MQ update

62

August 2004

In this issue

MQ Update

Published by

Xephon Inc
PO Box 550547
Dallas, Texas 75355
USA

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

Editor

Trevor Eddolls
E-mail: trevore@xephon.com

Publisher

Nicole Thomas
E-mail: nicole@xephon.com

Subscriptions and back-issues

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

Contributions

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

MQ Update on-line

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

Customizing Unix queue managers in an enterprise

INTRODUCTION

Unix administrators often have to manage lots of programs and tools on their systems, such as application servers, databases, security tools, back-up and monitoring utilities, and other applications. One of these applications may be WebSphere MQ (WMQ). Mostly, administrators do not want to know too many details about tools and utilities like WMQ; they just want to use it as a black box that performs the tasks it is designed for.

In the case of WebSphere MQ, there are lots of objects with many attributes that have to be customized. With graphical configuration tools like MQExplorer or MQMON (IBM SupportPac MO71) or other products, it is a little bit easier to configure WMQ. But most of the tools require a Windows platform for the tool itself and a client connection with a running command server on the WMQ system. It is difficult or even impossible to automate WMQ configuration with these tools.

The mechanism described in this article simplifies the administration and configuration of WMQ queue managers on Unix systems. It reduces to a minimum the knowledge of WMQ required by administrators and it allows interactive as well as automated queue manager configuration.

DESCRIPTION OF THE TOOL

Intention for mkqmgr

In WMQ environments, often several servers have to be set up with a similar, but not identical, configuration (eg development, application testing, integration testing, production, back-up). The names and attributes of most of the WMQ objects (queues, processes, etc) could be the same or similar, but at least the name of the queue manager and the names and attributes of the channel (eg IP addresses or IP ports) have to be different.

One possibility is to clone queue managers (eg with saveqmgr, the IBM SupportPac MS03), modify the created file, and read it into another queue manager. In this case, the administrator has to know the names and attributes that need to be modified and must alter them with a text editor like vi. The administrator has also to remove test and dummy queues, which otherwise will be copied to the new queue manager too.

An alternative procedure would be to provide the administrators with a set of WMQ definition files that they have to read into the queue managers. In this case, the different definition files have to be stored somewhere. Changes (eg a new queue) have to be made in all of the files. Some mechanism to handle the definition files has to be set up and managed.

The solution described in this article is script-based and solves the problems mentioned above. With the script, mkqmgr administrators are able to configure queue managers without needing huge amounts of WMQ skill. The script can be automated and is usable with software distribution tools or via job control. It is developed and tested on AIX and Solaris, but an adaptation to other Unix systems should not be a big deal.

How mkqmgr works

Generally, queue managers differ only in a few attributes. For example remote queues have to address another queue manager, a channel may use another transmission queue, and so on. The trick is to separate the variable values from the fixed definitions. In the solution described here, the variable parts are stored in a configuration file specific to the queue manager and the fixed definitions are put into a template file.

The template file is independent of the environment, which means it is the same for development, testing, production, and so on. The template file contains MQSC definition commands for every WMQ object that has to be defined to the queue manager. Labels or wildcards replace names and attributes, or even parts of them, that need to be variable. Wildcards are surrounded by a less than and a greater than sign (< and >):

```
*-----------------------------------------------------------------
* Definition of the remote request queue.
*-----------------------------------------------------------------
DEFINE QREMOTE(<Q_PREFIX>.QUESTION) +
       DESCR('Request queue for <Q_PREFIX>') +
       XMITQ(<REMOTE_QMGR>) +
       RNAME(<Q_PREFIX>.QUESTION) +
       RQMNAME(<REMOTE_QMGR>) +
       REPLACE
```

The configuration file differs for each system or queue manager and contains all the environment-specific data; this means translations for the wildcards. This file contains lines with pairs of wildcards and a value for each. The values may be followed by a comment with a preceding hash sign (#):

```
Q_PREFIX            TEST      # Prefix for the queues.
REMOTE_QMGR         QMTEST1   # Name of the remote queue manager.
```

In general, the configuration file will be a very short and easily-readable file, whereas the template file looks very similar to an MQSC commands file, and so a WMQ novice would find it hard to read.

mkqmgr is a script that uses these two files for input for the creation and configuration of a queue manager. mkqmgr first parses the template file and extracts any wildcards in it. Each wildcard is shown and the script asks the user to enter a value for it. When a configuration file exists and the wildcard is found there, the value will be shown as a default. The administrator may now change the value or accept it by pressing *Return*.

```
Enter the value for parameter "Q_PREFIX" [TEST]? PROD<RETURN>
Enter the value for parameter "REMOTE_QMGR" [QMTEST1]? QMPROD1<RETURN>
```

The new values are stored in the configuration file:

```
Q_PREFIX            PROD      # Prefix for the queues.
REMOTE_QMGR         QMPROD1   # Name of the remote queue manager.
```

mkqmgr stores the last ten versions of the configuration file.

Last but not least, mkqmgr creates a definition file, which looks similar to the template file, but the wildcards are replaced with the values from the configuration file:

```
*-----------------------------------------------------------------
```

```
* Definition of the remote request queue.
*------------------------------------------------------------------
DEFINE QREMOTE(PROD.QUESTION) +
       DESCR('Request queue for PROD) +
       XMITQ(QMPROD1) +
       RNAME(PROD.QUESTION) +
       RQMNAME(QMPROD1) +
       REPLACE
```

This definition file is now read into the queue manager. If the queue manager does not exist, mkqmgr asks the user and tries to create it.

The template file has first to be created by a WMQ administrator. The configuration file may be provided with the system configuration or may be created by the script mkqmgr itself. For use with software distribution, the WMQ administrator has to create a template file and one configuration file for each queue manager. New objects (eg a new queue, but with the same prefix) have only to be added to the template file.

For additional environments (eg a second production system) only a new configuration file has to be created. Even an additional wildcard – which needs a modification to the template file and the configuration files – is quite easy, because the WMQ administrator needs to alter only one template file (maybe adding a suffix for each channel or queue) and to add only one line per configuration file.

Installation of mkqmgr

There is nothing to install. Just copy the script to your working directory or to a directory within your search path. The script always looks in your current working directory for template and configuration files. You have, at least once, to create a template file, and run the script to create a configuration and a definition file. If no template file is used, the script creates an empty queue manager.

Call mkqmgr

When you run mkqmgr without valid options, you will get a short usage message:

```
usage: mkqmgr [-v {12}] queue_manager [-t template_file[.tpl]]
       mkqmgr -q queue_manager -t template_file[.tpl]

       -v: Set verbose (1, 2); default level is "Ø".
       -q: Run in "quiet" mode; default mode is "interactive".
```

mkqmgr has two different modes – an interactive and a quiet mode. Interactive mode means that the script guides the administrator through the creation and configuration of a queue manager. The administrator may use a template file and, if a configuration file exists, this will be read. The administrator has to specify the name of the queue manager. If no template file is specified, mkqmgr asks for one. If you ignore it, mkqmgr creates an empty queue manager. Some more output is generated when the verbose level (-v number) is increased to ‘1’ or ‘2’.

In quiet mode, mkqmgr may be used for automatic WMQ configuration, eg via software distribution tools. In this case both the template file and the configuration file must exist. The script does not ask for the attributes, it just replaces the wildcards. Each wildcard in the template file must have a value defined in the configuration file. The software distribution mechanism has to distribute mkqmgr, the template, and the configuration file. A postinstall script has to call mkqmgr with the correct parameters and perform the steps in the created README file (see below).

DESCRIPTION OF THE CODE

The script mkqmgr is divided into six parts. The first three parts of the script contain general functions, which may be used in the same or a similar way in any other scripts. Part IV contains the main logic of the script. These functions handle the input and output files and create an MQSC definition file. Parts V and VI contain functions to create and configure a queue manager and to set up some start-up scripts.

Part I – general input functions

The first part of the mkqmgr consists of two general input functions. Both functions are able to display a default answer, which will be used by pressing *Return*. The first attribute contains a string with the question that is to be displayed. The second attribute is the name of the parameter where the answer will be stored (not the answer itself!). If the second attribute was set previously, that value will be used as a default answer:

```
# Set the default answer to 'n'.
enter_tpl="n"
read_answer "   Do you have a template file" enter_tpl
```

The difference between the two functions is that the first one (read_answer) allows only Y and N, whereas the second function (read_value) accepts any text as input. In addition, read_value accepts the string optionally as a third parameter. In this case, the entered value may be empty, otherwise only non-empty values will be accepted:

```
# Ask for the name of the template file.
read_value " Enter the name of the template file" TEMPLATE_FILE optional
```

Part II – script initialization

The next set of functions is used to initialize the script environment. The function initialise defines default values for several parameters. It calls the function check_arguments, which reads and checks all command line parameters. If invalid arguments are found, a usage message is displayed and the script exits (function show_usage).

Part III – logging functions

The third part of the script code contains some logging functions. The function log_start creates the log file and prints a short message with a time stamp. In interactive mode a short description about the script function is shown. The function log_finish again prints a time stamp and exits the program. In interactive mode some statistics are shown.

Screen output (in interactive mode only):

```
--------------------------------------------------------------------
Welcome to the WebSphere MQ configuration.
The log file is "TESTQM.log".
This script will guide you through the configuration
of a WebSphere MQ queue manager. If the queue manager
```

```
does not already exist, it will be created.
-----------------------------------------------------------------------

...

-----------------------------------------------------------------------
Queue manager configuration finished with:
   2 information(s)
-----------------------------------------------------------------------
```

Log file output:

```
***********************************************************************
***********************************************************************
***
*** Queue manager configuration started at Thu Mar 25 15:18:59 MET 2004.
***
***********************************************************************
***********************************************************************

...

***********************************************************************
***********************************************************************
***
***Queue manager configuration finished at Thu Mar 25 15:19:16 MET 2004.
***
***********************************************************************
***********************************************************************
```

The function log_msg is used to print out a message to the log file as well as (in interactive mode) to the screen. The first argument of this function defines the type of message (normal (m)essage, (i)nformation message, (w)arning message, (e)rror message, or a (s)eparator line):

```
...
log_msg "S"
log_msg "M" "Read configuration..."
...
log_msg "I" "   Use template file \"$TEMPLATE_FILE\"."
...
```

Screen output (in interactive mode only):

```
...
-----------------------------------------------------------------------
Read configuration...
...
   INFO: Use template file "APPL.tpl".
...
```

Log file output:

```
...
**********************************************************************
* Read configuration...
...
*    INFO: Use template file "APPL.tpl".
...
```

Part IV – prepare configuration

Part IV of mkqmgr consists of several functions to read and write template, configuration, and definition files. The ‘core’ function of this part is prepare_qmgr_creation:

```
#
# Set up the input and output files to prepare
# the queue manager configuration.
#

function prepare_qmgr_configuration
{
   log_msg "S"
   log_msg "M" "Read configuration..."

   # Read the queue manager data.
   get_qmgr_data

   # Look for a template file.
   find_template_file

   if [ "$TEMPLATE_FILE" != "" ]
   then
      # Parse the template file and extract the wildcards.
      parse_template_file

      # Test if wildcards are to be replaced.
      if [ $NO_OF_PARAMETERS -gt 0 ]
      then
         # Read or create the local configuration file.
         read_config_file

         # Enter new values for the WebSphere MQ configuration.
         enter_parameters

         # Store the new values in the local configuration file.
         store_parameters
      else
989:          log_msg "I" "      Nothing to replace in template file."
990:       fi
991:    fi
992: }
```

This function calls first get_qmgr_data (in line 967), which finds out the next unused IP port in */etc/services*. This function now asks whether this queue manager should be used as a default. When the port 1414 is used, the ‘use as default’ flag is pre-selected.

The next function called by prepare_qmgr_creation is find_template_file (in line 970). This function checks whether a template file was specified on the command line and whether the name is valid. Otherwise every file located in the current working directory ending with .tpl is listed and the user is asked for a valid name. If only one template file has been found, its name will be shown as a default value.

When a template is used, it will be parsed by the function parse_template_file. This function extracts any strings surrounded by a less than and a greater than sign (< and >), and writes them to a temporary file (lines 773 to 799). In line 806, the wildcards are sorted and duplicate entries are removed. The function sets the value of NO_OF_PARAMETERS to the number of wildcards found (line 809) and stores any wildcard in the template file in an array PARAMETERS (line 814):

```
753: #
754: # Create a configuration file, if it does not already
755: # exist. The function reads the template file and
756: # extracts all wildcards, separated by '<' and '>'.
757: #
758:
759: function parse_template_file
760: {
761:    parse_file=$QMGR_NAME.parse
762:    tmp_parse=$CONFIG_FILE.tmp
763:
764:    log_msg "M" "   Parsing template file..."
765:
766:    # Remove the temporary file, if necessary.
767:    if [ -e "$tmp_parse" ]
768:    then
769:       rm $tmp_parse
```

```
fi

# Search the wildcards in the template file.
line_list='grep '<.*>' $TEMPLATE_FILE'

# Add a blank before each '<'.
line_list='echo "$line_list" | sed -e "s/</ </g"'

# Extract the wildcard names from the lines.
for line in $line_list
do
   # Calculate the length of the line.
   len='echo $line | awk '{print length ($1)}''

   # The line must contain at least three characters.
   if [ $len -gt 2 ]
   then
      # First character of a valid line must be '<'.
      buffer="<'echo $line | cut -c 2-$len'"

      if [ "$buffer" = "$line" ]
      then
         # Extract the string between '<' and '>'.
   param='echo "$line" | sed -e "s/^<\([A-Za-z_].*\)>.*$/\1/g"'

         # Store the string in a temporary file.
         echo "$param\t" >> $tmp_parse
      fi
   fi
done

# Test if wildcards are to be replaced.
if [ -e $tmp_parse ]
then
   # Remove duplicates lines, sort and store the parameters in
   # the configuration file.
   cat $tmp_parse | sort -u > $parse_file

   # Count the number of configuration parameters.
   NO_OF_PARAMETERS='cat $parse_file | wc -l'

   # Store the parameter names in an array.
   if [ $NO_OF_PARAMETERS -gt 0 ]
   then
      set -A PARAMETERS 'awk '{print $1}' $parse_file'
   fi

   # Remove the temporary files.
   rm $tmp_parse
   rm $parse_file
```

```
820:    fi
821: }
```

If any wildcards are found, the function read_config_file looks for an existing configuration file. Each element of the array PARAMETERS will be searched in the first column of this configuration file (lines 839 and 840). If the parameter is found, the second column of the line is set into the string tmp_values and the rest of the line behind the hash sign (#) is stored into the string tmp_comments (lines 842 to 854). Blanks within the comments are replaced in line 840 by underscores (_). At the end of the function (lines 861 and 862), the contents of the temporary strings are stored into the arrays VALUES and COMMENTS:

```
823: #
824: # Read an existing configuration file. The
825: # parameters will be stored in two arrays.
826: #
827:
828: function read_config_file
829: {
830:    tmp_values=""
831:    tmp_comments=""
832:
833:    if [ -e $CONFIG_FILE ]
834:    then
835:       let idx=0
836:       while [ $idx -lt $NO_OF_PARAMETERS ]
837:       do
838:          # Store the old parameter value and the comment.
839:          val='grep -w "^${PARAMETERS[$idx]}" $CONFIG_FILE | awk
'{print $2}''
840:          cmt='grep -w "^${PARAMETERS[$idx]}" $CONFIG_FILE | awk -
F'#' '{print $2}' | tr ' ' '_''
841:
842:          if [ "$val" = "" ]
843:          then
844:             tmp_values="$tmp_values \"\""
845:          else
846:             tmp_values="$tmp_values $val"
847:          fi
848:
849:          if [ "$cmt" = "" ]
850:          then
851:             tmp_comments="$tmp_comments \"\""
852:          else
853:             tmp_comments="$tmp_comments $cmt"
```

```
         fi

         let idx=$idx+1
      done
   fi

   # Store the parameter values and comments in two arrays.
   set -A VALUES $tmp_values
   set -A COMMENTS $tmp_comments
}
```

The function enter_parameters now shows the stored parameters and its values. The administrator now may change the value, or accept it by pressing *Return*.

The last step in this part is to store the new values back into the configuration file, if anything has changed. Underscores within the comments are replaced by spaces. The previous 10 versions of the configuration file are saved as files ending with .cfg.0 to .cfg.9, with the oldest configuration in the file ending with .cfg.9. The actual version has the extension .cfg.

Part V – execute configuration

The functions in part V now configure the queue manager. The 'core' function of this part is execute_qmgr_creation:

```
#
# Execute the queue manager configuration.
#

function execute_qmgr_configuration
{
   ret=0

   log_msg "S"
   log_msg "M" "Starting the queue manager configuration..."

   ans=""
   read_answer "   Start now" ans

   if [ "$ans" = "n" ]
   then
      log_msg "I" "Queue manager configuration cancelled by user!"

      # Finish the script logging and exit the program.
      log_finish 0
1179:    fi
1180:
1181:    # Suppress the usage of CTRL-C and other interrupts.
1182:    trap '' HUP INT ERR TERM
1183:
1184:    log_msg "S"
1185:    log_msg "M" "Setting up queue manager $QMGR_NAME (takes a
while, output follows) ..."
1186:
1187:    # If the queue manager does not exist ...
1188:    if [ $QMGR_EXISTS = "n" ]
1189:    then
1190:       # ... create the new queue manager.
1191:       create_qmgr
1192:       ret=$?
1193:    fi
1194:
1195:    if [ $ret -eq 0 ]
1196:    then
1197:       # Start the queue manager.
1198:       start_qmgr
1199:       ret=$?
1200:
1201:       if [ $ret -eq 0 ]
1202:       then
1203:          # Configure the queue manager objects.
1204:          configure_qmgr
1205:       fi
1206:
1207:       # End the queue manager.
1208:       end_qmgr
1209:    fi
1210:
1211:    return $ret
1212: }
```

First this function checks whether the queue manager exists (line 1188). If no, the queue manager will be created (call create_qmgr in line 1191). If the queue manager did exist before or has been successfully created, the function tries to start the queue manager by calling the function start_qmgr (line 1198). This function first checks whether the queue manager is already active. If so, the parameter STOP_QMGR is set to NO, otherwise it is set to YES and the queue manager will be started.

Now the function configure_qmgr is called (line 1204), and it reads the created definition file into the queue manager. Afterwards the queue manager will be stopped, if the parameter STOP_QMGR is set to YES.

Part VI – creating sample files

The last part of mkqmgr consists of functions to create some sample scripts and a README file. These files are stored in a directory, which has a name like the created queue manager, but ending with .postinstall. The README file describes further steps to configure an automatic start and stop script and a channel listening via inetd.

Perform the following steps as user 'root':

1. Add the contents of the file *TESTQM.postinstall/mqs_start_script* to your WebSphere MQ start-up script (eg */etc/rc2.d/S98mqm*).
2. Add the contents of the file *TESTQM.postinstall/mqs_stop_script* to your WebSphere MQ stop script (eg */etc/rc0.d/K18mqm*).
3. Add the contents of the file *TESTQM.postinstall/etc_services* to */etc/services*.
4. Add the contents of the file *TESTQM.postinstall/etc_inetdconf* to */etc/inetd.conf*.
5. Re-initialize inetd with:

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

The steps above require root privileges, so they have to be performed by a Unix administrator, whereas the other steps described before require only mqm privileges. If a WMQ administrator is not able to get root privileges, he or she can provide the system administrator with these files. When the queue manager configuration is executed by software distribution, these steps have to be included in a postinstall script.

The lines below show the postinstall files created my mkqmgr for a Sun Solaris system.

Start-up script:

```
if [ -e /opt/mqm/bin/strmqm ]
then
   echo "starting WebSphere MQ daemons"
   /bin/su - mqm -c "/opt/mqm/bin/strmqm TESTQM"
   /bin/su - mqm -c "/opt/mqm/bin/strmqcsv TESTQM"
fi
```

Stop script:

```
if [ -e /opt/mqm/bin/endmqm ]
then
   echo "stopping WebSphere MQ daemons"
   /bin/su - mqm -c "/opt/mqm/bin/endmqm -i TESTQM"
fi
```

Entry for */etc/services*:

```
WMQ1418      1418/tcp      # WMQ channel listener for qmgr TESTQM
```

Entry for */etc/inetd.conf*:

```
WMQ1418      stream      tcp      nowait      mqm      /opt/mqm/bin/
amqcrsta amqcrsta -m TESTQM
```

MAIN part

The ‘main’ part of mkqmgr consists of only a few lines, which call the script parts I to VI described above:

```
######################################################################
#
# MAIN part:
#
# Calling the other script parts described above.
#
######################################################################

# Initialize the script environment.
initialise $0 $*

# Start the script logging.
log_start

# Prepare the qmgr configuration.
prepare_qmgr_configuration

# Execute the qmgr configuration.
execute_qmgr_configuration
```

```
ret=$?

# Create some system configuration and README files.
[ $ret -eq 0 ] && create_sysfiles

# Finish the script logging and exit the program.
log_finish 0
```

DESCRIPTION OF THE FILES

The script mkqmgr uses several specific file extensions for its task. The name of the template file should point to the application. All other file names contain the queue manager with a specific ending. In the samples below an application named APPL (perhaps a shortcut) and a queue manager TESTQM are assumed:

- APPL.tpl – template file for an application APPL.
- APPL.TESTQM.cfg – actual configuration file of the application APPL for the queue manager TESTQM.
- APPL.TESTQM.cfg.N (*N* = 0 to 9) – previous versions of the configuration file.
- APPL.TESTQM.def – created definition file of the application APPL for the queue manager TESTQM.
- TESTQM.log – log file of the script mkqmgr.
- TESTQM.postinstall – directory containing the steps to be performed by a Unix administrator.

The directory TESTQM.postinstall contains several files, which describe further tasks to be performed by a system administrator. The file README.txt lists the next steps whereas the other files contain sample lines, which have to be added to some system files.

Listing of mkqmgr

The full listing for the script mkqmgr is too long to be written here. The script and a sample template and configuration file may be downloaded from Xephon's Web site at www.xephon.com/extras/uqm.zip. Parts of the files are shown in the text above.

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

List all the queues in the queue manager

Here is a Java application that will list all the queues in the queue manager. This supports wildcard usage for queue name and it will also list the system queues on request. Additionally, the program will display the queue type and the current depth of the queue.

```
/*********************************************************************/
/* Program name: ListAllQs                                          */
/* mail Id: balaji_srajan@yahoo.com                                 */
/*********************************************************************/
/* Function:                                                        */
/*      This program list all the queues in the queue manager. This */
/*      supports wildcard usage for queue name and will also list the */
/*      system queues upon request                                  */
/*      The program will also display the queue type and the current */
/*      depth of the queue                                          */
/*                                                                  */
/* Command line arguments:                                          */
/*       java ListAllQs - <-m queue manager name >                  */
/*<-q queuetype (local,remote,alias,model)>*/
/*                            <-w wildcard (use # instead of *, ABC#)> */
/*                            <-s system queues (Y/N)>              */
/*      - This application can be executed on the MQ Server         */
/*        environment. Pls make sure that the MQ jar files are      */
/*      in the class path (com.ibm.mq.jar, com.ibm.mq.pcf.jar)      */
/*                                                                  */
/*      Sample command line parameter:                              */
/*      ListAllQs -m QMGR -q LOCAL -w ABC# -s N                     */
/*****************+***************************************************/
trevore@xephon.com
import java.io.*;
import java.util.*;
import com.ibm.mq.*;
import com.ibm.mq.pcf.*;
```

```
public class ListAllQs
{
      private String              qmgrName;
                                    // for storing the Queue manager name
      private String              systemqueues;    // options
      private String              queueType;       // queue types
      private String              wildcard;        // for wildcard search
      public static void main (String arg[])
      {
            try
            {
                  if ( arg.length != 8)
                  {
        System.out.println("Please enter the arguments in,  <-m queue
manager name > <-q queuetype(use # instead of *)> <-w wildcard (use #
instead of *)> <-s system queues (Y/N)> \n" );
                        System.exit(1);
                  }
                  ListAllQs listallqueue = new ListAllQs();
                  listallqueue.Init(arg);
            }      // end of try
            catch (Exception exp)
            {
                  System.out.println("error in main.....");
                  exp.printStackTrace();
            }
      }      // end of main
      private void Init(String[] arg1)
      {
            try
            {
                  Hashtable params = new Hashtable(4);
                  if (arg1.length > 0 && (arg1.length % 2) == 0)
                  {
                        for (int i = 0; i < arg1.length; i+=2)
                        {
                              params.put(arg1[i], arg1[i+1]);
                        }      //end of for
   if (!(params.containsKey("-m") &&      params.containsKey("-q") &&
              params.containsKey("-w") && params.containsKey("-s") ) )
                        {
                              throw new IllegalArgumentException() ;
                        }
                  }
                  qmgrName             = (String) (params.get("-m"));
                  queueType            = (String) (params.get("-q"));
                  wildcard           = (String) (params.get("-w"));
                  systemqueues      = (String) (params.get("-s"));
                  wildcard = wildcard.replace('#','*');
                  queueType = queueType.replace('#','*');
```

```
System.out.println("-------------------------------------------------------
------");
     System.out.println("Queue manager Name            : " + qmgrName );
System.out.println("System queues (Y/N)           : " + systemqueues );
      System.out.println("Queue type                   : " + queueType );
     System.out.println("Wildcard (use # for all)      : " + wildcard );
System.out.println("-------------------------------------------------------
-----");
                  queueList();
            }      //end of try - Init
            catch (IllegalArgumentException IglExp)
            {
       System.out.println("  <-m queue manager name > <-q queuetype> <-w
wildcard (use # instead of *)> <-s system queues (Y/N)>");
                  System.exit(1);
            }
            catch(Exception exp)
            {
                  System.out.println("error in Init...");
                  exp.printStackTrace();
            }
      }      //end of Init
      private void queueList()
      {
            try
            {
           PCFMessageAgent       pcfAgent;               // PCF agent
           PCFMessage            pcfRequest;             // PCF request
           PCFMessage            pcfResponse[];          // PCF response
                  System.out.println("qmgrName..." + qmgrName);
                  pcfAgent = new PCFMessageAgent (qmgrName);
           System.out.println ("Queue Manager connect successfull... ");
                  System.out.println ("Building PCF request....");
                  pcfRequest = new PCFMessage (CMQCFC.MQCMD_INQUIRE_Q);
                  pcfRequest.addParameter (CMQC.MQCA_Q_NAME, wildcard);
                  if  ( (queueType.compareToIgnoreCase("*") == 0) )
                  {
                        System.out.println("in all queue type....");
               pcfRequest.addParameter (CMQC.MQIA_Q_TYPE, CMQC.MQQT_ALL);
                  }
              else if  ( (queueType.compareToIgnoreCase("local") == 0) )
                  {
                        System.out.println("in queue type local....");
             pcfRequest.addParameter (CMQC.MQIA_Q_TYPE, CMQC.MQQT_LOCAL);
                  }
             else if  ( (queueType.compareToIgnoreCase("remote") == 0) )
                  {
                        System.out.println("in queue type remote....");
           pcfRequest.addParameter (CMQC.MQIA_Q_TYPE, CMQC.MQQT_REMOTE);
                  }
              else if  ( (queueType.compareToIgnoreCase("alias") == 0) )
                  {
                        System.out.println("in queue type alias....");
            pcfRequest.addParameter (CMQC.MQIA_Q_TYPE, CMQC.MQQT_ALIAS);
                  }
              else if  ( (queueType.compareToIgnoreCase("model") == 0) )
                  {
                        System.out.println("in queue type model....");
            pcfRequest.addParameter (CMQC.MQIA_Q_TYPE, CMQC.MQQT_MODEL);
                  }
                  else
                  {
System.out.println("Invalid queue type, displaying all the queues....");
                  }
                  System.out.println ("requesting... ");
                  pcfResponse = pcfAgent.send(pcfRequest);
                  System.out.println ("got the response ... ");
                  System.out.println ("displaying queues name.....\n");
                  displayPCFAttributes(pcfResponse);
            }      // end of try
            catch (PCFException pcfExp)
            {
                  System.out.println ( "PCF exception...");
                 System.out.println ( "CC  : " + pcfExp.completionCode);
                  System.out.println ( "RC  : " + pcfExp.reasonCode);
                  System.out.println (pcfExp.exceptionSource);
            }  // end of PCFException catch
            catch (MQException mqExp)
            {
                  System.out.println("MQ Exception ....");
                  System.out.println ( "CC  : " + mqExp.completionCode);
                  System.out.println ( "RC  : " + mqExp.reasonCode);
            }      //end of MQException
            catch (IOException ioExp)
            {
                  System.out.println("IO exception ....");
            }
            catch(Exception exp)
            {
                  System.out.println("error in queue list...");
                  exp.printStackTrace();
            }
      }      //end of queueList
      private void displayPCFAttributes(PCFMessage[] pcfResp ) throws
PCFException
      {
            try
            {
                  int         qType;            // queue type
                  int         qDepth;           // for current depth
                  String      qName;            // for queue name
System.out.println("Queue name              QType      QDepth      \n");
                  for (int i =0; i < pcfResp.length; i++)
                  {
                        qDepth=0;
                        qName = "";
         qName = pcfResp [i].getStringParameterValue (CMQC.MQCA_Q_NAME);
                        qType = ((Integer)
(pcfResp[i].getParameterValue(CMQC.MQIA_Q_TYPE))).intValue();
                        switch (qType)
                        {
                        case 1:            // local queues
                        if ( systemqueues.compareToIgnoreCase("N") ==0 )
                              {
                                    if (!
(qName.substring(0,6).equals("SYSTEM")))
                                    {
   qDepth = pcfResp[i].getIntParameterValue (CMQC.MQIA_CURRENT_Q_DEPTH);

System.out.println(pcfResp[i].getStringParameterValue(CMQC.MQCA_Q_NAME)
                                     +   " \t" +      queueType(qType) +
   " \t" + pcfResp[i].getIntParameterValue(CMQC.MQIA_CURRENT_Q_DEPTH)) ;
                                    }
                              }
                              else
                              {
   qDepth = pcfResp[i].getIntParameterValue (CMQC.MQIA_CURRENT_Q_DEPTH);
System.out.println(pcfResp[i].getStringParameterValue(CMQC.MQCA_Q_NAME)
                                     +  " \t" +      queueType(qType)  +
   " \t" + pcfResp[i].getIntParameterValue(CMQC.MQIA_CURRENT_Q_DEPTH)) ;
                              }
                              break;
                        case 6:            // remote queues
                        if ( systemqueues.compareToIgnoreCase("N") ==0 )
                              {
                          if (! (qName.substring(0,6).equals("SYSTEM")))
                                    {
System.out.println(pcfResp[i].getStringParameterValue(CMQC.MQCA_Q_NAME)
                                 +    " \t" +      queueType(qType))   ;
                                    }
                              }
                              else
                              {
System.out.println(pcfResp[i].getStringParameterValue(CMQC.MQCA_Q_NAME)
                                +    " \t" +      queueType(qType))   ;
                              }
                              break;
                        case 3:            // alias queues
                        if ( systemqueues.compareToIgnoreCase("N") ==0 )
                              {
                                    if (!
(qName.substring(Ø,6).equals("SYSTEM")))
                                    {
System.out.println(pcfResp[i].getStringParameterValue(CMQC.MQCA_Q_NAME)
                                  +    " \t" +      queueType(qType) ) ;
                                    }
                              }
                              else
                              {
System.out.println(pcfResp[i].getStringParameterValue(CMQC.MQCA_Q_NAME)
                               +     " \t" +      queueType(qType))   ;
                              }
                              break;
                        case 2:            // model queues
                        if ( systemqueues.compareToIgnoreCase("N") ==Ø )
                              {
                        if (! (qName.substring(Ø,6).equals("SYSTEM")))
                                    {
System.out.println(pcfResp[i].getStringParameterValue(CMQC.MQCA_Q_NAME)
                                 +    " \t" +      queueType(qType) )  ;
                                    }
                              }
                              else
                              {
System.out.println(pcfResp[i].getStringParameterValue(CMQC.MQCA_Q_NAME)
                                +    " \t" +      queueType(qType))   ;
                              }
                              break;
                        default:      //
                        if ( systemqueues.compareToIgnoreCase("N") ==Ø )
                              {
                          if (! (qName.substring(Ø,6).equals("SYSTEM")))
                                    {
System.out.println(pcfResp[i].getStringParameterValue(CMQC.MQCA_Q_NAME)
                                  +    " \t" +      queueType(qType))  ;
                                    }
                              }
                              else
                              {
System.out.println(pcfResp[i].getStringParameterValue(CMQC.MQCA_Q_NAME)
                               +     " \t" +      queueType(qType))   ;
                              }
                        }      // end of switch
                  }      // end of for loop
            }
            catch ( PCFException pcfExp)
            {
       System.out.println ( "PCF exception in displayPCFAttributes...");
                 System.out.println ( "CC  : " + pcfExp.completionCode);
                  System.out.println ( "RC  : " + pcfExp.reasonCode);
```

```
                System.out.println (pcfExp.exceptionSource);
            }
        } // end of displayPCFAttributes
        private String queueType(int qTypeInt)
        {
            try
            {
                //Queue Types
                if ( qTypeInt == 1 )
                {
                    return "Local" ;
                }
                else if (qTypeInt == 3)
                {
                    return "Alias";
                }
                else if (qTypeInt == 6)
                {
                    return "Remote";
                }
                else if (qTypeInt == 2)
                {
                    return "Model";
                }
                //return "qType not supported";
            }
            catch (Exception e)
            {
                System.out.println("Error in qType function  ");
                e.printStackTrace();
            }
            return "qType not supported";
        }      //end of queueType
        private void readPropertyFile(String fileName) throws Exception
        {
            try
            {
                System.out.println("Reading from the file name....");
                Properties mqProperties = new Properties();
            mqProperties.load(getClass().getResourceAsStream(fileName));
             qmgrName           = mqProperties.getProperty("qmgrname");
           systemqueues      = mqProperties.getProperty("systemqueues");
           queueType           = mqProperties.getProperty("queuetype");
             wildcard           = mqProperties.getProperty("wildcard");
            }
            catch (Exception exp)
            {
           System.out.println("Error in reading the property file ..." +
fileName );
                exp.printStackTrace();
```

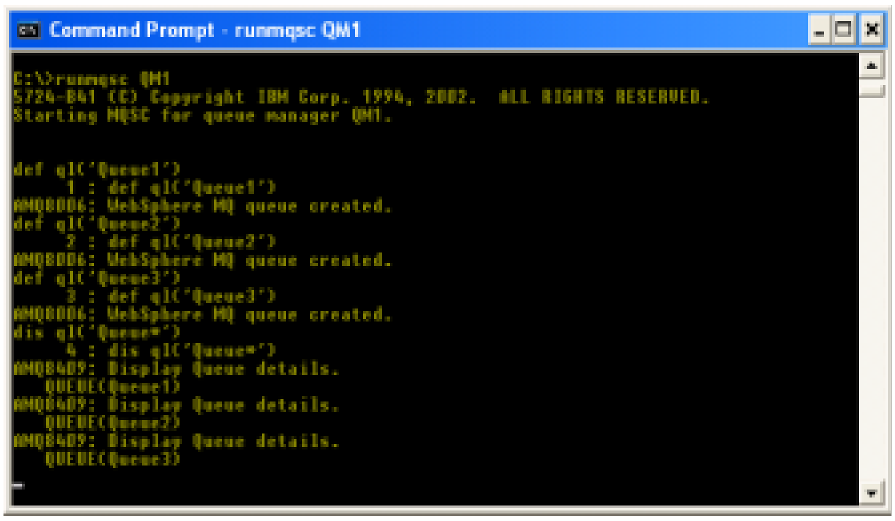
```
                        System.exit(Ø);
                }
        }
}
```

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

Using WebSphere MQ as a JNDI repository for JMS administrable objects – part 2

This month we conclude the article looking at using WMQ as a JNDI repository for JMS administrable objects.

ME01 can be used to create real objects on the messaging transport as well. For example, if you were to bind() a queue type destination into the namespace then a real underlying WMQ queue will be created with the name that you supply. You would then, of course, be able to perform a lookup() on that name and ME01 will again generate a queue type destination for you to use. Note that any additional properties you set on the queue that was



```
C:\>runmqsc QM1
5724-B41 (C) Copyright IBM Corp. 1994, 2002.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager QM1.


def ql('Queue1')
     1 : def ql('Queue1')
AMQ8006: WebSphere MQ queue created.
def ql('Queue2')
     2 : def ql('Queue2')
AMQ8006: WebSphere MQ queue created.
def ql('Queue3')
     3 : def ql('Queue3')
AMQ8006: WebSphere MQ queue created.
dis ql('Queue*')
     4 : dis ql('Queue*')
AMQ8409: Display Queue details.
   QUEUE(Queue1)
AMQ8409: Display Queue details.
   QUEUE(Queue2)
AMQ8409: Display Queue details.
   QUEUE(Queue3)
_
```

Figure 1: Example screen

bound will be lost in basic mode. Also in basic mode, you will be unable to bind in objects of any type except WMQ JMS Queue destination objects.

This can easily be seen by using the JMSAdmin program with ME01. See the *Set-up* section below for information on how to set up JMSAdmin to use ME01. If I were to create a queue manager called QM1, start it, start the command server and define my three queues, I would see something similar to Figure 1.

Without performing any JMS administration, I can now start the JMSAdmin tool and I will see Figure 2.

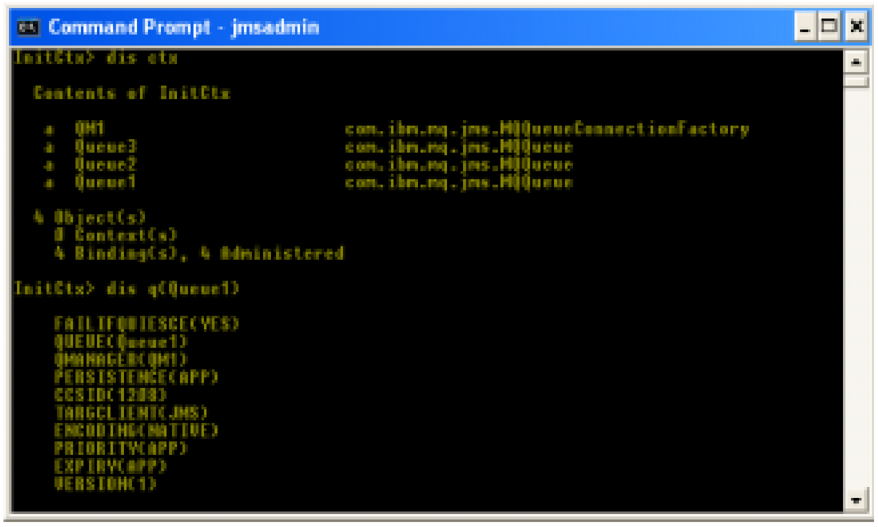
Note that ME01 will display only non-temporary local queues whose names do not start with 'SYSTEM.'.

In advanced mode, ME01 can do everything that it can do in basic mode, but it also boasts the ability to allow you to change the JMS properties of the administrable objects and persist them as well as allowing you to bind any other type of object (such as a topic type destination for example). ME01 does this by storing a serialized copy of the administrable object on a special queue. This means that, if you delete your queue manager, you will lose any modified definitions of administrable objects that ME01 had stored for you.

ME01 will still allow you to create real WMQ queues in advanced mode by binding in a WMQ JMS Queue administrable object. The only difference being that in advanced mode, all the properties on the object will persist and the default ones will not be used.

HOW DOES ME01 WORK?

At the heart of ME01 is the use of Programmable Control Format (PCF) messages. PCF messages are used to administer and monitor a WMQ queue manager programmatically. This is how ME01 can gather information on what queues you have available. Note that because of this, you will need to have the command server started on your queue manager for ME01 to function correctly.



```
InitCtx> dis ctx

  Contents of InitCtx

   a  QM1                           com.ibm.mq.jms.MQQueueConnectionFactory
   a  Queue3                        com.ibm.mq.jms.MQQueue
   a  Queue2                        com.ibm.mq.jms.MQQueue
   a  Queue1                        com.ibm.mq.jms.MQQueue

  4 Object(s)
    0 Context(s)
    4 Binding(s), 4 Administered

InitCtx> dis q(Queue1)

    FAILIFQUIESCE(YES)
    QUEUE(Queue1)
    QMANAGER(QM1)
    PERSISTENCE(APP)
    CCSID(1208)
    TARGCLIENT(JMS)
    ENCODING(NATIVE)
    PRIORITY(APP)
    EXPIRY(APP)
    VERSION(1)
```

Figure 2: JMSAdmin tool

In basic mode, ME01 maps the common JNDI commands to the following WMQ operations:

- bind() – if a WMQ JMS Queue object is passed in, ME01 will send a PCF command to the queue manager asking it to create a local queue. Any other object type will be rejected.
- lookup() – if the name passed into lookup() matches the name of the queue manager, ME01 will generate a QueueConnectionFactory providing a connection to the queue manager, and will return it. Otherwise, ME01 sends a PCF message to the queue manager asking for specific properties about the named queue. If the queue exists, ME01 will construct a WMQ JMS Queue object and return it. Otherwise, a NamingException will be thrown.
- list() – the queue manager will be asked for a list of all the non-temporary non-system queues, and these will be returned along with a QueueConnectionFactory definition for the queue manager.

In advanced mode, things are slightly more complicated:

- bind() – the object passed in will be serialized and stored on the ME01 admin queue. If a WMQ JMS Queue object is passed in, ME01 will also send a PCF command to the queue manager asking it to create a local queue.

- lookup() – if the name passed into lookup() matches the name of the queue manager, ME01 will generate a QueueConnectionFactory providing a connection to the queue manager, and will return it. Otherwise, ME01 will look for a matching object on its admin queue. If one is found it is returned; otherwise a PCF message is sent to the queue manager asking for specific properties about the named object. If a queue exists with that name, ME01 will construct a WMQ JMS Queue object and return it. Otherwise, a NamingException will be thrown.
- list() – the queue manager will be asked for a list of all the messages on the ME01 admin queue and will then add (if not already on the list) any non-temporary non-system queues, and these will be returned along with a QueueConnectionFactory definition for the queue manager.

EXAMPLES OF USE

If you are unfamiliar with JNDI there is an excellent tutorial from Sun, which can be found at http://java.sun.com/products/jndi/tutorial. This explains not only how to use JNDI in a Java program, but it also gives details of how to implement the JNDI interface to become a name service provider. Essentially, ME01 is just another JNDI implementation.

Below are some examples of how to use the JNDI interface with ME01. To get these examples working, refer to the *Set-up* section below.

This first example shows how a Java application can use ME01 to do a simple bind and then look-up on a queue manager of a queue object. This can be run on a queue manager with no prior administration needed. The result will be that a real WMQ queue will be created and JMS programs can perform look-ups on it.

```
import java.util.Hashtable;
import javax.jms.JMSException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import com.ibm.mq.jms.MQQueue;
/**
 * Simple Demo to show how MEØ1 and JNDI can be used to create a WMQ
 * queue and do a look-up to get a JMS administrable object back.
 *
 * @author Gareth Matthews
 */
public class MEØ1Demo1
{
   public static void main (String[] args)
   {
      MEØ1Demo1 demo = new MEØ1Demo1();
      demo.go();
   }
   public void go()
   {
      // First set up the properties of the JNDI connection
      Hashtable props = new Hashtable();
      props.put(Context.INITIAL_CONTEXT_FACTORY,
                             // This is what kind of repository we want
                "com.ibm.mq.jms.context.WMQInitialContextFactory");
// MEØ1
      props.put(Context.PROVIDER_URL,
                            // This is how to connect to the repository
                "QM1");                    // Bindings connect to 'QM1'

      try
      {
         // First connect to the queue manager
         System.out.print("Connecting....");
         Context qmContext = new InitialContext(props);
         System.out.println("done");
         // Now programatically create an administrable object to bind
         // into the queue manager that points to a called called QUEUE1
         MQQueue mqQueueAdminObj = new MQQueue("QUEUE1");
// Now bind in the new queue. Note that if we give the queue a different
// name here (on the bind call), then this will overwrite the queue name
         // we just set in the administrable object above.
  // This will now cause a real WMQ queue called 'QUEUE1' to be created.
         System.out.print("Binding admin object...");
         qmContext.bind("QUEUE1", mqQueueAdminObj);
         System.out.println("done");
         // Now try and look up the administrable object
         System.out.print("Looking up...");
        MQQueue lookedUpAdminObj = (MQQueue) qmContext.lookup("QUEUE1");
```

```
            System.out.println("done");
            // Lets see if the objects are equal - they should be
            if (lookedUpAdminObj.equals(mqQueueAdminObj))
            {
               System.out.println("** The objects were equal");
            }
      // Always make sure we shut down the connection to the queue manager
            System.out.print("Closing connection...");
            qmContext.close();
            System.out.println("done");
         }
         catch (JMSException je)
         {
            // Can be thrown when we programatically create the MQQueue
            System.out.println();
            System.err.println("Caught a JMS exception:");
            je.printStackTrace();
         }
         catch (NamingException ne)
         {
            // Can be thrown if the connect, bind, lookup or close fail.
            System.out.println();
            System.err.println("Caught a naming exception:");
            ne.printStackTrace();
         }
   }
}
```

So, this program first of all makes a bindings connect to the queue manager. As such the queue manager must be on the same machine as the program being run. Making ME01 run over a client connection is very simple though, and can be done by changing the provider URL. The syntax is 'hostname:port/ SVRCONN channel name'. For example:

```
props.put(Context.PROVIDER_URL, "myqm.mycompany.com:1414/
MY.SVRCONN.CHL");
```

will connect to the queue manager running on the host *myqm.mycompany.com* on port *1414*, using the SVRCONN channel called *MY.SVRCONN.CHL*. You will need a listener running on the specified port for that to work. See the *Set-up* section below for more information.

Then we connect to the queue manager by creating a new InitialContext and passing the properties. We then programmatically create an MQQueue administrable object and bind it into the context. This:

- Creates a real WMQ queue called ‘QUEUE1’.
- Updates the queue name in the administrable object to be the same as the actual WMQ queue.
- Saves the administrable object on the admin queue.

Then the look-up is done. Typically this will be the only JNDI operation that will be done in your JMS application and you will get returned the corresponding administrable object.

Note: if you create a WMQ queue directly (with runmqsc, for example), ME01 will dynamically generate an administrable object when you do a look-up on the queue name. If you create it by binding a JMS MQ Queue with JMSAdmin or by using this test program, ME01 will save the administrable object on the admin queue and so no dynamic generation willbe performed.

Observe what happens if this program is run twice, though – when we try the bind() again, you see ‘javax.naming. NameAlreadyBoundException: The alias is already bound to another object’ because the queue already exists. Now, using a tool such as runmqsc or the MQ Explorer, delete the queue called QUEUE1 that ME01 created. Running the program will still produce exactly the same results – this is because while we have deleted the queue that ME01 dynamically created for us, the administrable object is still saved on the admin queue. As such, lookup() will still work and bind() will still fail. To completely remove the object you should modify your program to remove the object by using the JNDI unbind() call. You could also use the JMSAdmin program to do this.

If this program were used with ME01 in basic mode, the behaviour would be slightly different. Of course, no administrable properties set on our object would persist. But if after running the program we delete the underlying queue, the program could be run straight away because no administrable object was stored on the admin queue. ME01 can be put into basic mode in two ways:

- Add the following item in the properties hashtable passed to new InitialContext:

  ```
  props.put("MQCONTEXTMODE", "BASIC");
  ```

- Add a Java system property MQCONTEXTMODE with the value ‘BASIC’. This can be done on the Java command line. For example:

  ```
  java -DMQCONTEXTMODE=BASIC
  ```

Let’s look at another piece of code that demonstrates how to use the list() call. This example puts ME01 into basic mode and asks it for a list of all the objects it knows about. You should find you get returned a list of all the non-system non-temporary queues, and a QueueConnectionFactory that represents the queue manager.

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NameClassPair;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;
/**
 * Simple Demo to show how ME01 in basic mode can be used to display a
 * list of all the non-system non-temporary queues on a queue manager.
 *
 * @author Gareth Matthews
 */
public class ME01Demo2
{
   public static void main (String[] args)
   {
      ME01Demo2 demo = new ME01Demo2();
      demo.go();
   }
   public void go()
   {
      // First set up the properties of the JNDI connection
      Hashtable props = new Hashtable();
      props.put(Context.INITIAL_CONTEXT_FACTORY,
                          // This is what kind of repository we want
                "com.ibm.mq.jms.context.WMQInitialContextFactory");
// ME01
      props.put(Context.PROVIDER_URL,
                          // This is how to connect to the repository
                "QM1");                  // Bindings connect to 'QM1'
```

```
      props.put("MQCONTEXTMODE",        // Specifies the WMQContext mode
                "BASIC");
// Basic mode
      try
      {
         // First connect to the queue manager
         System.out.print("Connecting....");
         Context qmContext = new InitialContext(props);
         System.out.println("done");
         // Now ask the queue manager for its list of queues
         NamingEnumeration enum = qmContext.list("");
         while (enum.hasMoreElements())
         {
            NameClassPair ncp = (NameClassPair) enum.nextElement();
          System.out.println(" - Got object of name: " + ncp.getName() +
                             " - Class: " + ncp.getClassName());
         }
     // Always make sure we shut down the connection to the queue manager
         System.out.print("Closing connection...");
         qmContext.close();
         System.out.println("done");
      }
      catch (NamingException ne)
      {
         // Can be thrown if the connect, list or close fail.
         System.out.println();
         System.err.println("Caught a naming exception:");
         ne.printStackTrace();
      }
   }
}
```

Note that if you get an object returned in the NamingEnumeration then you can use lookup() to get this object back. An object that appears here will never cause a NameNotFoundException to occur on lookup().

This last code snippet shows how to use ME01 in your JMS application. This simple program will put and get a message from a queue called QUEUE1 on queue manager QM1. Note that this queue will need to exist before this program is run:

```
import java.util.Hashtable;
import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueReceiver;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
/**
 * This demo shows how you can use MEØ1 in the context of a real JMS
 *  application.
 *
 * @author Gareth Matthews
 */
public class MEØ1Demo3
{
   public static void main(String[] args)
   {
      MEØ1Demo3 demo = new MEØ1Demo3();
      demo.go();
   }
   public void go()
   {
      // First set up the properties of the JNDI connection
      Hashtable props = new Hashtable();
      props.put(Context.INITIAL_CONTEXT_FACTORY,
                            // This is what kind of repository we want
                "com.ibm.mq.jms.context.WMQInitialContextFactory");
// MEØ1
      props.put(Context.PROVIDER_URL,
                             // This is how to connect to the repository
                "QM1");                    // Bindings connect to 'QM1'
      try
      {
         // First connect to the queue manager
         System.out.print("Connecting....");
         Context qmContext = new InitialContext(props);
         System.out.println("done");
  // Now lookup the connection factory and destination that we will need
         System.out.print("Performing lookups...");
         QueueConnectionFactory qcf = (QueueConnectionFactory)
qmContext.lookup("QM1");
         Queue q = (Queue) qmContext.lookup("QUEUE1");
         System.out.println("done");
         // Now connect to the queue manager JMS style
         System.out.print("Creating connection...");
         QueueConnection conn = qcf.createQueueConnection();
         QueueSession sess = conn.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);
         QueueSender sender = sess.createSender(q);
         QueueReceiver receiver = sess.createReceiver(q);
         conn.start();
         System.out.println("done");
         // Now send a message
         System.out.print("Sending message...");
      TextMessage mess = sess.createTextMessage("Hello from MEØ1Demo3");
         sender.send(mess);
         System.out.println("done");
         // And get it back
         System.out.print("Receiving message...");
         TextMessage rcvMess = (TextMessage) receiver.receiveNoWait();
         System.out.println("done");
         if (rcvMess != null) System.out.println("Message text was: " +
rcvMess.getText());
         else     System.out.println("*** No message was received ***");
         // Now close down
         System.out.print("Closing down...");
         sender.close();
         sess.close();
         conn.close();
         qmContext.close();
         System.out.println("done");
      }
      catch (JMSException je)
      {
         // Can be thrown by a JMS call
         System.out.println();
         System.err.println("Caught a JMS exception:");
         je.printStackTrace();
      }
      catch (NamingException ne)
      {
         // Can be thrown if the connect, lookup or close fail.
         System.out.println();
         System.err.println("Caught a naming exception:");
         ne.printStackTrace();
      }
   }
}
```

OTHER JNDI CALLS

There are many other JNDI calls that can be used with ME01:

- rebind() – this call will replace an existing object. The call will fail in basic mode, but in advanced mode can be used to update the administrable object properties.
- unbind() – this call will remove an object. In basic mode just

the WMQ queue is deleted, but in advanced mode the administrable object is also removed. Note that a real WMQ queue cannot be deleted if it has messages on it. However, to override this protection you can set the Java system variable MQJMS_PURGE_ ON_DELETE. Setting this to 'YES' will cause the queue to be deleted with the purge option set.

- rename() – this call will rename an object. If the object is a queue then the WMQ queue is renamed along with the base queue name in the administrable object.

Note that all other calls will fail with an OperationNotSupportedException.

SET-UP

For ME01 to work with a queue manager it needs to be started and the command server needs to have been started (use the command **strmqcsv**). If you are connecting to a remote queue manager, it needs a listener running and a suitable SVRCONN channel defined.

To use ME01 with JMSAdmin or another suitable application you need to ensure that the mqcontext.jar and com.ibm.pcf.jar (the MS0B Support Pac) is located in your classpath along with all the WMQ classes for Java class file that are located in the *Java\lib* directory of your WMQ installation.

Your JMSAdmin.config file should then be updated with the following information:

```
#  The following line specifies which JNDI service provider is in use.
#  It currently indicates an LDAP service provider. If a different
#  service provider is used, this line should be commented out and the
#  appropriate one should be uncommented.
#
INITIAL_CONTEXT_FACTORY=com.ibm.mq.jms.context.WMQInitialContextFactory
#
#  The following line specifies the URL of the service provider's
#  initial context. It currently refers to an LDAP root context.
#  Examples of a file system URL and WebSphere's JNDI namespace are
#  also shown, commented out.
```

```
#
PROVIDER_URL=QM1
```

Of course, the provider URL parameter should be changed to match the settings on your system and all other parameters should be commented out.

To compile the samples, simply copy and paste them into your favourite text editor and save them as the appropriate name (ie the first demo needs to be *ME01Demo1.java* etc). Then ensure that all the JAR files that come with the WMQ classes for Java, the ME01 Support Pac, and the MS0B Support Pac are located in your classpath. Then compile them by typing **javac *.java**.

Gareth Matthews
WebSphere Platform Messaging Developer
IBM (UK)

IGQ in practice

This article will explain how IGQ works in both theory and practice and is relevant to Versions 5.2 and 5.3 of WebSphere MQ for z/OS.

DEFINTION AND THEORY

IGQ stands for Intra Group Queueing, which was introduced with Version 5.2.

It forms part of the Shared Queue infrastructure (and hence is only available on z/OS) and allows messages targeted at remote or cluster queues (which are normally transferred between queue managers using traditional sender-receiver, server-requestor, or cluster channels) to use the coupling facility instead. It does this through a special shared transmission queue called SYSTEM.QSG.TRANSMIT.QUEUE.

As a shared queue, it is available to all queue managers that form part of the given QSG (Queue Sharing Group), and, as such, provides a cheaper way of transferring messages compared with using the channel initiator and the IP network. When a queue manager is started, the IGQ agent starts as well – see the following message:

```
CSQMØ5ØI ?QM1P CSQMIGQA Intra-group queuing agent starting, TCB=ØØ89581Ø
```

This agent starts regardless of whether IGQ is switched on or not.

Functionally, each agent (on each of the queue managers in the QSG) issues an MQGET on the shared queue with a CORRELID (correlation ID) equal to its queue manager name. To ensure good performance, an index based on CORRELID is defined on SYSTEM.QSG.TRANSMIT.QUEUE.

It must be noted, however, that the usual Shared Queue restrictions apply, namely:

- Messages must be less than or equal to 63KB (64,512 bytes) in length. This refers to the 'payload' only and includes the transmission header.
- The Coupling Facility has a finite size and has most probably been sized for a set of predetermined Shared Queues. The last thing an MQ administrator wants is to flood the CF with IGQ-eligible messages.

So what happens if a batch of messages destined for remote queue managers has a mix of sizes, some less than 63KB and some greater?

Fortunately, MQSeries is clever enough to detect this situation, and will, on a message-by-message basis, choose the most appropriate route. Those larger than 63KB will be sent via the traditional route using standard traditional channels for 'normal' queues, and cluster channels for cluster queues. Those less than or equal to 63KB go via the CF.

Unfortunately, there is currently no logic within MQSeries that will detect whether the CF is full or nearly full and redirect messages via the traditional route.

It would, however, be possible for an automated process to check the status of the CF and, if a structure is nearing its capacity and no further growth is possible, the IGQ could be (temporarily) switched off via the following command:

```
    ALTER QMGR IGQ(DISABLED)
```

What is the exact size at which point the IGQ route is chosen?

The answer lies in the way messages are routed between queue managers and this is the same for remote and cluster queues. Basically, MQ adds a ‘transmission’ header (XQH) at the front of the original message payload as shown here:

Message sent to a local queue:

```
|… MQMD 324 bytes …|… Message data …|
```

Message sent to a remote or cluster queue via a transmit queue:

```
|.. MQMD 324 bytes ..|.. XQH 1Ø4 bytes ..|.. ORIG MQMD 324 bytes ..|..
Message data ..|
```

Remember that the IGQ route goes via a Shared Transmission Queue, which is restricted to a payload size of 63KB (or 64,512 bytes).

Therefore, if the payload is less than or equal to (64,512-XQH header) = (64,512-104-324) = 64,084 bytes, the IGQ route is chosen.

Of course, all this is transparent to the application, and, unless you use the method shown later on (by checking the RESET QSTATS), the administrator wouldn’t know either.

Note that the usual cluster workload balancing takes place provided the DEFBIND flag is set to ‘N’, meaning that MQ will decide (on a message-by-message basis) where to place the message. Even if the IGQ route is chosen, it is the cluster workload exit that determines the target queue.

Could the IGQ route be chosen for administration commands, eg when using the TSO panels to list all local queues in the QSG? Internally it issues the equivalent of this command:

```
DIS QLOCAL(*) CMDSCOPE(*)
```

The CMDSCOPE(*) tells MQ to issue this command to all queue managers in the QSG, so how does it do that? Does it use the IGQ to place the command on the SYSTEM.COMMAND.QUEUE of each of the queue managers?

No, an internal method is used that is invisible to the administrator.

TESING TO 'PROVE' IGQ WORKS

Here, two queue managers exist called QM1 and QM2. Only high-level steps are shown, which are not exhaustive. Traffic is assumed to travel from QM2 to QM1.

- On QM2 define a sender channel to QM1, a transmit queue, and a remote queue.
- On QM1 define a receiver channel from QM2 and a local queue.
- On QM1 define a cluster receiver channel pointing at itself and a cluster queue.
- On QM2 define a cluster receiver channel pointing at itself and a cluster sender channel pointing at QM1.
- Ensure IGQ=Y on both queue managers. If the 'sending' queue manager does not have IGQ switched on, then all messages end up on the standard transmit queue.
- Check that the Shared Transmission Queue is defined, its CF structure is large enough and has been defined to automatically grow.
- Create some test data, for example 100 records with every odd record of size 64,085 and every even record 64,084.
- Issue **reset qstats(xxx) cmdscope(*)** on one of the queue managers in the QSG in order to reset the queue statistics counters where *xxx* are the queues:
  - SYSTEM.QSG.TRANSMIT.QUEUE

- SYSTEM.CLUSTER.TRANSMIT.QUEUE
- the transmit queue on QM2.

- Add the 100 records to the remote queue on QM2.
- Display the status of the standard channel.

The test shows that 50 messages were transmitted – as expected:

```
Channel name         Connection name
State
  Start time           Messages  Last message time   Type
Disposition
QM1.TO.QM2                                         CHANNEL   ALL
QSG1
QM1.TO.QM2         123.45.38.17Ø                                    RUN
  2ØØ4-Ø2-24 13.16.24  5Ø       2ØØ4-Ø2-24 13.16.28 RECEIVER  PRIVATE
QM2
QM1.TO.QM2         123.45.38.198                                    RUN
  2ØØ4-Ø2-24 13.16.24  5Ø       2ØØ4-Ø2-24 13.16.28 SENDER    PRIVATE
QM1
                  ******** End of list ********
```

- Display which processes have queue ‘open’.

  It shows the channel initiator, which would have been used for the traditional channel route, as well as ‘QM2 SYSTEM’, which denotes the IGQ route:

```
Queue name                                       Disposition   Access
  Application        ASID  Application information         User ID
QL*                                              ALL     QSG1
QLOCAL1                                  QMGR    QM2  O -  - -
  QM2     SYSTEM
QLOCAL1                                  QMGR    QM2  O -  - -
  QM2CHIN CHINIT    ØØ6B  QM1.TO.QM2                     QM2CHIN
                         123.45.38.17Ø
                 ******** End of list ********
```

- Issue the same ‘reset’ command as above.

  It shows that 50 messages were taken from the shared transmission queue and 100 messages added to the local queue:

```
QSTATS(SYSTEM.QSG.TRANSMIT.QUEUE)
QSGDISP(SHARED)
RESETINT(392)
```

```
HIQDEPTH(0)
MSGSIN(0)
MSGSOUT(50)

QSTATS(QLOCAL1)
QSGDISP(QMGR)
RESETINT(392)
HIQDEPTH(100)
MSGSIN(100)
MSGSOUT(0)
```

This same exercise can be repeated for cluster queues.

EFFECT ON DIFFERENT ROUTES

As with all MQ traffic, if there is more than one route to a given queue, the order is not maintained. This can be seen clearly using the same 100 records.

The checks showed that 50 messages went via the cluster channel route, and 50 went via the QSG route. It is also known that the QSG route uses less of the processor, so it was expected that the order would be different, but also that the messages ‘at the front’ of the target queue would be the ‘even’ numbered ones. The actual order was found to be:

```
Records 2-18, then Record 1
Records 20-24, then Record 3
Records 26-30, then Record 5
Records 32-36, then Record 7
Records 38-44, then Record 9
Records 46-52, then Record 11
Records 54-58, then Record 13
Records 60-70, then Record 15
Records 72-78, then Record 17
Records 80-84, then Record 19
Records 86-90, then Record 21
Records 92-96, then Record 23
Records 98-100, then Records 25-99
```

This showed that messages going via the QSG route completed leaving the cluster route still to process 38 messages, or, put another way, the QSG processed 50 and the cluster 12, making the QSG route four times faster. Care has to be taken, however, on how to define ‘faster’ because this was an isolated test.

In fact, the test was repeated with 1000 records, again with alternative sizes. This time, however, the result was a little different. The 500 messages sent via IGQ completed leaving the traditional route still to do 248. The ratio is 500:252, or roughly double the rate.

Another effect is that error situations can occur. Imagine that the above mixed file of 100 records came in a different order, eg 50 of the larger messages followed by 50 of the smaller ones. Delivering the first 50 may not pose a problem via the traditional channel route, but imagine the situation where the CF has a (temporary) space problem and code 2192 was returned to the application – an incomplete 'set' could be the result. If this set of records was processed as a batch – ie under syncpoint and a commit issued at the end – then there is no issue because all updates should be/could be backed out. As always, keep the number of updates within an LUW small (the IBM manual suggests <100).

The 2192 code can be returned in a busy system in which the time taken by the system to automatically increase the structure size is longer than the time MQ is prepared to wait.

What if the target queue was full or not defined? Normal rules apply and messages that could not be delivered would be stored in the Dead Letter Queue. If the DLQ was not defined or was full, then messages would remain on the transmit queue. For traditional channels that would mean the normal transmit queue, which uses the pagesets to store messages; and for those going via the QSG, messages would be stored in the CF with potentially serious results if the CF structure becomes full.

In fact, as is true with a normal transmission queue, if messages back up on the shared transmission queue, no further messages can be delivered to the target queue manager until the first problem (queue full) is resolved – even if messages are destined for a different queue on the same queue manager.

What if the shared transmission queue itself becomes full? In this instance the application gets a non-zero return code.

RECOMMENDATIONS

1 If the reduction of CPU costs is important, as well as faster delivery to the target queues, switch on IGQ but check the other recommendations. If possible, use SMF to measure the CPU cost using both traditional and IGQ routes.

2 Define a separate CF structure for queue SYSTEM.QSG.TRANSMIT.QUEUE (and possibly allow SYSTEM.QSG.CHANNEL.SYNCQ as well) and size it properly. Remember, however, that the IGQ is either 'on' or 'off' and is used for all remote and cluster queues whose payload is less than 64,084 bytes. If the majority of messages are below 64,084 then an assessment needs to be made of how risky the switchover to IGQ is going to be. I can't stress this too much: check what proportion of messages will go via the IGQ route.

   In fact, when figures were checked on the client system, it was discovered that between 550,000 and 650,000 messages were being sent via the cluster channel, amounting to 750–830MB per day. It was decided, because of storage constraints in the CF, to wait until the IGQ could be controlled at a more granular level, or more storage was added.

   An idea was sent to IBM to request an enhancement so that queues can be chosen by the administrator to be 'eligible' for IGQ in a similar way to CICS programs which have the 'eligible for LPA' flag. Whichever method is chosen, a more granular method of selecting IGQ is required.

3 If the order of messages sent in a 'batch' is important, and the sizes of the messages fall on both sides of 64,084 bytes, do not use IGQ; or use an alternative method to re-establish the original order.

4 Ensure that a Dead Letter Queue is defined to ensure that messages do not stack up on the IGQ transmit queue.

5 Review how applications issue updates when adding a batch of messages that could use both traditional and IGQ routes.

The number of updates within an LUW should be kept to a small number – the IBM recommendation is to issue a commit every 100 updates.

In fact, the MQ 'system' CF structure (called CSQ_ADMIN) is used to track LUWs, so care has to be taken with large LUWs. Here is an excerpt from the MP16 Support Pac:

CSQ_ADMIN usage is affected by the number of messages in each unit of work, but only for the duration of the commit for each UOW. This need only be a concern for extremely large UOWs as the minimum size structure is enough for a UOW of about 40000 messages. This is larger than the default maximum size UOW of 10000, defined by MAXUMSGS. The use of UOWs with very large numbers of messages is NOT recommended.

6 For all CF structures, define a monitoring process that detects whether a structure is nearing its maximum size. In addition, include specific processes to detect whether shared queues are near their maximum queue depth. Unless this is done, and the application does not send out an alert when it gets 2192 return codes (to MQ, this is just an application issue and so it does not send out any alerts), error situations will be missed.

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

MQ news

IBM has announced WebSphere Extended Deployment, which is designed to automatically optimize the performance of a company's software and hardware, on demand, particularly during unexpected spikes in usage or changing market conditions.

The product runs on WebSphere Application Server infrastructure software, and more efficiently utilizes, balances, and shares the workload among applications and application servers. It helps allow IT resources to adjust on-the-fly to the demands of critical business applications.

In concert with IBM Tivoli Intelligent Orchestrator, the new software monitors the efficiency of the network, constantly re-balancing and farming out unexpected workloads to underutilized hardware and software. Network managers have the option to manually confirm suggested optimizations before they take place.

It also partitions large jobs over many processors, databases, application software, and application servers.

For further information contact your local IBM representative.
URL: http://www.ibm.com/websphere.

* * *

Open Terra is partnering with Novell to deliver mobile and wireless business solutions to customers. Through this partnership, Open Terra will offer its mSolve Enterprise Mobility Suite solution using Novell technology to customers pursuing mobile business integration strategies.

Novell's SUSE LINUX and NetWare customers can now create mobile applications with Open Terra's mSolve Enterprise Mobility Suite. The mSolve suite provides a robust mobility framework for rapid application development and deployment that is always online, allowing for seamless integration to all major databases, Web services, MQ systems, EJB, and existing business applications and logic.

For further information contact:
Openterra, 20 Reddington Drive, Matawan, NJ 07747, USA.
Tel: (732) 765 9600.
URL: http://www.openterra.com/NewsAndEvents/Press_060904.jsp.

* * *

DataPower has announced firmware release Version 3.0 for its XML-aware networking hardware, ie DataPower XS40 XML Security Gateway and DataPower XA35 XML Accelerator.

Version 3.0 includes integration with Eclipse and WebSphere Studio Web services development environments, enhanced MQ support, and additional security for SOAP with Attachments (SwA).

For further information contact:
DataPower Technology, One Alewife Center, 4th Floor, Cambridge, MA 02140, USA.
Telephone: (617) 864 0455.
URL: http://www.datapower.com/newsroom/pr_060704_3dot0.html.

* * *