set source box, as shown in Figure 4. This is a very useful feature as it includes the source of the message flow and message set in the BAR file. A user can then extract the source message flow and message set from the BAR file if required. Click OK.

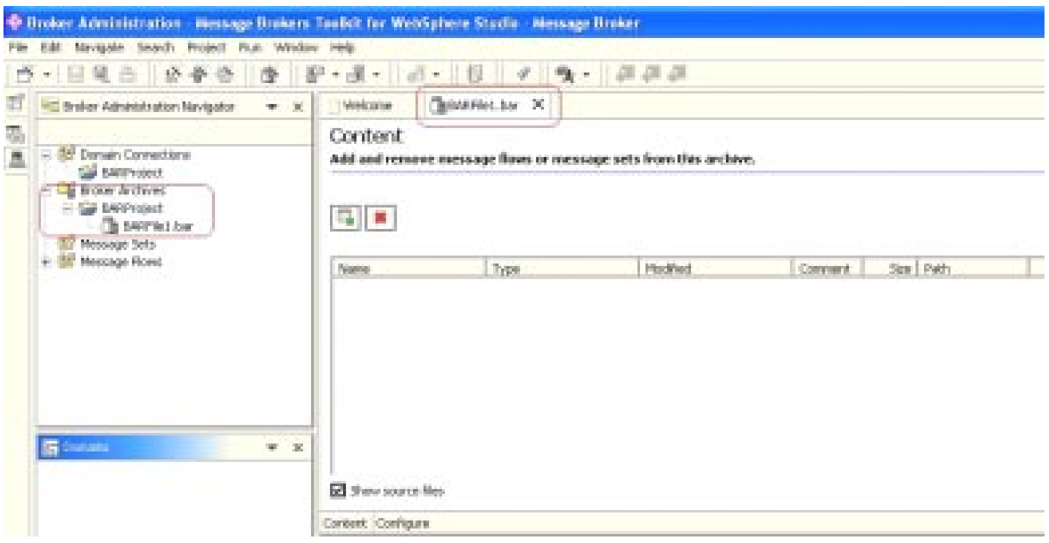

Broker Administration - Message Brokers Toolkit for WebSphere Studio - Message Broker
File Edit Navigate Search Project Run Window Help
Broker Administration Navigator
Domain Connections
BARProject
Broker Archives
BARProject
BARFile1.bar
Message Sets
Message Flows
Outline
Welcome
BARFile1.bar
Content
Add and remove message flows or message sets from this archive.
Name
Type
Modified
Comment
Size
Path
Show source files
Content
Configure

Figure 2: Viewing the created BAR file

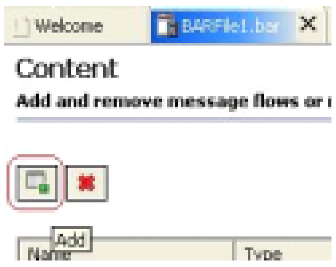
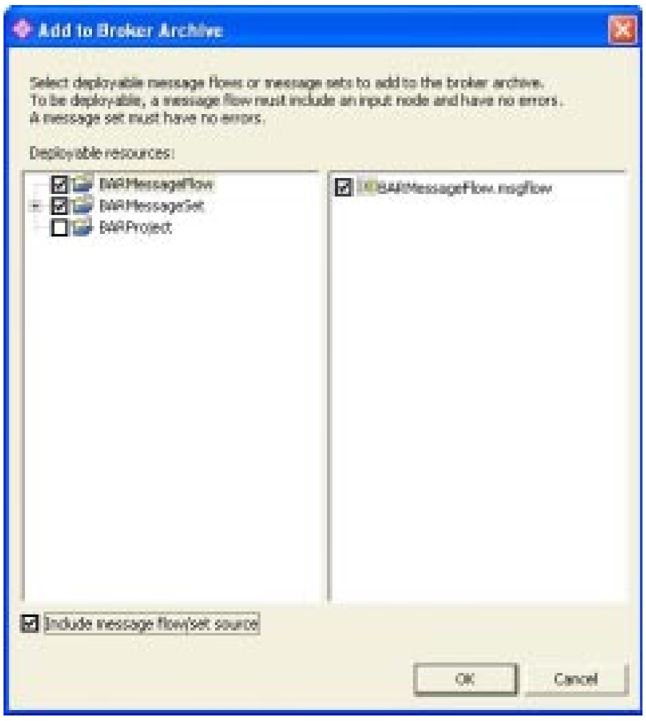


Figure 3: Adding and removing message flows



Select deployable message flows or message sets to add to the broker archive.
To be deployable, a message flow must include an input node and have no errors.
A message set must have no errors.

Deployable resources:

- ✓ BARMessageFlow
- ✓ BARMessageSet
- ☐ BARProject

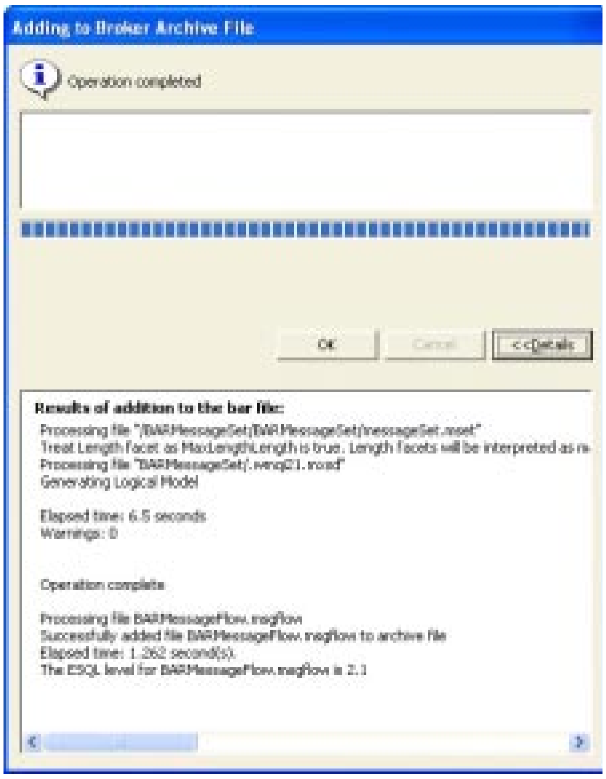
✓ BARMessageFlow.msgflow

✓ Include message flow/set source

Figure 4: Selecting the deployable resources

- A window opens showing the progress of adding resources to the BAR file. This is useful because any errors and warnings resulting from the actual message flows and message sets can be seen in the details pane of the window. Click OK. You can also check for successful deploy results in the event log (see Figure 5).

  (Note: message flow and message set projects with errors will not be added to the BAR file and the Details pane will outline the projects that have errors.)

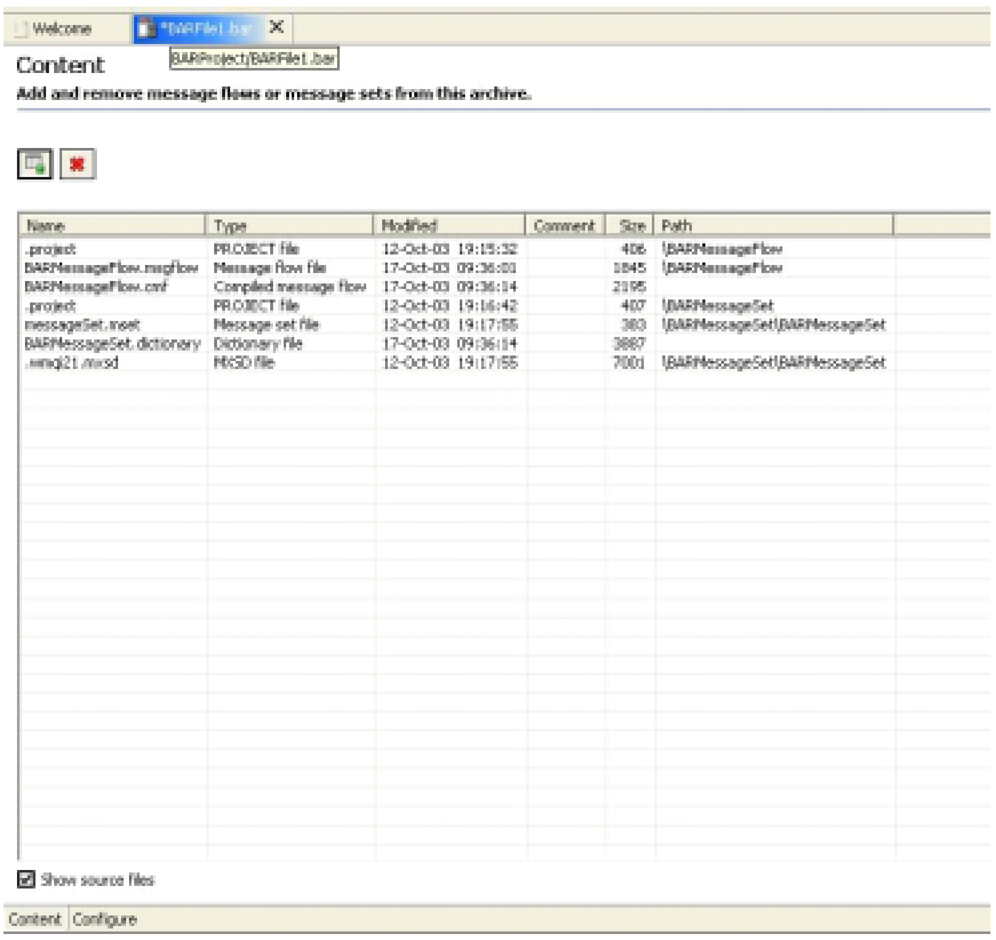


```
Results of addition to the bar file:
 Processing file "/BARMessageSet/BARMessageSet/messageSet.mset"
 Treat Length facet as MaxLength|Length is true. Length facets will be interpreted as re
 Processing file "BARMessageSet/.wmq21.mxsd"
 Generating Logical Model

 Elapsed time: 6.5 seconds
 Warnings: 0


 Operation complete

 Processing file BARMessageFlow.msgflow
 Successfully added file BARMessageFlow.msgflow to archive file
 Elapsed time: 1.262 second(s).
 The ESQL level for BARMessageFlow.msgflow is 2.1
```

Figure 5: Adding resources and checking results

- Once the resources have been successfully added to the BAR file the editor pane will display the list of the resources (see Figure 6).
- As can be seen from Figure 5, the BAR file now contains the resources with their source. The Compiled Message Flow and Dictionary File are generated as a result of the addition to the BAR file. These files bring the portability factor to the BAR file. With these files forming part of the BAR file the user can take the BAR file and deploy it in another workspace without any prior knowledge of the message flow and message set in the BAR file.
- The Show Source Files checkbox can be toggled to display or hide the source files. Also, the user can remove resources from the BAR file using the Remove button. Save the BAR file.

  (Note: if the BAR file is not saved the user will be asked to save it at the time of deploy. The asterisk in the title of the BAR file editor pane indicates that the BAR file has not been saved.)



Content

Add and remove message flows or message sets from this archive.

Name	Type	Modified	Comment	Size	Path
.project	PROJECT file	12-Oct-03 19:15:32		406	\BARMessageFlow
BARMessageFlow.msgflow	Message flow file	17-Oct-03 09:36:01		1845	\BARMessageFlow
BARMessageFlow.cmf	Compiled message flow	17-Oct-03 09:36:14		2195	
.project	PROJECT file	12-Oct-03 19:16:42		407	\BARMessageSet
messageSet.mset	Message set file	12-Oct-03 19:17:55		383	\BARMessageSet\BARMessageSet
BARMessageSet.dictionary	Dictionary file	17-Oct-03 09:36:14		3887	
.wmqi21.mxsd	MXSD file	12-Oct-03 19:17:55		7001	\BARMessageSet\BARMessageSet

Figure 6:Displaying the list of resources

DEPLOY A BAR FILE

Once all the deployable resources are added to the BAR file, it can be deployed using one of three methods.

Drag and drop

- In the Broker Administration Navigator pane click on the BAR file to be deployed.
- Drag the BAR file and drop it on the execution group of the broker in the Domains pane to initiate a deploy (see Figure 7).

(Note: the BAR file can only be dropped onto an execution group. A ‘no entry’ sign will appear if the user attempts to drop it anywhere else.)

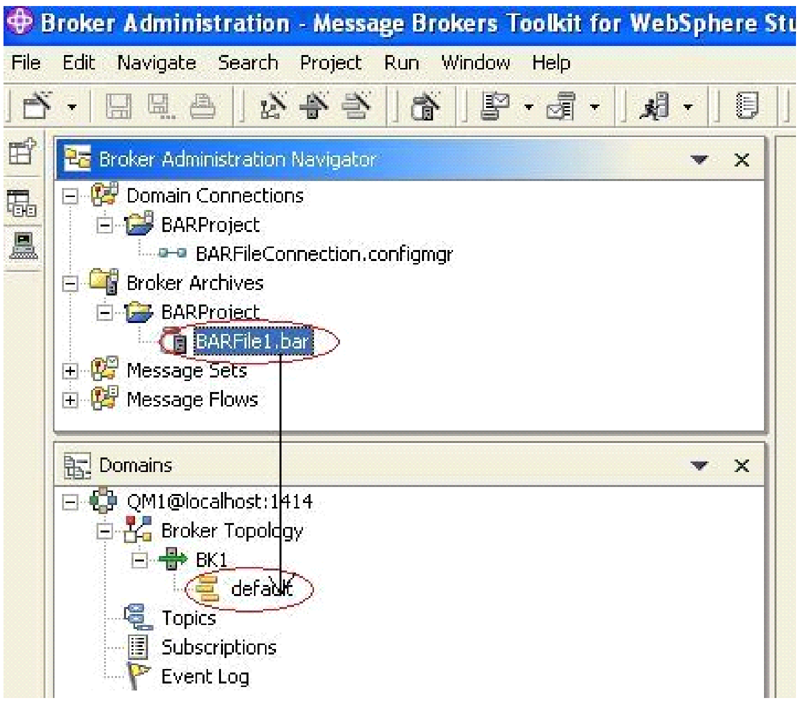

Broker Administration - Message Brokers Toolkit for WebSphere Stu
File Edit Navigate Search Project Run Window Help
Broker Administration Navigator
Domain Connections
BARProject
BARFileConnection.configmgr
Broker Archives
BARProject
BARFile1.bar
Message Sets
Message Flows
Domains
QM1@localhost:1414
Broker Topology
BK1
default
Topics
Subscriptions
Event Log

Figure 7: Drag and Drop method of initiating a deploy

Menu popup

- Right-click on the BAR file in the Broker Administration pane and select Deploy File… from the list.
- In the Deploy a BAR File… window that opens, select the execution group of the broker to initiate the deploy (see Figure 8). Click OK .

Command line deploy

The user also has an option to deploy the BAR file from the command line. This can be done using the **mqsideploy** command at the command prompt. In this case the command will be:

```
mqsideploy -b<broker name> -e<execution group name> -bar <BAR File
name>
```

The **mqsideploy** command also has additional options that can be used to dictate other aspects of the deploy, such as timeout value, read logs, etc.

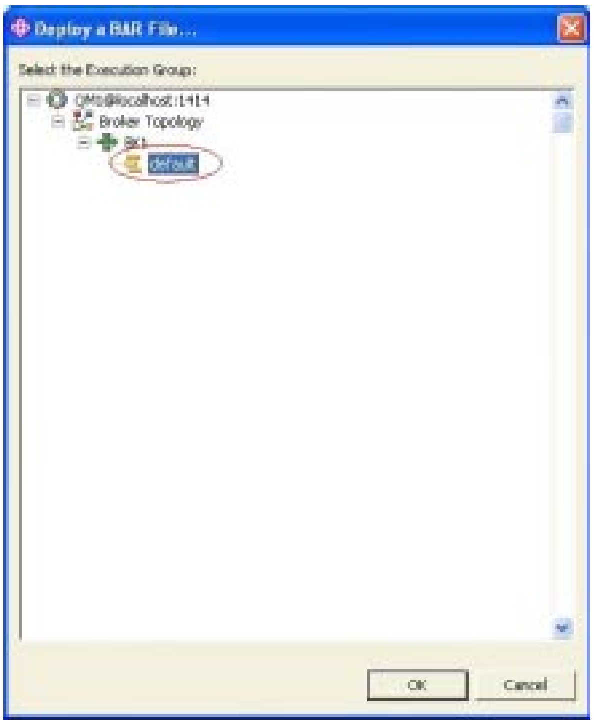


Figure 8: Using the popup menu to initiate a deploy

Check for deploy results

Once the deploy has been initiated and is successful, the following processes should be checked to confirm that the deploy was successful.

- A dialog box will appear, confirming a successful response from the configuration manager. Click 'OK'.

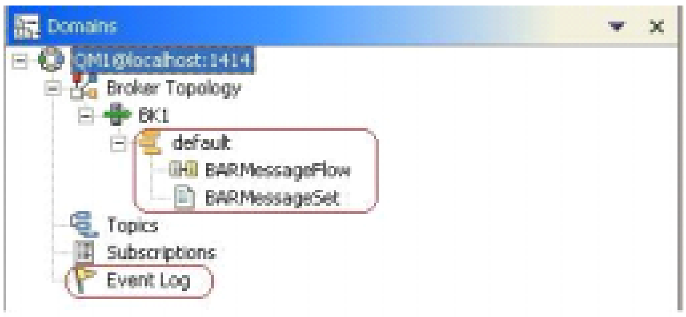
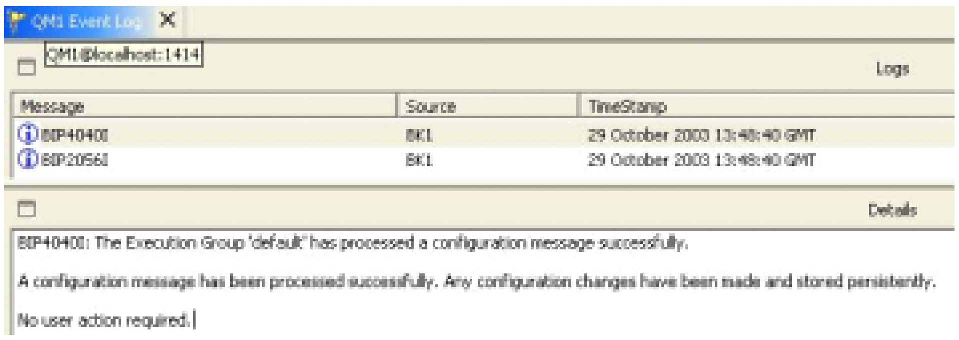


Figure 9: The Domains pane



Message	Source	TimeStamp
BIP4040I	BK1	29 October 2003 13:48:40 GMT
BIP2056I	BK1	29 October 2003 13:48:40 GMT

BIP4040I: The Execution Group 'default' has processed a configuration message successfully.

A configuration message has been processed successfully. Any configuration changes have been made and stored persistently.

No user action required.

Figure 10: Check for successful deploy messages

- The message flows and message sets added to the BAR file should appear under the execution group the BAR File was deployed to in the Domains pane, as shown in Figure 9.
- Open the Event Log from the Domain pane and check for successful deploy messages, as shown in Figure 10.

(Note: check the local system log for the broker to ensure no errors were given as a result of the deploy.)

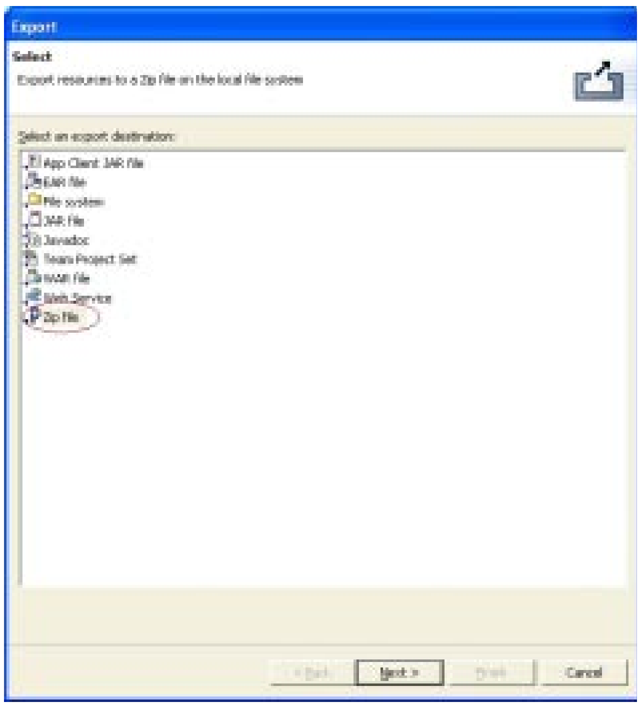
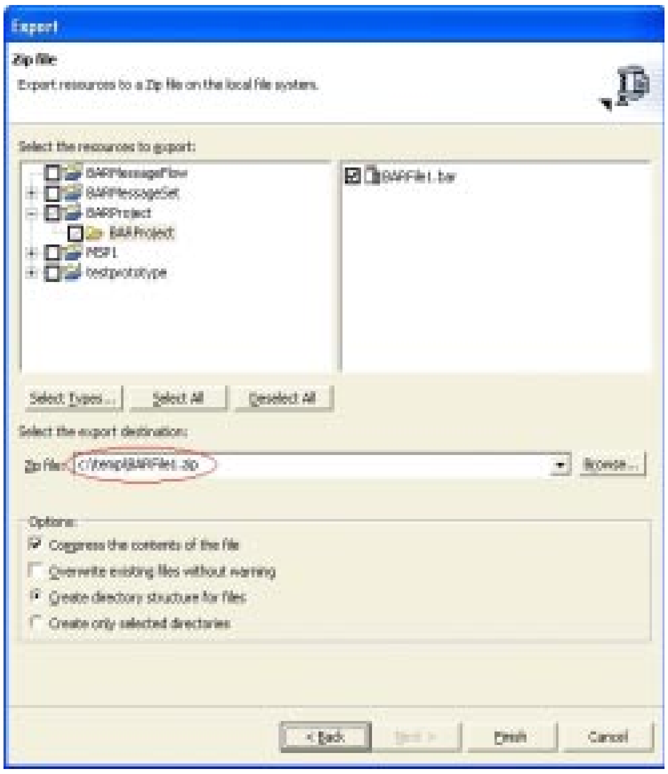


Figure 11: Exporting the BAR file

Export/import/reuse of the BAR file

As mentioned previously, the BAR file can be exported as a zip file and can be deployed in another workspace without needing the actual message flows and message sets. The BAR file can be exported using the following steps:

- Highlight the BAR File and select File -> Export. In the Export window, select Zip file, as shown in Figure 11. Click Next.
- Select the BAR File to be exported and give the name of the zip file to be created. You can compress the contents of the file in the options section, as Figure 12 illustrates. Click Finish.
- The *.zip* file now created is a portable unit of deployment that can be imported in another workspace and deployed. To import the BAR File, select File -> Import. In the Import window select Zip file and Click Next.
- Browse the *.zip* file to be imported and select the destination



Zip file

Export resources to a Zip file on the local file system.

Select the resources to export:

- [ ] BARMessageFlow
- [ ] BARMessageSet
- [ ] BARProject
  - [ ] BAR Project
- [ ] MSP1
- [ ] testprototype

- [✓] BARFile1.bar

Select the export destination:

Zip file: c:\temp\BARFiles.zip

Options:

- [✓] Compress the contents of the file
- [ ] Overwrite existing files without warning
- (●) Create directory structure for files
- ( ) Create only selected directories

Figure 12: Compressing the contents of the file

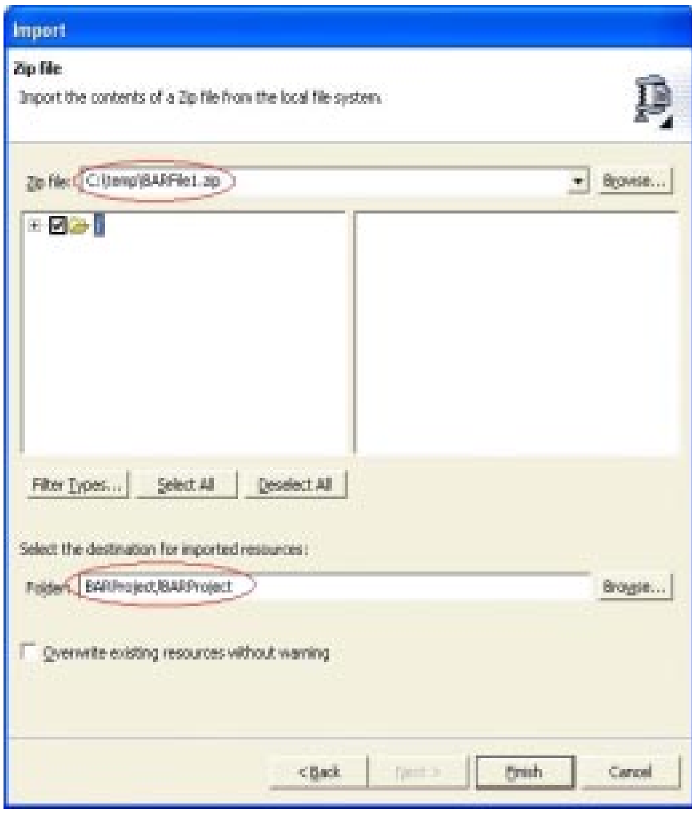

Import
Zip file
Import the contents of a Zip file from the local file system.
Zip file: C:\temp\BARFile1.zip
Browse...
Filter Types...
Select All
Deselect All
Select the destination for imported resources:
Folder: BARProject/BARProject
Browse...
Overwrite existing resources without warning
< Back
Next >
Finish
Cancel

Figure 13: Selecting the destination project folder

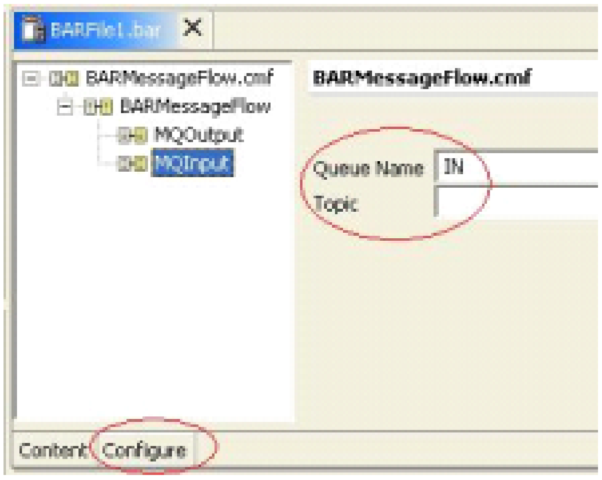


Figure 14: Configuring properties

project folder where the BAR file will be imported, as shown in Figure 13. Click Finish.

- The BAR File will appear under the name of the project it was imported into, and can be deployed as it is.

- If the BAR file contains a message flow, some properties can be configured. Open the BAR File editor and click on the Configure tab, as shown in Figure 14.
- The properties of the nodes can be configured and changed. Change the values as required and deploy the BAR File. Thus the user can modify the values to be in sync with respect to different deployment scenarios. The actual message flow is not required in order to change values.

Rohit Bhasin
IBM Hursley (UK)

MQ news

In a recent announcement on WebSphere security services IBM claims that forthcoming features in WebSphere Business Integration and WebSphere MQ will enable its mainframe and distributed customers to improve network performance by defining security policies for a select group of Web or legacy applications.

For instance, a customer may want to limit access to 20% of their applications – those that contain sensitive data – and provide more open access to other applications. This function is claimed to improve network performance because systems won't be tied up by unnecessary security checks.

For more information contact your local IBM representative.

* * *

Bristol Technology is shipping TransactionVision 4.0, the latest version of its transaction tracking and analysis software, featuring support for both WebSphere MQ and J2EE transactions.

The company claims that TransactionVision tracks transactions across each touch point, self-discovering transaction flows and content while providing real-time monitoring to pinpoint failures and ensure service levels.

The software includes: J2EE Sensor Support, automatic categorization of tracked transactions into user-defined classes, ie equity trades, foreign exchange transactions, etc, and enhanced reporting capabilities.

For more information contact:
Bristol Technology, 39 Old Ridgebury Road, Danbury, CT 06810-5113, USA.
Tel: +1 203 798 1007.
Fax: +1 203 798 1008.
Web: http://www.Bristol.com

Bristol Technology, Plotterweg 2A 3821 BB, Amersfoort, The Netherlands.
Tel: +31 33 450 50 50.
Fax: +31 33 450 50 51.

* * *

NEON Systems has recently launched Shadow Event Publisher, which is claimed to provide a single interface for the realtime capture and publishing of critical mainframe business events occurring within DB2, IMS, and CICS environments.

The company claims that, without touching the application code, events are captured in real time and 'pushed' asynchronously via multiple messaging protocols, HTTP and WebSphere MQ, to drive heterogeneous business processes and maintain data consistency.

For more information contact:
NEON Systems, 14100 Southwest Freeway Suite 500, Sugar Land, TX 77478, USA.
Tel: +1 281 491 4200.
Fax: +1 281 242 3880.
Web: http://www.neonsys.com

NEON Systems, 1 High Street, Windsor, Berks, SL4 1LD, UK.
Tel: + 44 1753 752800.
Fax: + 44 1753 752818.