# 55

# MQ update

*January 2004*

## In this issue

# MQ Update


## Published by

Xephon
27-35 London Road
Newbury
Berkshire RG14 1JL
England
Telephone: 01635 38126
From USA: 01144 1635 38126
Fax: 01635 38345
E-mail: info@xephon.com

## North American office

Xephon/QNA
Post Office Box 350100
Westminster CO 80035-0100, USA
Telephone: (303) 410 9344
Fax: (303) 438 0290


## Disclaimer

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


## Editor

Madeleine Hudson
E-mail: MadeleineH@xephon.com


## Subscriptions and back-issues

A year's subscription to *MQ Update*, comprising twelve monthly issues, costs £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 £22.50 ($33.75) each including postage.

## Contributions

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

## *MQ Update* on-line

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




*Printed in England.*

# Interfacing WMQ with Web applications

## OVERVIEW

Web server platforms such as Active Server Pages (ASP) and ASP.Net have difficulty connecting to remote queue managers because their applications run under a local user-ID that can't access network resources. This article explores solutions to this problem.

## THE PROBLEM

I base my discussion on the ASP.Net platform; however, the same problems and solutions also apply to ASP Classic and many others.

ASP.Net applications and Web services all run under the ASPNET user-ID. This user-ID is local to the Web server machine and can belong to local security groups on that machine but not domain groups on the network. You can grant it access to local resources but not network resources. This has two implications for WMQ development:

- If you normally use a network-based channel table to locate queue managers a local user account cannot access it. When this happens your attempts to connect to a queue manager will return a reason code of 2058 – MQRC_QUEUE MANAGER_NAME_ERROR – even though you specified a perfectly good queue manager name. The NT event log on the Web server will show error messages reporting that the WMQ client cannot find the channel table. The file path will look OK but it isn't available to the application since it is a network path.
- The second problem is that WMQ can grant access only to security IDs it can see, that is domain user-IDs and groups or user-IDs and groups that are local to the queue manager machine. When you try to grant ASPNET access to WMQ resources using **SetMQAut** you will get an 'Invalid Principal' error.

## SOLUTIONS

To solve the channel table problem copy it to the Web server's local hard drive and change environment variable MQCHLLIB to point to the new location.

There are two ways you can make WMQ grant access to ASPNET; one is to put a queue manager on the Web server machine. The ASPNET account will be visible to that queue manager because it is local to that queue manager. Remote queue managers may be reached via remote queues. Unfortunately, queue managers are not inexpensive so it may not be cost-effective to put a queue manager on every one of your Web servers.

As an alternative to proliferating queue managers there is a way to trick WMQ into granting access to ASPNET. On the queue manager machine create a new local user named ASPNET. Add this user to the appropriate local groups to provide the necessary access or use the **SetMQAut** command on that user-ID. After restarting WMQ your ASP.Net applications will connect to that queue manager without a problem.

## IMPLICATIONS

Most companies have a good reason for putting their channel table on the network. As new queue managers are added or existing ones moved to different machines a shared network-based channel table will always be up-to-date. By switching to a local channel table you must assume the responsibility for keeping it current.

A more important issue is this – every ASP.Net application uses the same user-ID regardless of what machine it is running on. Therefore, when you grant access to ASPNET you can't pick and choose which Web servers you are authorizing – you grant the same access to every ASP.Net application in your company. This would be undesirable for queue managers that pass confidential data.

Although this particular issue cannot be avoided it can be managed.

I suggest you designate one of your queue managers as a 'gateway' and allow all your ASP.Net applications to connect only to that one. If you currently use a hub-and-spoke architecture, where all queue managers connect through a single central queue manager, I suggest you use that hub as your gateway.

*Mills Perry*
*Zyquest (USA)*


# Investigating server channel performance

This article is aimed at WMQ system administrators and architects who have an interest in the performance of server channels between two queue managers. It is an advantage for the reader to be familiar with the WMQ **runmqsc** command line processor, server channel statistics, and general performance concepts.

The primary focus of this article is two of the server channel statistics, MSGS and BATCHES; three of the server channel attributes, BATCHSZ, BATCHINT, and NPMSPEED; and the queue attribute CURDEPTH. These can be queried using the command line processor **runmqsc**, WMQ Explorer for Windows, or z/OS methods, such as **CSQUTIL** or the ISPF panels.

## OVERVIEW OF MSGS AND BATCHES

The number of messages and batches transmitted is a running total from the time the channel was started – channel start date 'CHSTADA' – and channel start time 'CHSTATI'. When a channel is restarted some attribute values are reset, including messages (MSGS) and batches (BATCHES). When a message is transmitted over a server channel between two queue managers it is transmitted as part of a batch (preferably on its own in a batch). One or more messages are transmitted in a single batch, ie at least one, but never zero.

Communication data is transmitted between queue managers outside the scope of a batch – the number of bytes sent (BYTSSENT) and bytes received (BYTSRCVD) increases while the channel is active but the number of messages (MSGS) and batches (BATCHES) may not. The number of bytes sent and received changes at the heartbeat interval (HBINT) when the two queue managers transmit data to keep alive the transport mechanism underpinning the server channel. This 'outside of batch scope' is not discussed in this article; however, the effects of the heartbeat interval will be examined later.

## CALCULATING AN EFFECTIVE BATCH SIZE

The server channel attribute values for MSGS and BATCHES can be used to derive the effective size of batches being transmitted over a server channel. The reader may be familiar with the term 'effective batch size' from previous discussions of WMQ server channel performance. It is not a metric generated by the queue manager; it should be considered as a cumulative rolling average. The effective batch size is calculated by taking the change in values for MSGS and BATCHES, ideally (but not necessarily) for an increment of 1 in the value of BATCHES.

Use the procedure outlined below to record values for MSGS and BATCHES:

1 To obtain a list of channels:

```
DIS CHL(*) TYPE(SDR|SVR|RCVR|RQSTR)
```

2 To obtain channel attribute values:

```
DIS CHS(<CHNAME>) CURRENT ALL
```

3 Divide the number of MSGS by the number of BATCHES:

- effective batch size=MSGS÷BATCHES (average messages per batch).

From two attributes with little individual use for monitoring performance this derived metric uses the total number of messages transmitted over the channel and the number of batches used to

transmit them to determine the ‘effective size’ of each batch of messages. Using totals in this way the effective size is the average number of messages in each batch since the channel was started.

The absolute size of the very next batch can now be found if values for MSGS and BATCHES can be recorded for an increase of 1 in the number of batches. If the channel has an even utilization, ie there are no bursts of messaging, the effective batch size since the channel was started will approximately follow the effective size for further batches and there may be no added benefit in obtaining an absolute size.

## BEHAVIOUR IMPACTING EFFECTIVE BATCH SIZE

The channel and queue attributes listed below have an impact on the effective batch size.

- Channel batch size (BATCHSZ).
- Batch interval (BATCHINT).
- Queue current depth (CURDEPTH).

Application parallelism, non-persistent message speed, and pipeline length are also discussed in the context of the channel batch size.

A batch is closed on one of two conditions, the first of which is when the number of messages in the batch reaches the maximum batch size for the channel (BATCHSZ). At this point the batch is closed and no more messages are added. The default for BATCHSZ is 50 but even under a heavy load it is rare to see the effective batch size reach this maximum.

The second condition that leads to the batch being closed is at the point the channel transmission queue becomes empty, ie the current queue depth (CURDEPTH) is zero. Strictly speaking the batch is closed only when the batch interval is also zero. If the effective batch size on a channel is only a fraction of the maximum batch size this does not indicate a performance problem. The Message Channel Agent (MCA) is an efficient program and is

dedicated to the task of processing and transmitting batches.

The effective batch size of a channel can be increased if the transmission queue depth can be prevented from becoming empty so quickly. The simplest way to achieve this is to increase the parallelism of messaging applications using that channel. The reader should not use the batch interval for increasing the effective batch size unless they understand the circumstances where it should be used.

If the batch interval is greater than zero when the channel transmission queue is empty the batch interval timer is started. When the timer reaches the value of the batch interval the batch is closed. If a message arrives on the transmission queue while the timer is running, the message is added to the batch and the timer is not reset.

The batch interval provides a window of opportunity for increasing the batch size by keeping the batch open while the channel transmission queue is empty.

The default batch interval is zero since, normally, when the transmission queue is empty, it is undesirable to introduce delay before closing the batch. However, for small batches in certain situations a short delay in waiting for the next message to arrive and adding it to the current batch can provide a performance enhancement. With a batch interval of zero the next message arrives on the transmission queue while the MCA is processing the last batch and there is a delay waiting for the previous batch to complete (see the following section on pipeline length).

It is useful to remember that, while a larger batch has a greater processing cost, the cost per message is lower and fewer batches need to be transmitted for the same number of messages.

Although the batch size can be reduced to allow the effective batch size (MSGS÷BATCHES) to reach its maximum, this does not provide superior performance. The default maximum batch size is configured at 50 because this is a number where the batch processing cost per message is low; adding more messages to the batch does not significantly reduce the per-message cost.

Furthermore, the larger the batch the more data is transmitted over the channel and the longer the time can be in waiting for messages to be delivered to the destination queue.

## NON-PERSISTENT MESSAGE SPEED

Using the default non-persistent message speed (NPMSPEED) of 'FAST', non-persistent messages can be transmitted over the channel outside of syncpoint (ie not part of a Unit of Work). These messages are visible on the destination queue before persistent messages and this can be misconstrued as non-persistent messages overtaking persistent messages on the channel.

For each batch of messages the MCA performs end-of-batch processing. During this phase non-persistent (when NPMSPEED is set to 'NORMAL') and persistent messages are committed by the sending and receiving MCAs. The messages in the batch arrive in the same order that they are sent but, because a persistent message appears on a queue only when the Unit of Work that it is part of is committed, a non-persistent message sent as part of the same batch (when NPMSPEED is FAST) can become visible on the destination queue first.

It is important to note that, when using an NPMSPEED of FAST, if there is a communication failure while the MCAs are transmitting or committing the batch, persistent messages will be returned to the sending MCA and non-persistent messages will either be delivered to the receiving MCA or dropped *en route* because they are not part of the MCA Unit of Work.

Considering these factors, it is no surprise to hear that both persistent and non-persistent messages contribute to the batch size.

### Pipeline length

When a batch is closed the MCA performs end of batch. Some queue managers can be configured to use two threads of control in the MCA (Channels stanza, PipelineLength=2) so that during end of batch processing the second thread can start a new batch

while the first thread of control is still handling a batch. This is particularly useful for channels with a long 'line turn-around' time (as discussed earlier in the context of batch interval).

## INSTALLING THE MONITORING SCRIPT

Copy the monitoring script (BATCHSZ, included below) to a sensible place on the queue manager machine. PERL is the only prerequisite; there are no other files, modules, or packages required by the script. The script has been tested on Windows and HP-UX. Invoking the script with '-h' gives a usage statement.

A variation of the tool provided was used by the author to measure the effective batch size for a few high-utilization, high-throughput server channels and, separately, for a few randomly selected (from thousands of active channels) low-utilization, low-throughput server channels. This was to ensure that there were messages flowing across those channels at an anticipated frequency between two iterations of the script.

## LIMITATIONS OF A MANUAL TECHNIQUE

The manual technique is not good for monitoring a number of channels on an iterative basis. The quickest method of querying the queue manager channel of interest is to name channels similarly or use the smallest subset of **DISPLAY CHSTATUS** commands, using wildcard matching where possible and reducing the attributes to the minimum: MSGS and BATCHSZ.

## LIMITATIONS OF AN AUTOMATED TECHNIQUE

Although an automated technique is easiest, when a channel is in use the data collected becomes very quickly outdated. The more channels queried the more intrusive and more time-consuming the process becomes, which can introduce contention for system resources; so much so that with thousands of channels it is not possible to take a subsequent measurement for each channel until all the channels have been queried. Alternatively, if quick, successive measurements are taken for the same channel

(attempting to record a single increase in the number of transmitted batches), subsequent channels may not be queried until some time after the measurements for earlier channels.

## SUMMARY

The important points to note from this article are:

- Persistent and non-persistent messages contribute to the batch size.
- The maximum number of messages sent or received in a batch is limited by the server channel attribute BATCHSZ.
- A batch of messages is sent over the channel when either:
  - the maximum number of messages in one batch is reached, or
  - the transmission queue is empty and the batch interval has expired.
- The batch interval should not be used to drive up batch size. This attribute should be tuned and the effective batch size observed.
- The effective batch size can be derived by dividing the number of MSGS by the number of BATCHES.
- A small, effective batch size does not necessarily indicate bad performance. It is important to note that you should not reduce BATCHSZ to allow the effective batch size to reach this value.

## BATCHSZ

```
#!/usr/bin/perl
# batchsz.pl
# Copyright International Business Machines Corporation 1998,2001
# Change History:
use strict;
$| = 1;
my $OUTPUT = 0;
my %qmname = (
);
```

```
my ($qmname, $channel_name_prefix, $number_of_channels) = (undef,
undef, Ø);
my ($bytes_threshold, $msgs_threshold, $batches_threshold) = (Ø, Ø, Ø);
print "WebSphere MQ Channel Performance Tool\n";
print "(C)IBM 2ØØ3. Version 2.Ø, 7th Oct 2ØØ3\n";
print "WebSphere MQ Performance Group, SWG\n";
print "\n";
my @qmname;
my %params = ();
my $METHOD = undef;
my %MQSC;
my $CHSTATUS_ALL = 1;
my %STATUS;
my %STATUS_init = (
  'STARTING'     => Ø,
  'BINDING'      => Ø,
  'INITIALIZING' => Ø,
  'RUNNING'      => Ø,
  'STOPPING'     => Ø,
  'RETRYING'     => Ø,
  'PAUSED'       => Ø,
  'STOPPED'      => Ø,
  'REQUESTING'   => Ø,
);
my %BYTSSENT_total = ();
my %BYTSRCVD_total = ();
my %MSGS_total = ();
my %BATCHES_total = ();
my $spin_text = '-\|/';
my $spin_counter = Ø;
my $spin_length = length($spin_text);
if ($#ARGV == Ø && ($ARGV[Ø] eq '-?' or $ARGV[Ø] eq '-h')) {
  &usage;
  &help;
  exit(Ø);
}
my ($interval, $count) = (Ø, 1);
if ($#ARGV >= 1) {
  $interval = shift @ARGV;
  $count = shift @ARGV;
} elsif ($#ARGV != 2 and $#ARGV != 3) {
  &usage;
  exit(Ø);
}
if ($^O !~ /Win32/i && $#ARGV == -1) {
  print "Finding active queue manager(s) using 'amqzxmaØ -m'...\n";
  $METHOD = '(discovered)';
  my @pids = 'ps -ef | grep "amqzxmaØ -m" | grep -v grep';
  if ($#pids >= Ø) {
    for (my $p = Ø; $p <= $#pids; $p++) {
```

```
      chomp($pids[$p]);
      $pids[$p] =~ s/^.*amqzxmaØ\s+\S+\s+(\S+)\s*$/$1/;
      next if (!defined($qmname{$pids[$p]}));
      print "Found active queue manager: $pids[$p]\n";
      push (@qmname, $pids[$p]);
    }
  } else {
    print "No active queue manager(s) found\n";
  }
  print "\n";
} else {
  $METHOD = '(manual)';
}
if ($#ARGV >= 1) {
  $qmname[Ø] = shift @ARGV;
  $channel_name_prefix = shift @ARGV;
  if ($#ARGV >= Ø) {
    $number_of_channels = shift @ARGV;
  }
  if ($#ARGV == Ø) {
    $bytes_threshold = shift @ARGV;
  }
}
if ($#ARGV >= Ø) {
  &usage;
  exit(Ø);
}
print "Parameters $METHOD:\n";
print "- qmname(s)          : ".join(', ', @qmname)."\n";
print "- channel_name_prefix: $channel_name_prefix\n";
print "- number_of_channels : $number_of_channels\n";
print "- bytes_threshold    : $bytes_threshold\n";
print "\n";
if ($count > 1) {
  print "Iterating $count time(s) at sleep frequency: $interval
second(s)\n";
  print "Note: sleep can be pre-empted by sending SIGALRM to pid:
$$\n";
  print "\n";
}
for (my $c = Ø; $c < $count; $c++) {
  if ($count > 1) {
    print "Iteration number ".($c+1)." of $count\n";
    print "\n";
  }
  &get_all_channel_throughput;
  if ($c == $count-1) {
  } else {
    print "".($count-$c-1)." more iteration(s) to go\n";
    print "Going to sleep for $interval second(s)...\n";
```

```
    print "\n";
    sleep($interval);
  }
}
exit(0);
sub reset_spinner
{
  $spin_counter = 0;
  print "\n";
}
sub update_spinner
{
  my ($prefix, $suffix) = (shift, shift);
  print "\r$prefix ".substr($spin_text, $spin_counter, 1)." $suffix";
  $spin_counter++;
  if ($spin_counter >= $spin_length) {
    $spin_counter = 0;
  }
}
sub get_localtime
{
  my $time = localtime(time);
  $time =~ s/(\s+)/\ /g;
  return ($time);
}
sub get_channel_throughput
{
  my ($cmd, $c, $qmname) = (shift, shift, shift);
  my $rc = 0;
  my (@BYTSSENT, @BYTSRCVD, @MSGS, @BATCHES);
  if ($cmd =~ /\s+CHS\w*\(/i) {
    if (open(MQSC, "echo '$cmd' | runmqsc $qmname |")) {
      while (my $mqsc = <MQSC>) {
        chomp($mqsc);
        if ($mqsc =~ /AMQ8147/) {
          print "\n** Warning: WebSphere MQ object not found **\n";
          return (-1, undef, undef);
          last;
        }
      }
      close(MQSC);
    }
  }
  if (!$CHSTATUS_ALL) {
    if ($cmd =~ /\sALL\s*$/i) {
      $CHSTATUS_ALL = 1;
    }
  }
  if (open MQSC, "echo '$cmd' | runmqsc $qmname |") {
    my $rc = 1;
```

```
    my ($BYTSSENT_is_same, $BYTSSENT_below_threshold, $BYTSSENT) = (0,
0, -1);
    my ($BYTSRCVD_is_same, $BYTSRCVD_below_threshold, $BYTSRCVD) = (0,
0, -1);
    my (    $MSGS_is_same,     $MSGS_below_threshold, $MSGS)     = (0,
0, -1);
    my ( $BATCHES_is_same,  $BATCHES_below_threshold, $BATCHES)  = (0,
0, -1);
    while (my $mqsc = <MQSC>) {
      chomp($mqsc);
      if ($mqsc =~ /AMQ8118/) {
        print "\n** Warning: queue manager '$qmname' does not exist
**\n";
        $rc = 0;
        last;
      }
      if ($mqsc =~ /AMQ8146/) {
        print "\n** Warning: queue manager '$qmname' is not started
**\n";
        $rc = 0;
        last;
      }
      if ($mqsc =~ /AMQ8420/) {
        print "\n** Warning: channel status not found **\n";
        next;
      }
      if ($mqsc =~ /STATUS/) {
        $mqsc =~ s/STATUS\((\w+)\s*\)/$1/;
        $STATUS{$1}++;
      }
      if ($CHSTATUS_ALL) {
        if ($mqsc =~ /BYTSSENT/) {
          $mqsc =~ /BYTSSENT\((\d+)\s*\)/;
          $BYTSSENT = $1;
          ($BYTSSENT_is_same, $BYTSSENT_below_threshold) =
            do_threshold($BYTSSENT, $bytes_threshold, $c, "BYTSSENT");
        }
        if ($mqsc =~ /BYTSRCVD/) {
          $mqsc =~ /BYTSRCVD\((\d*).*\)/;
          $BYTSRCVD = $1;
          ($BYTSRCVD_is_same, $BYTSRCVD_below_threshold) =
            do_threshold($BYTSRCVD, 0, $c, "BYTSRCVD");
        }
        if ($mqsc =~ /MSGS/) {
          $mqsc =~ /MSGS\((\d+)\s*\)/;
          $MSGS = $1;
          ($MSGS_is_same, $MSGS_below_threshold) =
            do_threshold($MSGS, $msgs_threshold, $c, "MSGS");
        }
        if ($mqsc =~ /BATCHES/) {
```

```
          $mqsc =~ /BATCHES\((\d+)\s*\)/;
          $BATCHES = $1;
          ($BATCHES_is_same, $BATCHES_below_threshold) =
            do_threshold($BATCHES, $batches_threshold, $c, "BATCHES");
        }
      }
    }
    close (MQSC);
    my @BYTSSENT = ($BYTSSENT_is_same, $BYTSSENT_below_threshold,
$BYTSSENT);
    my @BYTSRCVD = ($BYTSRCVD_is_same, $BYTSRCVD_below_threshold,
$BYTSRCVD);
    my @MSGS     = (    $MSGS_is_same,     $MSGS_below_threshold,
$MSGS);
    my @BATCHES  = ( $BATCHES_is_same,  $BATCHES_below_threshold,
$BATCHES);
    return ($rc, \@BYTSSENT, \@BYTSRCVD, \@MSGS, \@BATCHES);
  } else {
    print "** Error: failed to invoke runmqsc for queue manager
'$qmname' **\n";
    return (0);
  }
}
sub do_threshold
{
  my ($value, $value_threshold, $c, $name) = (shift, shift, shift,
shift);
  my ($value_is_same, $value_below_threshold) = (0, 0);
  my ($hash, $array) = ("", "");
  if ($c > 0) {
    $hash = "$c:";
    $array = "[$c]";
  }
  if (defined($MQSC{$hash.$name})) {
    $MQSC{$hash.$name.'~'} = $MQSC{$hash.$name};
    if ($MQSC{$hash.$name} == $value) {
      $value_is_same = 1;
    } elsif ($value_threshold > 0) {
      my $previous = $MQSC{$hash.$name.'~'};
      my $present  = $value;
      if (abs($previous-$present) < $value_threshold) {
        $value_below_threshold = 1;
      }
    }
  }
  $MQSC{$hash.$name} = $value;
  print $name.$array.'='.$MQSC{$hash.$name}."\n" if ($OUTPUT);
  return ($value_is_same, $value_below_threshold);
}
sub get_all_channel_throughput
```

```
{
  print "Preparing to invoke WebSphere MQ command console 'runmqsc'
for: ";
  for (my $m = Ø; $m <= $#qmname; $m++) {
    print $qmname[$m];
    if ($m < $#qmname) {
      print ", ";
    }
  }
  print "\n";
  my @revisedqm;
  for (my $m = Ø; $m <= $#qmname; $m++) {
    print "Invoking command console to query queue manager
'$qmname[$m]'...\n";
    if (open (MQSC, "echo 'DIS QMGR' | runmqsc $qmname[$m] |")) {
      my ($AMQ8118, $AMQ8146) = (Ø, Ø);
      while (my $mqsc = <MQSC>) {
       chomp($mqsc);
        print "MQSC: $mqsc\n" if ($OUTPUT);
        if ($mqsc =~ /AMQ8118/) {
          print "** Warning: queue manager '$qmname[$m]' does not exist
**\n";
          $AMQ8118 = 1;
          last;
        }
        if ($mqsc =~ /AMQ8146/) {
          print "** Warning: queue manager '$qmname[$m]' is not started
**\n";
          $AMQ8146 = 1;
          last;
        }
      }
      push (@revisedqm, $qmname[$m]) if (!$AMQ8118 and !$AMQ8146);
      close(MQSC);
    } else {
      print "** Error: failed to invoke runmqsc for queue manager
'$qmname[$m]' **\n";
    }
    print "\n";
  }
  if ($#revisedqm != $#qmname) {
    print "** Warning: ";
    print "non-existent and inactive queue managers will not be invoked
**\n";
    print "\n";
  }
  @qmname = @revisedqm;
  if ($#qmname < Ø) {
    print "There are no remaining active queue managers to work
with\n";
```

```
    print "=> Exited normally\n";
    exit(Ø);
  }
  print "Start time is: ".&get_localtime()."\n";
  print "\n";
  FOR: for (my $m = Ø; $m <= $#qmname; $m++) {
    %STATUS = %STATUS_init;
    my $qmname = $qmname[$m];
    my (@BYTSSENT_same_list, @BYTSSENT_below_list);
    $BYTSSENT_total{$qmname.'~'} = $BYTSSENT_total{$qmname};
    $BYTSSENT_total{$qmname} = Ø;
    my (@BYTSRCVD_same_list, @BYTSRCVD_below_list);
    $BYTSRCVD_total{$qmname.'~'} = $BYTSRCVD_total{$qmname};
    $BYTSRCVD_total{$qmname} = Ø;
    my (@MSGS_same_list, @MSGS_below_list);
    $MSGS_total{$qmname.'~'} = $MSGS_total{$qmname};
    $MSGS_total{$qmname} = Ø;
    my (@BATCHES_same_list, @BATCHES_below_list);
    $BATCHES_total{$qmname.'~'} = $BATCHES_total{$qmname};
    $BATCHES_total{$qmname} = Ø;
    if ($^O !~ /Win32/i) {
      print "Cursory check for active channel(s) for queue manager
'$qmname' ";
      print "using 'amqcrsta -m'...\n";
    }
    my @pids = ();
    if ($^O !~ /Win32/i) {
      push (@pids, 'ps -ef | grep "amqcrsta" | grep $qmname | grep -v
grep');
      push (@pids, 'ps -ef | grep "runmqlsr" | grep $qmname | grep -v
grep');
    }
    if ($^O =~ /Win32/i || $#pids >= Ø) {
      if ($^O !~ /Win32/i) {
        print "Found ".($#pids+1)." active channel(s) for queue manager
'$qmname'\n";
        if ($number_of_channels != $#pids+1) {
          print "\n";
          print "** Warning: parameter ";
          print "'number_of_channels' different to active channels
**\n";
          print "** Warning: number_of_channels=$number_of_channels, ";
          print "active channels=".($#pids+1)." **\n";
          print "\n";
        }
      }
      print "Invoking command console to query channel(s) ";
      print "for queue manager '$qmname'...\n";
      my ($cmd, $spinner, $spooner) = (undef, '', '');
      my ($rc, $c) = (Ø, Ø);
```

```
      my ($BYTSSENT_is_same, $BYTSSENT_below_threshold, $BYTSSENT);
      my ($BYTSRCVD_is_same, $BYTSRCVD_below_threshold, $BYTSRCVD);
      my (    $MSGS_is_same,     $MSGS_below_threshold, $MSGS);
      my ( $BATCHES_is_same,  $BATCHES_below_threshold, $BATCHES);
      while (1) {
        # WebSphere MQ Command Reference: channel-name max length 2Ø
characters
        if ($number_of_channels == Ø) {
          $cmd = "DIS CHS($channel_name_prefix) ALL";
          $c = -1;
        } else {
          $cmd = "DIS CHS($channel_name_prefix".sprintf("%-2Ød", $c).")
ALL";
        }
        if (defined($number_of_channels) && $number_of_channels > Ø) {
          last if ($c > $number_of_channels);
          $spinner = "$cmd: ";
          $spooner = "(".sprintf("%3d", ($c+1)/
$number_of_channels*1ØØ)."%)";
        }
        &update_spinner($spinner, $spooner);
        my @list = &get_channel_throughput($cmd, $c, $qmname);
        if ($CHSTATUS_ALL) {
          $rc = shift @list;
          last if ($rc == Ø);
          my $BYTSSENT_ref = shift @list;
          my @BYTSSENT = @$BYTSSENT_ref;
        ($BYTSSENT_is_same, $BYTSSENT_below_threshold, $BYTSSENT) =
@BYTSSENT;
          my $BYTSRCVD_ref = shift @list;
          my @BYTSRCVD = @$BYTSRCVD_ref;
         ($BYTSRCVD_is_same, $BYTSRCVD_below_threshold, $BYTSRCVD) =
@BYTSRCVD;
          my $MSGS_ref = shift @list;
          my @MSGS = @$MSGS_ref;
        ($MSGS_is_same, $MSGS_below_threshold, $MSGS) = @MSGS;
          my $BATCHES_ref = shift @list;
          my @BATCHES = @$BATCHES_ref;
        ($BATCHES_is_same, $BATCHES_below_threshold, $BATCHES) =
@BATCHES;
        }
        if ($m < $#qmname) {
          print "Intermediate end time: is ".&get_localtime()."\n";
          print "\n";
        }
        if ($rc == -1) {
          my $prematurely = '';
          if ($c < $number_of_channels) {
            $prematurely = ' prematurely';
          }
```

```
      print "** Warning: channel processing ended$prematurely at ";
      print "$c of $number_of_channels requested channel(s) **\n";
      next FOR;
    }
    if ($CHSTATUS_ALL) {
      if ($BYTSSENT_is_same) {
        push (@BYTSSENT_same_list, $c);
      }
      if ($BYTSSENT_below_threshold) {
        push (@BYTSSENT_below_list, $c);
      }
      if ($BYTSRCVD_is_same) {
        push (@BYTSRCVD_same_list, $c);
      }
      if ($BYTSRCVD_below_threshold) {
        push (@BYTSRCVD_below_list, $c);
      }
      if ($MSGS_is_same) {
        push (@MSGS_same_list, $c);
      }
      if ($MSGS_below_threshold) {
        push (@MSGS_below_list, $c);
      }
      if ($BATCHES_is_same) {
        push (@BATCHES_same_list, $c);
      }
      if ($BATCHES_below_threshold) {
        push (@BATCHES_below_list, $c);
      }
    }
    if ($BYTSSENT >= 0) {
      $BYTSSENT_total{$qmname} += $BYTSSENT;
    }
    if ($BYTSRCVD >= 0) {
      $BYTSRCVD_total{$qmname} += $BYTSRCVD;
    }
    if ($MSGS >= 0) {
      $MSGS_total{$qmname} += $MSGS;
    }
    if ($BATCHES >= 0) {
      $BATCHES_total{$qmname} += $BATCHES;
    }
    $c++;
    last if ($c == $number_of_channels);
  }
  &reset_spinner;
  my $total = 0;
  print "Channel STATUS statistics for queue manager '$qmname':\n";
  foreach my $status (keys %STATUS) {
    next if ($STATUS{$status} <= 0);
```

```
        my $field = sprintf("%-12s", $status);
        print "- $field: ".$STATUS{$status}."\n";
        $total += $STATUS{$status};
      }
      if (defined($number_of_channels)) {
        print "=> ";
        if ($number_of_channels == 0) {
          print "status found for requested channel
'$channel_name_prefix'\n";
        } elsif ($total == $number_of_channels) {
          print "status found for all requested $total channel(s)\n";
        } else {
          print "status not found for ".($number_of_channels-$total);
          print " channel(s)\n";
        }
      }
      if ($number_of_channels > 0) {
        print "Note: there are no channels for queue manager '$qmname'
in any other state\n";
      }
      if ($BYTSSENT_total{$qmname} >= 0) {
        if ($BYTSSENT_total{$qmname} == 0) {
          print "No channel throughput measured for queue manager
'$qmname'\n";
        } else {
          print "Channel statistics for queue manager '$qmname':\n";
          print "- BYTSSENT    : $BYTSSENT_total{$qmname}\n";
          print "- BYTSRCVD    : $BYTSRCVD_total{$qmname}\n";
          print "- MSGS        : $MSGS_total{$qmname}\n";
          print "- BATCHES     : $BATCHES_total{$qmname}\n";
          if ($BYTSSENT_total{$qmname.'~'} > 0) {
            my $sent = $BYTSSENT_total{$qmname}-
$BYTSSENT_total{$qmname.'~'};
            my $rcvd = $BYTSRCVD_total{$qmname}-
$BYTSRCVD_total{$qmname.'~'};
            my $bytes = $sent + $rcvd;
            print "=> $bytes additional byte(s) on this iteration\n" if
($bytes > 0);
            my $msgs = $MSGS_total{$qmname}-$MSGS_total{$qmname.'~'};
            my $batches = $BATCHES_total{$qmname}-
$BATCHES_total{$qmname.'~'};
            print "=> $msgs msg(s) and $batches batch(es) this
iteration\n" if ($msgs > 0);
          }
        }
      }
    } elsif ($^O !~ /Win32/i) {
      print "Found no active channels for queue manager '$qmname'\n";
    }
    if ($CHSTATUS_ALL) {
```

```
      if ($OUTPUT) {
        print "\n";
        for (my $s = Ø; $s < $#BYTSSENT_same_list; $s++) {
          print "** Warning:
CHL($channel_name_prefix$BYTSSENT_same_list[$s])";
          print " BYTSSENT no change **\n";
        }
      } elsif ($#BYTSSENT_same_list >= Ø) {
        print "\n";
        print "** Warning: ".($#BYTSSENT_same_list+1)." channel(s)";
        print " BYTSSENT no change **\n";
      } elsif ($#BYTSSENT_below_list < Ø and $#BYTSSENT_same_list < Ø)
{
        if (defined($BYTSSENT_total{$qmname.'~'})) {
          if ($BYTSSENT_total{$qmname} > $BYTSSENT_total{$qmname.'~'})
{
            print "\n";
            if ($number_of_channels == Ø) {
              print "Channel BYTSSENT changed this interation\n";
            } else {
              print "All channels BYTSSENT changed this interation\n";
            }
          }
        }
      }
      if ($#BYTSSENT_below_list >= Ø) {
        print "** Warning: ".($#BYTSSENT_below_list+1)." channel(s)
BYTSSENT";
        print " change below threshold '$bytes_threshold' **\n ";
      } elsif ($bytes_threshold > Ø and
defined($BYTSSENT_total{$qmname.'~'})) {
        if ($BYTSSENT_total{$qmname} > $BYTSSENT_total{$qmname.'~'}) {
          if ($number_of_channels == Ø) {
            print "Channel above BYTSSENT threshold this iteration\n";
          } else {
            print "All channels above BYTSSENT threshold this
iteration\n";
          }
        }
      }
    }
    print "\n";
  }
  print "End time is: ".&get_localtime()."\n";
  print "\n";
}
sub usage
{
  print "usage: perl batchsz.pl <interval> <count> [options...]\n";
  print "options => qmname                  : queue manager name\n";
```

```
  print "           channel_name            : server channel name\n";
  print "           number_of_channels      : number of channels\n";
  print "           BYTSSENT_below_threshold: threshold number of
bytes\n";
  print "For Example:\n";
  print "\$ perl chanperf.pl 120 2 QMGR CHANNEL 1000 2048\n";
}
sub help
{
#print "\$ perl batchsz.pl 120 2 QMGR CHANNEL 1000 2048\n";
  print '                  |   | |    |       |    |'."\n";
  print "                 (1) (2,3)  (4)     (5)  (6)\n";
  print "Commentary:\n";
  print " 1. wait two minutes between interations\n";
  print " 2. make two iterations\n";
  print " 3. use the queue manager called 'QMGR'\n";
  print "   (passed to the DISPLAY QMGR in runmqsc)\n";
  print " 4. use the channel prefix name 'CHANNEL'\n";
  print "   (passed to the DISPLAY CHANNEL and DISPLAY CHSTATUS in
runmqsc)\n";
  print " 5. use one thousand channels\n";
  print "  *(i.e. CHANNEL0, CHANNEL1, .... CHANNEL999)\n";
  print " 6. set the bytes sent threshold required increase to 2048
bytes\n";
  print "   (a warning is given when the increase between two
consecutive\n";
  print "    DISPLAY CHSTATUS(CHANNEL*) BYTSSENT values are less than
the\n";
  print "    threshold required increase)\n";
  print "*Note: this version of the tool must start at channel 0
(zero)\n";
  print "\n";
 print "\$ perl batchsz.pl 10 2 QMGR QMA.TO.QMB 0\n";
  print "Commentary:\n";
  print " Similar to before, but only query channel 'QMA.TO.QMB'\n";
  print " Do not impose a threshold in the number of bytes sent\n";
}
1;
```

*Alexander Russell*
*IBM Hursley (UK)*

# Writing a WMQI Broker input node

## INTRODUCTION

In 2000, when IBM delivered WebSphere MQ Integrator (WMQI) V2.0, the product was able to process only WMQ messages although it was positioned to handle any kind of input message. Over time, WMQI delivered other input nodes for WMQ Everyplace and SCADA devices.

Since the release of V2.1 of WMQI and WMQI Broker it has been possible to develop user-defined input nodes that allow message flows to process input messages from other sources.

This article describes our experience of developing a Timer Input node for WebSphere Business Integration for Financial Networks (WBI for FN). To understand the environment in which the Timer Input node is used we start with a general overview of the Timer Service. We go on to discuss the special tasks required when creating an input plug-in node compared with writing a 'normal' message processing plug-in. It is assumed that readers know how to write a message processing plug-in.

## OVERVIEW

The WMQI Broker documentation lists several potential reasons for writing input nodes: for example, there could be a need to use a specific file or database as message flow input and not an MQ queue. Such a situation applied to the Timer Service of WBI for FN. The overall structure of this service is shown in Figure 1.

This Timer Service provides a wake-up mechanism for other services, which can request timer event messages at a specified time or after a specified duration. The Timer Service generates these timer events and sends them as WMQ messages to the requesting service.

The Timer Service consists of a Timer Interface and a Timer

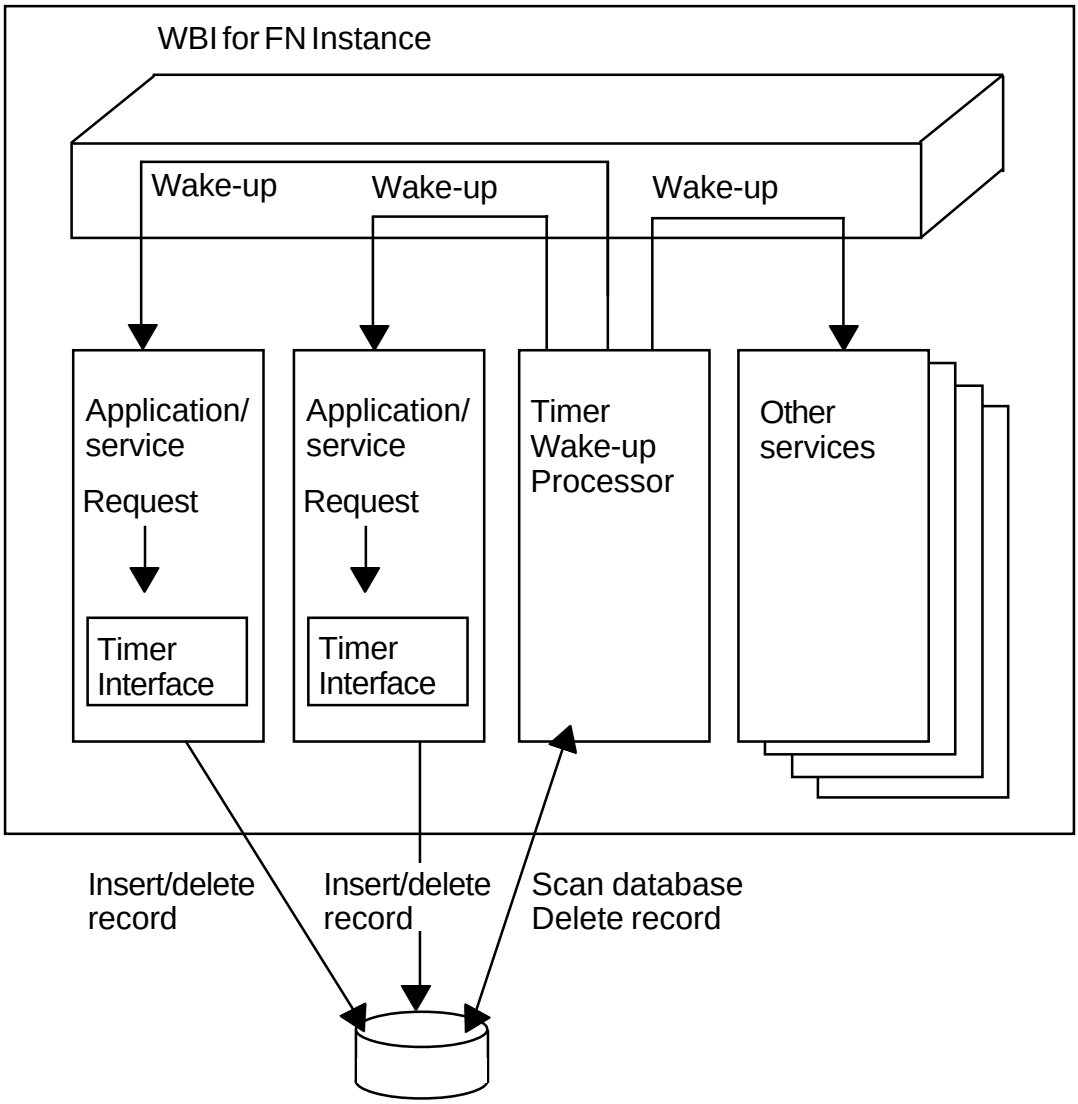


*Figure 1: Structure of the WBI for FN Timer Service*

Wake-up Processor. These parts communicate using a common database. The main reason for using a database is that it enables multiple brokers to access data simultaneously and the information is persistent so it will survive a broker restart.

The Timer Interface is an API that can be used within a WMQI Broker message flow. It is available as a set of nodes that a developer can use in the same way as any WMQI Broker node. The interface comprises a node used to request timer events as well as a node to cancel timer events that have already been scheduled.

For the node used to request timer events a user specifies the time when the event is to occur. This is called the wake-up time. Together with the wake-up time any additional information, such as the destination queue name for the timer event message, has to be passed with the message. This information is stored in the common timer database.

The wake-up time in the timer event indicates when a wake-up message is to be sent to trigger another service. For this task the Timer Service provides a message flow that regularly checks the timer database for expired timer events. If one is detected the message flow issues an MQPUT, using the MQOutput node provided by WMQI Broker to invoke the requested service via a WMQ message.

This message flow is called the Timer Wake-up Processor. The general structure of this message flow is shown in Figure 2. To start processing of the logic within the message flow on a regular basis, a special input node is needed.

Except for the special input node, the required processing reads expired events from the database, formats the wake-up messages as necessary, and sends them to the WMQ queue stored within the timer event.

The Timer Input node, the special input node, was implemented as part of WBI for FN. This node creates input messages for the flow and propagates them to the subsequent node in the message flow

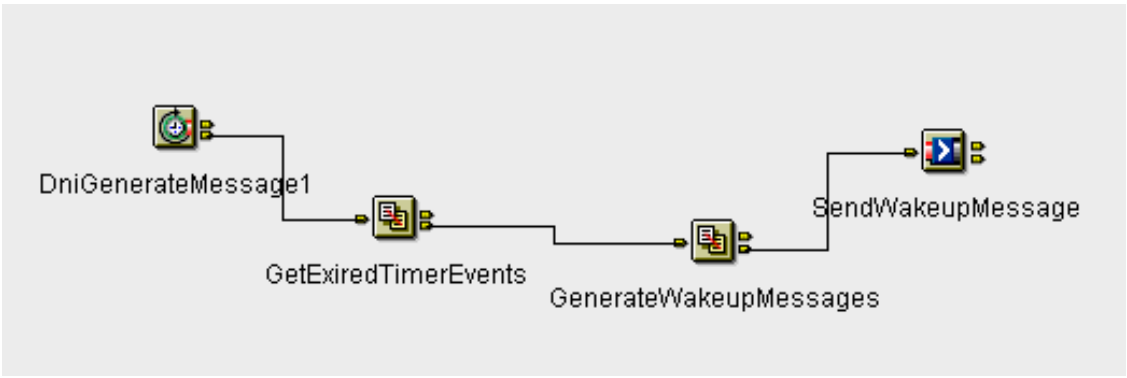


*Figure 2: General layout of the Timer Wake-up Processor*

through its out terminal. A compute node was used to get the expired events from the database and generate the wake-up messages; another compute node handles addressing the correct queue, and an MQOutput node sends the message.

Getting entries out of a database table, formatting messages, and sending them to an arbitrary queue are standard operations within WMQ Broker and are not shown in the general structure illustrated in Figure 2.

The setup of the Timer Wake-up Processor consists of several nodes that ensure both the main functionality needed in the message flow and the reusability of the input node.

When implementing the Timer Wake-up Processor the following design points require consideration:

- Whether the message flow should process one timer event at a time or all expired events at once.
- How the checks on the database should be carried out.

The ideal operation of the Timer Wake-up Processor message flow would be for the expired timer events to be processed at the time of expiry and the wake-up messages to be created and sent immediately. This would require real-time checks for changes in the common timer database. Unfortunately, real-time checks usually require a lot of resources in terms of CPU cycles and I/O bandwidth so it was decided to do the checking at defined intervals only.

For the interval processing it was decided to use a node attribute to control the time between the creation of messages. The attribute PollingInterval specifies the number of seconds the node has to wait between the creation of every new message.

The disadvantage of this approach is that it creates an inaccuracy in the delivery time of the wake-up messages. For example, if the polling interval value is one minute, in the worst case a wake-up message for a timer event could be delivered one minute too late because the Timer Wake-up Processor checked for expired timer entries immediately before the entry expired and the entry would

have to wait another minute (until the next polling interval expires) before a wake-up message could be sent.

This means that the accuracy of the Timer Wake-up Processor depends on the value of the PollingInterval attribute of the Timer Input node. This is usually acceptable and because all expired timer events are processed at once, with every message flow execution, this setup requires fewer resources than a real-time check as described above.

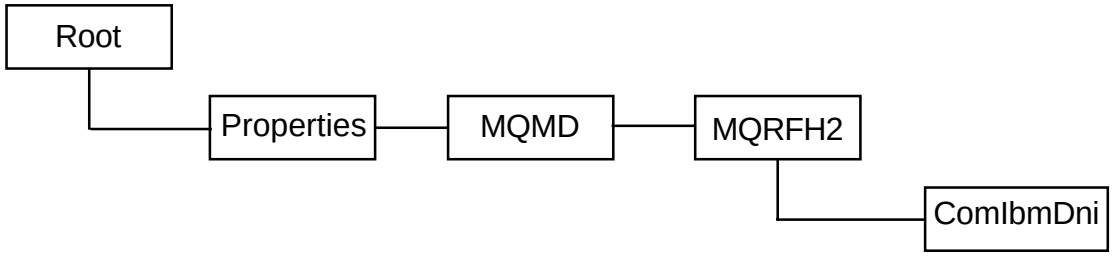


*Figure 3: Message structure created by the Timer Input plug-in node*

To ensure the reusability of the input node the structure of the input message is important. The folders generally used for processing in WMQI Broker are the MQMD and the MQRFH2. WBI for FN uses these two folders and a NameValueData section in the MQRFH2 folder for its internal processing data. Because the Timer Input node is currently used only within the WBI for FN environment, the Timer Input node creates a valid WBI for FN message.

The structure of the message is shown in Figure 3. The Properties folder in this figure is part of the structure required by WMQI Broker. For details of the message structure you should refer to the WMQI Broker documentation.

## THE NODE INTERFACE

Because the Timer Input node is used by several message flows

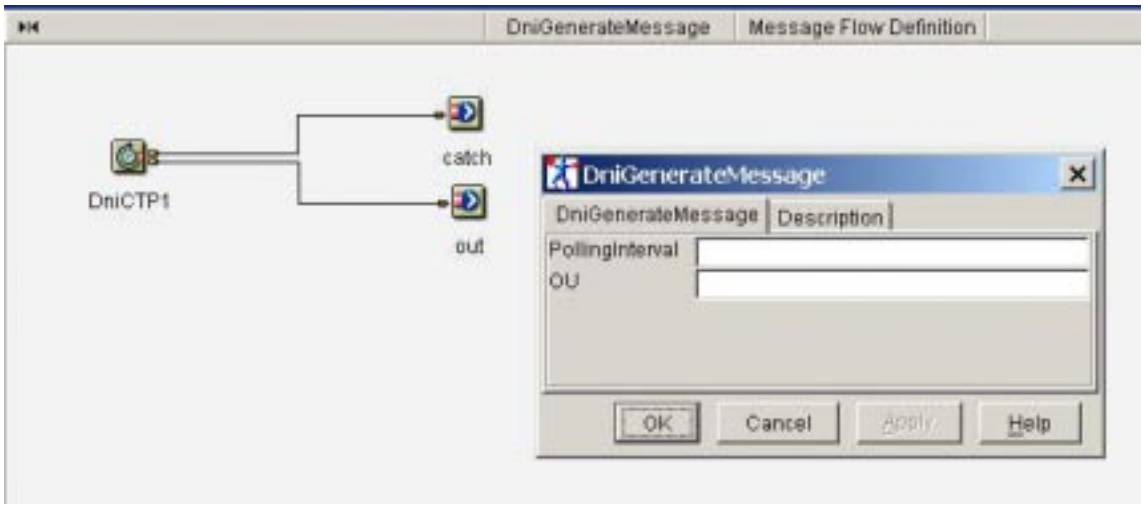


*Figure 4: DniGenerateMessage subflow encapsulating the Timer Input plug-in*

it was an important design consideration that it should be capable of being changed or enhanced without necessitating a change in the flows that use it.

It was decided, therefore, to encapsulate this node into a subflow. This allows the implementation of the node to be changed while leaving the interface to the node unchanged, eg by adding WMQI Broker nodes to the corresponding subflow.

Figure 4 shows subflow DniGenerateMessage, which encapsulates the Timer Input node DniCTP. DniGenerateMessage is the external interface for a message flow developer using the Timer Input function.

### Terminals of the Timer Input node

The Timer Input node has two output terminals: an out terminal and a catch terminal. The Timer Input node propagates a message to the out terminal every time the specified timer interval is expired. The catch terminal is used when an error occurs during the processing of the Timer Input node. Once this occurs the message flow developer can send a message to an operator, who then can resolve the problem.

### Properties of the Timer Input node

The Timer Input node has two properties:

- The polling interval (PollingInterval). The value of the PollingInterval property specifies the amount of time in seconds that needs to be between two messages that are sent to the node's out terminal.
- The name of a WBI for FN organizational unit (OU). The OU property is needed as an attribute of a valid WBI for FN message. An organizational unit in WBI for FN is used to separate resources within the processing of a message flow and for access control within a message flow. Further information about organizational units within WBI for FN can be found in the support documentation.

## IMPLEMENTATION ISSUES

The WBI for FN Timer Input node is a WMQI Broker plug-in that is implemented using the C interface to the broker. Its interface consists of two parts:

- Implementation functions, which are called by the broker during initialization and runtime. They must be implemented by the plug-in developer.
- Utility functions, which provide functionality for the plug-in's internal processing, for example, to manipulate a message or to request a service of the broker, such as propagating a message to subsequent nodes in a message flow. These utility functions are used by the plug-in itself.

For more information on these interfaces refer to the *WMQI Broker V2.1 Programming Guide* SC34-6170.

If the node initiates the implementation function cniRun() it signals to the broker that it has input node capabilities. cniRun() is called by the broker during runtime. It creates a message, propagates it to the subsequent nodes in the message flow, and returns control to the broker.

The periodical handing back of control to the broker is particularly important. As long as the control is within this function the broker cannot follow commands, eg a shutdown. The broker must wait until control is handed back to it by its message flows. Therefore, an input node must regularly return control to the broker to allow the broker to correctly maintain its state.

Because WBI for FN is available for z/OS, AIX, and Sun Solaris, the Timer Input node must support these platforms. The WMQI Broker C interface functions are identical on all platforms, so, in general, common source code could be generated, but these platforms use different character encoding. The International Classes for Unicode (ICU) were used to handle the code page transformation.

### Providing input data to the broker

As mentioned previously, the special behaviour of the input node is implemented in the implementation function cniRun(). In this function the input message must be created and propagated to subsequent nodes in the message flow.

Within the Timer Input node this function is used to create a message consisting of an MQMD and an MQRFH2, including a NameValueData section with the WBI for FN 'ComIbmDni' folder. See the *Overview* and Figure 3 for a detailed view of the message.

Using the WMQI Broker plug-in API there are three ways to create an output message:

- Use WMQI Broker utility functions to create each message element separately. The code fragments below show how this is done.

```
//
// Build up output message
// -> MQMD
CciMessageContext* pMessageContext; // message context
CciElement* pRootOut;               // output root element
// get message context
pMessageContext = cniGetMessageContext(&iRC, pMessageIn);
ctpCheckReasonCode(iRC, "invoke cniGetMessageContext()");
```

```
// create the output message with this message context
pMessageOut = cniCreateMessage(&iRC, pMessageContext);
ctpCheckReasonCode(iRC, "invoke cniCreateMessage()");
// create output message root element
pRootOut = cniRootElement(&iRC, pMessageOut);
ctpCheckReasonCode(iRC, "invoke cniRootElement() for output
 message");
// create MQMD header
CciElement* pMQMDHeader;
UnicodeString usMQMDParser("MQHMD");
CciChar* szMQMDParser =
 (CciChar*)usMQMDParser.append(CH_EOS).getBuffer();
pMQMDHeader = cniCreateElementAsFirstChildUsingParser(&iRC,
 pRootOut,
    szMQMDParser);
ctpCheckReasonCode(iRC,
    "invoke cniCreateElementAsFirstChildUsingParser() for MQMD");
```

This example shows the steps for creating an output message consisting of an MQMD header only. At this point this header does not have an element assigned to it.

Using the utility functions is very time-consuming. Another problem with using this method is that WMQI Broker does not create its Properties folder on its own. This must probably be done by the plug-in using the same method, but this has not been tried.

- The second option to create an output message is to use the Timer Input node to create a dummy message and so start the message flow at the right time. A subsequent compute node then could insert the MQMD and MQRFH2 headers and the ComIbmDni folder.
- The third option is to use the WMQI Broker utility function cniSetInputBuffer() to cause the plug-in to assemble a message in memory, using the predefined header structures defined in the WMQ header file (cmqc.h). These headers can be initialized to their default values. These default values can also be found in the same header file.

  Once initialized, individual fields within the message can be set according to the requirements of subsequent message processing. This is true for the MQMD and for the MQRFH2 header used by the WBI for FN Timer node. Additional

information can be added when providing your own input nodes.

When calling the function cniSetInputBuffer() with this message as a parameter the function attaches this message to the internal message object. During this procedure WMQI Broker automatically creates a Properties folder and appends it to the front of the message.

When creating the message in this way the built-in node attribute 'firstParserClassName' is very important. It specifies the parser class name of the first message part (excluding the Properties folder). In the example given below, the parser class name 'MQHMD' has to be used to indicate the MQMD.

```
 // create attribute firstParserClassName and set its value to MQMD
UnicodeString firstParserClassName((char*)"firstParserClassName");
szAttrName =
  (CciChar*)firstParserClassName.append(CH_EOS).getBuffer();
UnicodeString firstParserClassValue((char*)"MQHMD");
szAttrValue =
  (CciChar*)firstParserClassValue.append(CH_EOS).getBuffer();
DnitCciAttribute attrFirstParserClassName;
attrFirstParserClassName.setName(szAttrName);
pContext->add(attrFirstParserClassName);
ctpSetAttribute(pContext,
                szAttrName,
                szAttrValue);
```

The default value of the 'firstParserClassName' attribute is 'XML'. This value causes the first part of the message at least, which is constructed by the input node and forwarded to the subsequent nodes, to have the parser name set to XML. So if a header in a format other than XML should be the first header of the generated message this parser class name has to be set to another value. Furthermore, to use parser class names other than XML on z/OS for WMQI Broker, including CSD3 or CSD4, the fix for WMQI Broker APAR PQ73602 must be installed.

A list of all parsers provided by WMQI Broker and their parser class names is available in the manual *WMQI Broker V2.1 Programming Guide* SC34-6170.

The example code shown below creates a message following this approach.

```
    MQMD           MQMDHeader = {MQMD_DEFAULT};
    MQRFH2         MQRFH2Header = {MQRFH2_DEFAULT};
    int            iMQMDHeaderSize = 0;
    int            iMQRFH2HeaderFixPartSize = 0;
    int            iComIbmDniFolderSize = 0;
    MQBYTE         completeMessage[1024];
    MQBYTE         ComIbmDniFolder[512];
    memset(completeMessage, 0, sizeof(completeMessage));
    memset(ComIbmDniFolder, 0, sizeof(ComIbmDniFolder));
    // Prepare MQMD Header
    MQMDHeader.MsgType = MQMT_DATAGRAM;
    // set Format to following header: MQRFH2
    memcpy((char*)MQMDHeader.Format,
        MQFMT_RF_HEADER_2,
        sizeof(MQMDHeader.Format));
    iMQMDHeaderSize=sizeof(MQMDHeader);
    // Move MQMD to completeMessage
    memcpy(completeMessage,
           &MQMDHeader,
           iMQMDHeaderSize);
    // Prepare MQRFH2 header
    // fixed part of MQRFH2
    MQRFH2Header.NameValueCCSID = 1200;
    iMQRFH2HeaderFixPartSize=sizeof(MQRFH2Header);
    // variable part of MQRFH2
    // -> generate ComIbmDni folder
    UnicodeString usComIbmDniFolder(DNIT_RFH2_COMIBMDNI_PART1);
    // append OU value
    usComIbmDniFolder.append(usAttrValue);
    usComIbmDniFolder.append(DNIT_RFH2_COMIBMDNI_PART2);
    // a NameValueData folders size must a multiple of 4
    // pad " " if necessary
    int lCntSpacesToPadDniFolder = (4 -
((usComIbmDniFolder.length())%4))%4;
    for (int i = 1; i <= lCntSpacesToPadDniFolder; i++)
        usComIbmDniFolder.append(" ");
    iComIbmDniFolderSize = (usComIbmDniFolder.length()*2);
    char* chComIbmDniFolder = new char[iComIbmDniFolderSize];
    lLen = usComIbmDniFolder.length();
    usComIbmDniFolder.extract(0, lLen, (UChar*)chComIbmDniFolder);
    // set ComIbmDni folder
    memcpy((char*)ComIbmDniFolder,
           (const char*)chComIbmDniFolder,
           iComIbmDniFolderSize);
    // set length of complete MQRFH2 folder
    MQRFH2Header.StrucLength = iMQRFH2HeaderFixPartSize
        + sizeof(iComIbmDniFolderSize)
        + iComIbmDniFolderSize;
    // move fixed part of MQRFH2 to completeMessage
    memcpy(completeMessage + iMQMDHeaderSize,
```

```
        &MQRFH2Header,
        iMQRFH2HeaderFixPartSize
       );
// move NameValueLength (iComIbmDniFolderSize) of
// MQRFH2 to completeMessage
memcpy(completeMessage
    + iMQMDHeaderSize
    + iMQRFH2HeaderFixPartSize,
    &iComIbmDniFolderSize,
    sizeof(iComIbmDniFolderSize));
// move NameValueData (ComIbmDniFolder) of MQRFH2
// to completeMessage
memcpy(completeMessage
    + iMQMDHeaderSize
    + iMQRFH2HeaderFixPartSize
    + sizeof(iComIbmDniFolderSize),
    &ComIbmDniFolder,
    iComIbmDniFolderSize);
// set Input Buffer
memcpy(inputBuffer,
    completeMessage,
    (iMQMDHeaderSize + MQRFH2Header.StrucLength));
cniSetInputBuffer(&iRC,
    pMessageIn,
    inputBuffer,
    (iMQMDHeaderSize + MQRFH2Header.StrucLength));
```

The Timer Input node uses the third method because of the automatic generation of the WMQI Broker Properties folder and because it requires fewer function calls than the first method.

The second method was not used because variable fields, such as the OU in the ComIbmDni folder, could not be filled by the compute note and this part of the message would have to be created by the Timer Input node.

The low number of additional function calls for creating the whole message required by the third method did not affect our decision to use this method. Furthermore, with this method one message tree copy between the Timer Input node and this compute node is saved.

### Transactional processing and thread handling

Transactional processing within the message flow is handled by the broker. If the return value of function cniRun() signals a

successful completion of the message flow to the broker, the broker commits the processing. The broker rolls back the processing of the message flow if the cniRun() function returns a corresponding return code or if the broker detects an unhandled exception.

The input node in a WMQI Broker message flow is responsible for the thread handling of this flow. A message flow can be processed by multiple threads. The 'Additional Instances' property is a flow property that can be set in the Control Centre in the Assignments panel. It defines the maximum number of threads that are available in addition to the minimum number of threads.

The minimum number of threads is defined by the number of input nodes in this message flow. The default of the Additional Instances property is zero, which means that there is only one thread available for each input node in the message flow. Any other number creates a thread pool with unused threads for this message flow. The thread processing for the message flow can request an additional thread from this thread pool. This is done by calling the utility function cniDispatchThread() once a message is available for processing. The new thread starts execution in the cniRun() plug-in implementation function for the same input node.

After processing the message the node can decide whether the current thread is to be returned to the thread pool or whether it is to remain.

For example, the Timer Input node uses the return value CCI_TIMEOUT if it has to wait until the next message is created. This return value is used to return control to the broker. There is no commit or rollback and the same thread receives control from the broker again in the cniRun() function of its input node.

If the message processing was successful in the subsequent nodes of the message flow, indicated by the fact that the utility function cniPropagate() returns without an error, the input node signals the broker to commit the transaction.

The node must decide whether the current thread should remain or be returned to the thread pool. This decision depends upon how often the input node and the message flow are used. If there is a

 

large load on the message flow it probably makes no sense to return the thread so CCI_SUCCESS_CONTINUE can be returned. This return value signals that the operation was successful and can be committed and that the thread remains.

For a complete list of all possible return codes and their meanings refer to the manual *WMQI Broker V2.1 Programming Guide* SC34-6170.

### Error handling

When processing problems occur a node usually throws a WMQI Broker exception. If the broker detects that an exception is thrown within a plug-in node it performs the same actions regardless of whether the node is a message processing node or an input node.

WMQI Broker adds an entry to the ExceptionList tree and propagates it, as well as sending a message to the appropriate error terminal. This error terminal is named ‘failure’ for a message processing plug-in node and for an input plug-in the name of the error terminal is ‘catch’. If this message is handled successfully, eg without any exception, the processing is committed.

If the input node does not have a catch terminal the broker detects this. Since the broker can’t propagate the error now, this is handled as if the input node indicates to roll back the processing. The same processing will also occur if the input node has a catch terminal but an exception occurs in the processing of this path or the catch terminal is not connected.

A node can propagate its message to a subsequent node on an output terminal usually named ‘out’. During the processing that follows there can also be problems and one of the subsequent nodes can throw an exception. Such a situation can be detected by the node by analysing the return code from the propagate call. A node usually rethrows this exception to its preceding node in the message flow.

If an input node detects an exception from a subsequent node connected to its out terminal and rethrows it, the exception, including the message, is also propagated to its catch terminal.

WMQI Broker adds an ExceptionList entry for the message and propagates it, along with the message, to the catch terminal.

There is a specific problem that requires a mention here, which is the possibility of an error occurring during message creation in the input node. During this procedure the processing of the node in which the exception is thrown is cleared.

Given that the purpose and function of this node is to create a message, the problem with this behaviour in the Timer Input node is that message creation would be rolled back. This means that because no message is available to the broker the output message at the catch terminal is empty. There is valid information only in the ExceptionList that appears at the catch terminal. Any message flow programmer has to handle this situation when wiring the catch terminal to handle errors.

Propagating empty messages to the catch terminal was undesirable for the design of the WBI for FN Timer Input node so the Timer Input node does not throw its recoverable exceptions. It builds the ExceptionList tree on its own and propagates the message together with this tree to the catch terminal. At the same time it writes the error to the WMQI Broker log. Only if severe system problems occur, eg insufficient memory, does it throw an exception without a message.

## CONCLUSION

As part of the development of WBI for FN a method was developed that can be used to process messages that result from external resources such as files and databases. This was done using the Timer plug-in node that regularly generates messages. During the processing of such Timer messages the data can be retrieved from these resources.

Writing an input node is in most cases similar to writing a message processing node, but additional thought must be given to threading, transactional behaviour, and error handling.

*Susan Herrmann and Michael Groetzner*
*IBM (Germany)*

# Global Unit of Work and two-phase commit

## INTRODUCTION

Syncpoint in WMQ supports both local and global Units of Work (UoW). A local UoW is one in which the only resources updated are those of the WMQ queue manager. A global UoW is one in which resources belonging to other resource managers, such as databases, are updated.

In a global UoW a two-phase commit procedure has to be executed by a transaction resource coordinator, such as the database, to maintain full integrity. The global UoW is committed by WMQ using **MQCMIT**; this initiates a two-phase commit of all the resource managers involved in the UoW.

A two-phase commit process is used whereby resource managers (for example, XA-compliant database managers such as DB2, Oracle, and Sybase) are first all asked to prepare to commit. Only if all are prepared are they then asked to commit. If any resource manager signals that it cannot commit, each is asked to back out instead. Alternatively, **MQBACK** can be used to roll back the updates of all the resource managers.

This article presents two programs. The first one uses WMQ as an XA resource coordinator with DB2 V7.2 and performs a two-phase commit; this is a C program. The second program takes as input an update statement and puts it in the queue. The program then executes the statement against the database along with a two-phase commit procedure; this is a Java program.

We have used these two programs in our environment to establish global units of transactions and two-phase commits, which are initiated from the Siebel CRM system and passed to a home-grown provisioning system based on DB2 via WMQ.

## WEBMQDB2.C

```
/* The following C Program demonstrates using WebSphere MQ as an XA  */
/* Resource coordinator with DB2 UDB Version 7.2 and has been tested */
/* using DB2 UDB Version 7.2 FP7 with MQSeries for Windows 2000 V5.2 */
/* service level U200148 (CSD03).                                    */
/* This program requires include files from both MQSeries and        */
/* DB2 UDB and requires linking utilcli.obj (from DB2 UDB) and       */
/* linking in both db2cli.lib and mqm.lib.                           */
/* Once built, this application can be executed against the TEST     */
/* database after the following updates have been made to db2cli.ini */
/*                  [test]                                           */
/*                  DBALIAS=test                                     */
/*                  CURSORHOLD=0                                     */
/*                  SYNCPOINT=2                                      */
/*                  CONNECTTYPE=2                                    */
/*                  AUTOCOMMIT=0                                     */
/* For additional information on the necessary configuration changes */
/* to WebSphere MQ, refer to the System Administration Guide.        */
#include <stdio.h>    /* standard include file */
#include <string.h>   /* standard include file */
#include <stdlib.h>   /* standard include file */
#include <sqlcli1.h>      /* DB2 include file      */
#include <cmqc.h>         /* MQSeries include file */
#include "utilcli.h"      /* DB2 include file      */
#define  MAX_STMT_LEN 255
#define  MAXCOLS   100
#ifndef max
#define  max(a,b) (a > b ? a : b)
#endif
/* Global Variables for user id and password. */
extern SQLCHAR server[SQL_MAX_DSN_LENGTH + 1] ;
extern SQLCHAR uid[MAX_UID_LENGTH + 1] ;
extern SQLCHAR pwd[MAX_PWD_LENGTH + 1] ;
/* Function declarations: */
int process_stmt( SQLHANDLE, SQLCHAR * ) ;
structure of the program is as below:
** main
** - initialize
** - start a transaction
**   - get statement
**   - another statement?
**   - COMMIT or ROLLBACK
** - another transaction?
** - terminate
int main( int argc, char * argv[] ) {
    FILE *fp;
    SQLHANDLE henv, hdbc, hstmt ;
    SQLCHAR         sqlstmt[MAX_STMT_LEN + 1] = "";
    SQLCHAR         sqltrans[sizeof("ROLLBACK")];
```

```
SQLRETURN       rc;
MQOD    od = {MQOD_DEFAULT};    /* Object Descriptor for reply   */
MQMD    md = {MQMD_DEFAULT};    /* Message Descriptor            */
MQPMO   pmo = {MQPMO_DEFAULT}; /* put message options           */
MQBO    bo = {MQBO_DEFAULT};    /* begin options                 */
MQHCONN  Hcon;                  /* connection handle             */
MQHOBJ   Hobj;                  /* object handle, server queue   */
MQLONG   O_options;             /* MQOPEN options                */
MQLONG   C_options;             /* MQCLOSE options               */
MQLONG   OpenCode;              /* completion code               */
MQLONG   CompCode;              /* completion code               */
MQLONG   Reason;                /* reason code                   */
MQLONG   CReason;               /* reason code (MQCONN)          */
MQLONG   BackCode;              /* Completion code for MQBACK    */
MQLONG   CmitCode;              /* Completion code for MQCMIT    */
MQLONG   BackReason;            /* reason code for MQBACK        */
MQLONG   CmitReason;            /* reason code for MQCMIT        */
MQLONG   BeginCode;             /* completion code for MQBEGIN   */
MQLONG   BeginReason;           /* reason code for MQBEGIN       */
MQBYTE   buffer[256];           /* message buffer                */
MQLONG   buflen;                /* buffer length                 */
char     QMName[5Ø];            /* queue manager name            */
printf("test mqadhoc start\n");
if (argc < 2) {
   printf("Missing parameter - logging queue\n");
   exit(99);
}
/* Connect to the Queue Manager that will be used. */
QMName[Ø] = Ø;    /* default */
if (argc > 2) {
   strcpy(QMName, argv[2]);
}
MQCONN(QMName, &Hcon, &CompCode, &CReason);
/* report reason and stop if it failed     */
if (CompCode == MQCC_FAILED) {
   printf("MQCONN ended with reason code %ld\n", CReason);
   exit(CReason);
}
/* Use first parameter as the name of the logging queue          */
strncpy(od.ObjectName, argv[1], (size_t)MQ_Q_NAME_LENGTH);
printf("Logging queue is %s\n", od.ObjectName);
/* Open the target message queue for output                      */
O_options = MQOO_OUTPUT + MQOO_FAIL_IF_QUIESCING;
MQOPEN(Hcon, &od, O_options, &Hobj, &OpenCode, &Reason);
/* report reason, if any; stop if failed      */
if (Reason != MQRC_NONE) {
   printf("MQOPEN ended with reason code %ld\n", Reason);
}
if (OpenCode == MQCC_FAILED) {
   printf("unable to open queue for output\n");
```

```
    }
    fp = stdin;
    CompCode = OpenCode;
    while (CompCode != MQCC_FAILED) {
       MQBEGIN(Hcon, &bo, &BeginCode, &BeginReason);
       if (BeginCode != MQCC_OK) {
          printf("MQBEGIN failed: Code = %ld, Reason = %ld\n",
                 BeginCode, BeginReason);
          CompCode = BeginCode;
       } else {
          /* allocate an environment handle */
          rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv )
;
          HANDLE_CHECK( SQL_HANDLE_ENV, henv, rc, &henv, &hdbc ) ;
          /* Allocate a connect handle, and connect.            */
          /* The userid and password are not specified since MQ */
          /* already connected to the DB with the MQBEGIN.      */
          rc = SQLAllocHandle( SQL_HANDLE_DBC, henv, &hdbc ) ;
          HANDLE_CHECK( SQL_HANDLE_ENV, henv, rc, &henv, &hdbc ) ;
          rc = SQLConnect( hdbc,
                           "test", SQL_NTS,
                           (SQLCHAR *)NULL, SQL_NTS,
                           (SQLCHAR *)NULL, SQL_NTS
                         ) ;
          HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, rc, &henv, &hdbc ) ;
          /* allocate statement handle and process statement */
          rc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt ) ;
          HANDLE_CHECK( SQL_HANDLE_STMT, henv, rc, &henv, &hdbc ) ;
          printf("Enter SQL command to process:\n");
          if (fgets(buffer, sizeof(buffer), fp) != NULL) {
             buflen = strlen(buffer);
             if (buffer[buflen-1] == '\n') {
                buffer[buflen-1] = '\Ø';
                --buflen;
             }
          } else {
             buflen = Ø;
          }
          if (buflen > Ø) {
             strcpy((char *)sqlstmt, buffer);
             if ( process_stmt( hstmt, sqlstmt ) == SQL_ERROR ) {
                CompCode = MQCC_FAILED;
             } else {
                memcpy(md.Format,  MQFMT_STRING,
                (size_t)MQ_FORMAT_LENGTH);
                memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId) );
                memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId) );
                pmo.Options = MQPMO_SYNCPOINT;
                MQPUT(Hcon, Hobj, &md, &pmo, buflen, buffer, &CompCode,
                &Reason);
```

```
            /* See if the MQPUT worked. If it failed, notify the   */
            /* user and set the transaction to be rolled back. If  */
            /* worked, ask user whether to commit or roll back.    */
              if (CompCode != MQCC_OK) {
                 printf("MQPUT failed: Code = %ld, Reason = %ld\n",
                        CompCode, Reason);
                 strcpy(buffer, "r");
              } else {
                 printf("Enter 'c' to COMMIT or 'r' to ROLLBACK the
                 transaction\n");
                 if (fgets(buffer, sizeof(buffer), fp) != NULL) {
                    buflen = strlen(buffer);
                    if (buffer[buflen-1] == '\n') {
                       buffer[buflen-1] = '\0';
                       --buflen;
                    }
                 }
              }

              /* See if the user said to commit or roll back   */
              /* the transaction and make appropriate MQ call. */
              if ((strcmp(buffer, "C") == 0) || (strcmp(buffer, "c")
== 0)) {
                 printf("Changes would be committed...\n");
                 MQCMIT(Hcon, &CmitCode, &CmitReason);
                 if (CmitReason != MQRC_NONE) {
                    printf("MQCMIT ended with reason code %ld\n",
                    CmitReason);
                 }
              } else {
                 printf("Changes would be rolled back...\n");
                 MQBACK(Hcon, &BackCode, &BackReason);
                 if (BackReason != MQRC_NONE) {
                    printf("MQBACK ended with reason code %ld\n",
                    BackReason);
                 }
              }
           }
        } else {
           CompCode = MQCC_FAILED;
        }
        /* Free statement handle                    */
        rc = SQLFreeHandle( SQL_HANDLE_STMT, hstmt ) ;
        HANDLE_CHECK( SQL_HANDLE_STMT, hdbc, rc, &henv, &hdbc ) ;
        /* Disconnect from database and release handle */
        rc = SQLDisconnect( hdbc ) ;
        HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, rc, &henv, &hdbc ) ;
        rc = SQLFreeHandle( SQL_HANDLE_DBC, hdbc ) ;
        HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, rc, &henv, &hdbc ) ;
        /* Free environment handle                  */
```

```
          rc = SQLFreeHandle( SQL_HANDLE_ENV,  henv ) ;
          ENV_HANDLE_CHECK( henv, rc ) ;
       }
    } /* end while */
    if (OpenCode != MQCC_FAILED) {
      C_options = 0;
      MQCLOSE(Hcon, &Hobj, C_options, &CompCode, &Reason);
      /* report reason, if any     */
      if (Reason != MQRC_NONE) {
         printf("MQCLOSE ended with reason code %ld\n", Reason);
      }
   }
   /* Disconnect from the Queue Manager if not already connected.  */
   if (CReason != MQRC_ALREADY_CONNECTED) {
      MQDISC(&Hcon, &CompCode, &Reason);
      /* report reason, if any     */
      if (Reason != MQRC_NONE) {
         printf("MQDISC ended with reason code %ld\n", Reason);
      }
   }
   return( SQL_SUCCESS ) ;
}                               /* end main */
/*--> SQLL1X63.SCRIPT */
** process_stmt
** - allocates a statement resources
** - executes the statement
** - determines the type of statement
**   - if there are no result columns, therefore non-select statement
**      - if rowcount > 0, assume statement was UPDATE, INSERT, DELETE
**     else
**      - assume a DDL, or Grant/Revoke statement
**   else
**      - must be a select statement.
**      - display results
** - frees the statement resources
int process_stmt( SQLHANDLE hstmt, SQLCHAR * sqlstr ) {
    SQLSMALLINT     nresultcols;
    SQLINTEGER      rowcount;
    SQLRETURN       rc;
    /* execute the SQL statement in "sqlstr"    */

    rc = SQLExecDirect(hstmt, sqlstr, SQL_NTS);
    if (rc != SQL_SUCCESS)
        if (rc == SQL_NO_DATA_FOUND) {
            printf("\nStatement executed without error, however,\n");
            printf("no data was found or modified\n");
            return (SQL_SUCCESS);
        }
        else /*indicate an error executing the statement*/
        STMT_HANDLE_CHECK( hstmt, rc);
```

```
     /*   printf("Error issuing SQLExecDirect %i",rc) ; */
    rc = SQLNumResultCols(hstmt, &nresultcols);
      STMT_HANDLE_CHECK( hstmt, rc);
    /* determine statement type */
    if (nresultcols == 0) {   /* statement is not a select statement */
        rc = SQLRowCount(hstmt, &rowcount);
     if (rowcount > 0) /* assume statement is UPDATE, INSERT, DELETE */
           printf("Statement executed, %ld rows affected\n", rowcount);
       else /* assume statement is GRANT, REVOKE or a DLL statement */
           printf( "Statement completed successful\n" ) ;
    }
    else StmtResultPrint( hstmt ) ; /* display the result set */
    /* end determine statement type */
    /* free statement resources */
    rc = SQLFreeStmt( hstmt, SQL_UNBIND ) ;
    STMT_HANDLE_CHECK( hstmt, rc);
    rc = SQLFreeStmt( hstmt, SQL_RESET_PARAMS ) ;
    rc = SQLFreeStmt( hstmt, SQL_CLOSE ) ;
    return( 0 ) ;
}                          /* end process_stmt */
/*<-- */
```

## MQDB2LOG.JAVA

```
/* Program name: mqdb2log                                            */
/* Description: java program that shows MQSeries base java and DB2   */
/*              jdbc transactions in the same unit of work (ie, two  */
/*              phase commit)                                        */
/* Function:                                                         */
/*  This program accepts a user's db update statement. This          */
/*  statement is then written to an MQ queue and executed against a  */
/*  database inthe same unit of work. The user is then asked whether */
/*  this unit of work should be committed or backed out.             */
/*  This program is run as follows:                                  */
/*    java mqdb2log -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 checked to confirm whether or not the    */
/* database update was committed or rolled back.                     */
/* This program has been tested with:                                */
/*    MQSeries V5.2 CSD 3                                            */
```

```
/*    JDK 1.3 that ships with WebSphere 4.01                          */
/*    DB2 V7.2                                                        */
/*    Windows 2000                                                    */
/* 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 MQ queue manager must be updated to recognize the database   */
/*   as a resource. This can be done by using the MQ 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 quite a few changes to the way things work with  */
/* Java. Details are supplied in the Using Java manual, Chapter 7.    */
/* Note 2: The MQSeries 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 mqdb2log {
   private MQQueueManager qMgr;
   private String         qmgrName;
   private String         queueName;
   private String         dbName;
   private Connection     jdbcConn;
   public static void main (String args[]) {
        mqdb2log mySample = new mqdb2log(args);
        mySample.start();
   }
   public mqdb2log(String[] args) {
        /* Get the command-line arguments */
        for( int i=0; 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 qmgr, 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 dbname,
                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 mqdb2log -q ... -m ... -d ...");
          System.out.println("where -q is the queue");
          System.out.println("      -m is the qmgr");
          System.out.println("      -d is the database name");
          System.exit(-1);
       }
}
/* This program doesn't have any specific initialization. */
/* If it did, it could go here.                           */
public void init() {
}
public void start() {
   try {
      System.out.println("mqdb2log 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("mqdb2log ready for db command");
      /* As long as the user keeps entering data, */
```

```
 /* process it...                              */
 do {
    runShow = br.readLine();
    /* See if the user entered anything */
    if (runShow.length() > 0) {
       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("  mqdb2log 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") >= 0)
              || (runShow.indexOf("C") >= 0) ) {
             qMgr.commit();
          } else {
             qMgr.backout();
          }
       } else {
          qMgr.backout();
       }
    }
    System.out.println("mqdb2log ready for db command");
 } while (runShow.length() > 0) ;
 /* 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("mqdb2log finished...");
   }
```

*Vikas Baruah*
*American Management Systems (USA)*


Although the articles published in *MQ Update* are of a very high standard, the vast majority are not written by professional writers, and we rely heavily on our readers themselves taking the time and trouble to share their experiences with others. Many have discovered that writing an article is not the daunting task that it might appear to be at first glance.

They have found that the effort needed to pass on valuable information to others is more than offset by our generous terms and conditions and the recognition they gain from their fellow professionals. Often, just a few hundred words are sufficient to describe a problem and the steps taken to solve it.

If you have ever experienced any difficulties with MQ, or made an interesting discovery, you could receive a cash payment, a free subscription to any of our *Updates*, or a credit against any of Xephon's wide range of products and services, simply by telling us all about it. For a copy of our *Notes for Contributors*, which explains the terms and conditions under which we publish articles, please point your browser at www.xephon.com/nfc.

Original Software, a provider of testing solutions for the IBM iSeries, has announced the release of TestMQ, a new module in its TestBench for iSeries product.

TestBench for iSeries is an automated testing solution that claims to capture and track all PC and server activity related to an application under test.

The TestMQ module uses API exits to capture and analyse the content and data of MQ messages as they are sent to or from MQ-enabled applications on the iSeries.This, claims the company, enables TestBench to build a detailed picture of an iSeries application under test, including automated tracking, testing, checking, and verification of all server processes, plus data extraction and maintenance, environment protection, and central script and results storage, as well as thorough testing of the PC processes.

*For more information contact:*
The Original Software Group, 2500 South Highland Avenue, Suite 20, Lombard, IL 60148, USA.
Tel: +1 630 268 1488.
Fax: +1 630 268 1499.
Web: http://www.origsoft.com

The Original Software Group, Grove House, Chineham Court, Basingstoke, RG24 8AG, UK.
Tel: +44 1256 338666.
Fax: +44 1256 338678.

* * *

Solstice Software has recently launched its Integra Enterprise 4.0 integration testing suite.

The company claims that the product enables visibility and control of the messaging backbone of mainframe integration, Web services, and other complex software integration projects.

Integra Enterprise's protocol library and focus on messaging is claimed to give project teams the ability to go 'behind-the-screens' and unearth problems buried in messages and files. The product suite consists of four main components: Automate, Simulate, Validate, and the Protocol Library.

The Protocol Library contains support for: WebSphere MQ, Tibco, webMethods, JMS, Http/XML, Https, TCP/IP, FTP, and SOAP/ WSDL, as well as Solstice proprietary protocols.

*For more information contact:*
Solstice Software (Formerly Class IQ), Brandywine Corporate Center, 650 Naamans Road, Suite 207, Claymont, Delaware 19703, USA.
Tel: +1 302 791 9900.
Fax: +1 302 791 0322.
Web: http://www.solsticesoftware.com

* * *