# 56

# MQ

# update

*February 2004*

## In this issue

# MQ Update


## Published by

Xephon Inc
PO Box 550547
Dallas, Texas 75355
USA

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

## Editor

Madeleine Hudson
E-mail: MadeleineH@xephon.com

## Publisher

Nicole Thomas
E-mail: nicole@xephon.com


## Disclaimer

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

## Subscriptions and back-issues

A year's subscription to *MQ Update*, comprising twelve monthly issues, costs $380.00 in the USA and Canada; £255.00 in the UK; $380.00 in the USA and Canada; £261.00 in Europe; £267.00 in Australasia and Japan; and £265.50 elsewhere. In all cases the price includes postage. Individual issues, starting with the July 1999 issue, are available separately to subscribers for $33.75 (£22.50) each including postage.

## Contributions

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

## *MQ Update* on-line

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




*Printed in England.*

# REXX utility for MQ on z/OS administration

Despite the fact that there are already a number of user-written tools in existence I felt the need to write another one; without it I would have gone mad! I'll describe a recent project to illustrate the problem I faced. Imagine you arrive at a customer's site and you're confronted with nearly 3,000 queues, all of which need to be categorized and analysed. Specifically, this was the situation:

- A large number of the queues (500) were left over from the TSO program CSQOREXX and the batch utility CSQUTIL.
- A significant number of queues were not in use, but the customer didn't know which ones these were.
- A small number (around 100) were to become 'shared' persistent queues.
- A certain number were to become cluster queues.
- The queues were shared amongst a number of projects and it was the customer's express wish that these queues had their own pagesets (!).
- The queues had to be categorized and spread across a set of new storage classes. These storage classes were to be based on how many messages were processed against a queue as well as its 'latency' – a measure of how long individual messages stayed on a queue.
- The existing queue manager was to be 'cloned', with the clone running on a separate LPAR within the sysplex. This clone, however, wasn't an exact copy of the main one.

So, how was I to tackle a project like this? The main problem with ISPF panels is that they consume a lot of time and CPU resources and you can select on only a limited number of attributes. You really need to use a batch-like utility to list queues and select specific attributes. CSQUTIL is quite good at this but with so many queues it doesn't provide an ideal solution.

The REXX program listed at the end of this article takes the output from the CSQUTIL utility and selectively prints queues, depending on the attributes you've specified. It is based on the premise that what is printed is displayed as a group of lines, typically like this:

- MAIN line containing the item selected
  - ATTRIBUTES belonging to the main item.
- Next MAIN item
  - ATTRIBUTES belonging to this item.

Here is an example from CSQUTIL as a result of a **dis q(*) all** command:

```
CSQM402I ?PR2P
  QUEUE(BBOMVAO.COMMAND.REPLY.MODEL)
  TYPE(QMODEL)
  QSGDISP(QMGR)
  STGCLASS(DEFAULT)
CSQM401I ?PR2P
  QUEUE(BBSMVMQS.RR.PR2P)
  TYPE(QLOCAL)
  QSGDISP(QMGR)
  STGCLASS(DEFAULT)
```

## REQUIREMENTS

- Run CSQUTIL against the relevant queue manager with a command (eg **dis q(*) all**) and save the output as a file.
- Supply the following files:
  - DD statement FILEIN containing the file from step 1
  - DD statement MAIN containing a single keyword to scan for
  - DD statement PARMS containing zero or more attributes to scan for.

I have supplied some example files (allques.txt, main.txt, parms.txt, and qstatus.txt), which can be found at www.xephon.com/extras.

- Run, and check the output. It has successfully run on Windows in both IBM Object REXX and Reginald REXX environments.

## EXAMPLE REPORTS

The following examples show what you can specify and the results obtained. Obviously only a very small subset of queue names could be printed here.

```
MAIN keyword : QUEUE
PARMS keywords: not supplied
*** Main search key requested  :  QUEUE
BBOMVAO.COMMAND.REPLY.MODEL
BBOMVAO.EXEC.RR.SSØ2.PR2P
BBSMVMQS.RR.PR2P
CISSJ.BASE.BLOB.IMTDMSGID
TARGET.QUEUE2
PR2PØ.PJKULPØ2
...
TTTT
ZZTEMP
*** No. of items found  :  2448
*** No. of items ignored:  Ø
Started at  18:23:24  Ended at  18:24:13
```

The result of this is a complete list of all 2,448 queue names.

```
MAIN keyword : QUEUE
PARMS keyword: CURDEPTH* - The * means 'display the value'.
*** Main search key requested  :  QUEUE
*** Attribute Requested :  CURDEPTH*
BBOMVAO.EXEC.RR.SSØ2.PR2P  (Ø)
BBSMVMQS.RR.PR2P  (Ø)
CISSJ.BASE.BLOB.IMTDMSGID  (Ø)
...
TTTT  (Ø)
ZZTEMP  (Ø)
*** No. of items found  :  2391
*** No. of items ignored:  57
Started at  18:26:56  Ended at  18:28:Ø1
```

The result of this is 2,391 queues displayed with the value of CURDEPTH shown in brackets; the other queues ('ignored') were probably model and remote queues. Note that the total is still 2,448.

```
MAIN keyword :    QUEUE
PARMS keyword:    CURDEPTH(<>Ø)
         MAXDEPTH*
** Main search key requested  :  QUEUE
*** Attribute Requested :  CURDEPTH(<>Ø)
*** Attribute Requested :  MAXDEPTH*
```

```
LCDEF.GSJJUTER.LF1100.LOG.FGFG  (1)  (3000)
LCDEF.LCDEF.LF1100.WWV.FGFG  (23229)  (100000)
...
SYSTEM.CHANNEL.SYNCQ  (63)  (999999999)
SYSTEM.CLUSTER.REPOSITORY.QUEUE  (2)  (999999999)
*** No. of items found  :  347
*** No. of items ignored:  2101
Started at  18:28:59  Ended at  18:30:33
```

The result of this is 347 queues found, showing their non-zero queue CURDEPTH and the value of MAXDEPTH.

```
MAIN keyword :    QUEUE
PARMS keyword:    CURDEPTH(<>0)
         MAXDEPTH*
         INDXTYPE(MSGID)
*** Main search key requested  :  QUEUE
*** Attribute Requested :  CURDEPTH(<>0)
*** Attribute Requested :  MAXDEPTH*
*** Attribute Requested :  INDXTYPE(MSGID)
LSSB.LSSB.LF1100.SRSWTRZADTYFGFG  (9)  (50000)
LSSFF.LSSFF.LF1100.SRSWTRZADTYFGFG  (6)  (50000)
...
PR2P.00.KVS.DZ.KV101AU  (2851)  (50000)
SYSTEM.CHANNEL.SYNCQ  (63)  (999999999)
*** No. of items found  :  183
*** No. of items ignored:  2265
Started at  18:40:15  Ended at  18:42:05
```

The result of this is 183 queues with non-zero CURDEPTH, the MAXDEPTH whose INDXTYPE was set to MSGID. Of course, the utility isn't restricted to the above command. Issue the command **reset qstats(*)** and run with these values :

```
MAIN keyword :    QSTATS
PARMS keyword:    HIQDEPTH(<>0)
*** Main search key requested  :  QSTATS
*** Attribute Requested :  HIQDEPTH(<>0)
BBOMVAO.CLXCLC.REPLY.SS02.PR2P  (3)
GB.WDR.GT_WARM  (22)
...
SYSTEM.CSQOREXX.B96D732661689542  (5)
SYSTEM.CSQOREXX.B96D781DAFCLDC807  (3)
*** No. of items found  :  633
*** No. of items ignored:  1743
Started at  19:00:46  Ended at  19:01:02
```

The result of this is 633 queues with an HIQDEPTH greater than zero. This is useful when deciding which queues are 'in use'; if run

on a regular basis you may be able to delete those ‘unused’queues. In fact any file with the same ‘structure’ can be analysed.

## REXX SOURCE CODE

The original program used the EXECIO keyword, which is available to REXX on the IBM mainframe, but I also wanted to be able to run it on my PC so it was replaced with the LINEIN and LINEOUT keywords. The advantage is that, the file having been created on the mainframe, could easily be transferred to the PC and analysed via REXX, thereby offloading processing.

You may be surprised to learn that running CSQUTIL with a **dis q(*) all** command on 2,448 queues produced nearly 114,000 lines of output (one of the example files), which will take a fair amount of processing. Running this on the PC may take between 1and 5 minutes, which may well be faster than on the mainframe, where a job may have to wait before it’s even processed.

Producing a list of queues with the required parameters will be useful. What may be even more useful is to act on that set and automatically generate a set of commands.

This is achieved by setting the switch **gen_commands** to 1 and reviewing the internal procedure **proc_util_commands**. I suggest you always check the file of commands generated. Naturally, it will take longer to process.

For those of you who are real experts in REXX I’m sure there are improvements to be made, but it’s the ideas that are important here.

```
/**************** REXX ***********************************************/
/* Author   : Ruud van Zundert                                      */
/* Date     : November 2003                                         */
/* Function : To ease the burden of MQ administration on Z/OS.      */
/*            The main problem with the CSQUTIL output is that      */
/*            it lists things over more than one line making it     */
/*            difficult and time consuming to extract specific      */
/*            queues, eg all queues with non-zero queue depth.      */
/* Input    : 3 files                                               */
/* File 1     DD statement FILEIN is the input file usually         */
/*            the output from the CSQUTIL utility.                  */
```

```
/* File 2   : DD statement MAIN  contains the main keyword to    */
/*            scan for.                                          */
/* File 3   : DD statement PARMS contains extra keyword(s) to    */
/*            scan for. More than one keyword can be given. e.g. */
/*            INDXTYPE(MSGID) lists only those with this value   */
/*            CURDEPTH list queues with CURDEPTH but does not    */
/*              display the actual CURDEPTH                      */
/*            CURDEPTH* also displays the actual depth           */
/*            CURDEPTH(Ø) only those with this value             */
/*            CURDEPTH(<>Ø) only those with CURDEPTH NOT Ø       */
/* Output   : If the 'gen_commands' switch is 'on' then it will  */
/*            generate additional commands to administer the     */
/*            queues selected. Alter the internal procedure      */
/*            called 'proc_util_commands' accordingly.           */
sttime       = TIME()              /* store start time             */
gen_commands = Ø                   /* 1=generate commands          */
/*inptfile = 'C:/qstatus.txt'*/
inptfile = 'C:/rexxfiles/allques.txt'
mainfile = 'C:/rexxfiles/main.txt'
parmfile = 'C:/rexxfiles/parms.txt'
match_count  = Ø
ignore_count = Ø
call proc_readfiles
DO ix1 = 1 TO parmi.Ø               /* loop round all parameters     */
  parms_showkey.ix1 = Ø             /* initialize                    */
  parms_found.ix1 = Ø
  parms_ignore.ix1 = ''
  parms_ignore_found.ix1 = Ø
  PARSE UPPER VAR parmi.ix1 parms_search.ix1   /* find attribute     */
  parms_search.ix1 = STRIP(parms_search.ix1)
  SAY '*** Attribute Requested : ' parms_search.ix1
  /* If a parameter has an * behind it, it means display its value   */
  IF POS('*',parms_search.ix1) > Ø THEN parms_showkey.ix1 = 1
  parms_search.ix1 = STRIP(parms_search.ix1,,'*')
  /* if a parameter has <> specified, it means only display if the   */
  /* value is NOT the one specified                                  */
  IF POS('<>',parms_search.ix1) > Ø THEN  /* special search          */
    DO
      x = POS('(',parms_search.ix1)
      y = POS(')',parms_search.ix1)
      parms_showkey.ix1 = 1
      wlit2  = parms_search.ix1
      parms_search.ix1 = SUBSTR(parms_search.ix1,1,x-1)
      parms_ignore.ix1 = SUBSTR(wlit2,1,x) || SUBSTR(wlit2,x+3,y-x-2)
    END
END
main_found = Ø
IX1 = Ø
/* Open the file for reading, but only if it exists */
IF STREAM(inptfile, 'C', 'OPEN READ') == 'READY:' THEN DO
```

```
  OPTIONS 'FAST_LINES_BIF_DEFAULT'
  SIGNAL ON NOTREADY NAME MainFileError
END /* STREAM() */
wline = LINEIN(inptfile, , 1)       /* read first line              */
DO WHILE LINES(inptfile) > Ø
   IX1 = IX1 + 1                    /* increment read count         */
   wpos = POS(mainkey'(',wline,1)   /* main key in record read?     */
   IF wpos > Ø THEN DO              /* yes - process it             */
     CALL proc_mainkey
   END
   ELSE DO                          /* no -look for other matches   */
     CALL proc_other
   END
   wline = LINEIN(inptfile, , 1)    /* read next line               */
END
MainFileError:
   /* Close the file */
   CALL STREAM inptfile, 'C', 'CLOSE'
CALL proc_output
SAY '*** No. of items found  : ' match_count
SAY '*** No. of items ignored: ' ignore_count
SAY 'Started at ' sttime ' Ended at ' TIME()
SAY '*** End of REXX scan procedure ***'
EXIT                                /* end of program               */
/* Read the input file(s) */
proc_readfiles:
lineno = Ø
mainkey = LINEIN(mainfile)           /* read single main keyword    */
mainkey = STRIP(mainkey)             /* remove blanks               */
SAY '*** Main search key requested  : ' mainkey
/* Open the file for reading, but only if it exists                 */
IF STREAM(parmfile, 'C', 'OPEN READ') == 'READY:' THEN DO
   /* Use LINES() == 1 to indicate more lines                       */
   OPTIONS 'FAST_LINES_BIF_DEFAULT'
   /* Set up for any errors                                         */
   SIGNAL ON NOTREADY NAME FileError
   /* Any more lines? */
   DO WHILE LINES(parmfile) > Ø
      i = lineno + 1
      /* Read the next line */
      parmi.i = LINEIN(parmfile, , 1)   /* store in stem array      */
      /* Increment line count now that it is successfully read     */
      lineno = i
   END /* WHILE LINES()                                            */
   parmi.Ø = lineno                 /* store total lines            */
FileError:
   /* Close the file */
   CALL STREAM parmfile, 'C', 'CLOSE'
END /* STREAM() */
RETURN
```

```
proc_mainkey:                           /* Process the main key      */
 wpos2 = POS(')',wline)                 /* find the end bracket      */
 IF wpos2 > Ø THEN
   DO
     CALL proc_output
     main_found = 1
     LS = wpos+LENGTH(mainkey)
     wQname = SUBSTR(wline,LS+1,wpos2-LS-1) /* get queue name        */
     DO ix2 = 1 TO parmi.Ø              /* init parm switches        */
       wreq.ix2 = ''
       parms_found.ix2 = Ø
       parms_ignore_found.ix2 = Ø
     END
   END
RETURN
proc_other:                             /* Process additional attr   */
 DO ix2 = 1 TO parmi.Ø                  /* loop round each attribute */
   IF parms_ignore.ix2 <> '' THEN       /* are any to be ignored?    */
     DO
       wpos3 = POS(parms_ignore.ix2,wline)
       IF wpos3 > Ø THEN DO             /* found item to be ignored  */
         parms_ignore_found.ix2 = 1     /* set switch so that it is  */
         RETURN                         /* not printed later on.     */
       END
     END
/* Cater for search string already having a bracket */
   IF POS('(',parms_search.ix2) > Ø THEN
     wpos = POS(' 'parms_search.ix2,wline)
   ELSE
     wpos = POS(' 'parms_search.ix2'(',wline)
   IF wpos > Ø THEN
     DO
       parms_found.ix2 = 1
       /* found key - do we also want to show the key value? */
       IF parms_showkey.ix2 > Ø THEN
         DO
           wpos2 = POS(')',wline)
           IF wpos2 > Ø THEN
             DO
               LS = wpos+LENGTH(parms_search.ix2)
               wreq.ix2 = SUBSTR(wline,LS+1,wpos2-LS+1)
             END
         END
     END
 END
RETURN
proc_output:                    /* Display output                    */
 wResult = ''                   /* blank out print line              */
 wPrint = 1                     /* assume we're printing             */
 IF main_found = 1 THEN DO
```

```
   /* Found main key but only display stuff if it is eligible        */
   DO ix2 = 1 to parmi.Ø WHILE wPrint = 1
     IF parms_ignore_found.ix2 = 1 THEN wPrint = Ø
     IF parms_found.ix2 = Ø THEN wPrint = Ø
     wResult = wResult' 'wreq.ix2       /* append to display line    */
   END
   IF wPrint = 1 THEN DO                /* Display still required    */
     match_count = match_count + 1      /* increment match count     */
     SAY wQname wResult
   IF gen_commands THEN call proc_util_commands
   END
   ELSE                                 /* Display not required      */
     ignore_count = ignore_count + 1    /* increment ignore count    */
 END
RETURN
proc_util_commands:                     /* generate new commands     */
outfile1 = 'C:/rexxfiles/outq1.txt'
outfile2 = 'C:/rexxfiles/outq2.txt'
outfile3 = 'C:/rexxfiles/outq3.txt'
outfile4 = 'C:/rexxfiles/outq4.txt'
IF match_count = 1 THEN DO          /* clear the files at the start */
  call SysFileDelete outfile1
  call SysFileDelete outfile2
  call SysFileDelete outfile3
  call SysFileDelete outfile4
END
/* The following are some example commands you may want to use if    */
/* there are a set of queues that need to have their storage class   */
/* altering but if there are messages in the queue this is not       */
/* possible. The commands below will achieve that change.            */
cmdline = 'DEF QL('wQname'2) +'
LINEOUT(outfile1,cmdline)
cmdline = '  LIKE('wQname')'
LINEOUT(outfile1,cmdline)
cmdline = 'MOVE QL('wQname') +'
LINEOUT(outfile2,cmdline)
cmdline = ' TOQLOCAL('wQname'2)'
LINEOUT(outfile2,cmdline)
cmdline = 'ALTER QL('wQname') +'
LINEOUT(outfile3,cmdline)
cmdline = ' STGCLASS(NEWSTG1)'
LINEOUT(outfile3,cmdline)
cmdline = 'MOVE QL('wQname'2) +'
LINEOUT(outfile4,cmdline)
cmdline = ' TOQLOCAL('wQname')'
LINEOUT(outfile4,cmdline)
RETURN
```

*Ruud van Zundert (UK)*
*ruudvz@btclick.com*

# Soap WMQ transport

This article has been written from the perspective of a developer whose task was to create an alternative transport for Web services. It describes how to create such transport for Web services through WMQ, provided as an implementation of the IsoapTransport interface of the Web Services Enhancements (WSE 2.0). Correct implementation of the transport allows developers to incorporate the security, routing, and attachment features of WSE in their applications without incurring additional costs.

## WHY TRANSPORT THROUGH WMQ?

There are at least two answers to this question. The first is to satisfy the requirement to communicate with a service through WMQ (ie a legacy application). Another reason is to provide reliable delivery of messages.

One requirement of a particular project we recently worked on was to create a Web service with secure and reliable delivery and processing of messages. WSE 2.0 is claimed to support a message-oriented programming model, allowing the implemention of peer-to-peer programs or event-driven applications. Web services that leverage WSE can be hosted in multiple environments, including ASP.NET, standalone executables, NT Services etc, and can communicate over alternative transports. So we decided to implement our custom transport based on WMQ.

## IMPLEMENTATION OF WMQ TRANSPORT

Previously, in order to implement transport through WMQ you had to implement the IsoapTransport interface (which is quite small) and then register it in WSE; however, there have been changes to its implementation, including communication with WMQ, transaction support, configuration options, and other useful features.

In this article we examine the details of implementing WMQ transport. Each transport channel implementation contains a client application and a server application; so too does our implementation. You can find a description of the demo solution that includes these applications at the end of this article. It demonstrates how to use WMQ transport in a real task and its sources can be downloaded from www.xephon.com/MQTransport_src.zip.

SoapWMQTransport utilizes IBM's MA7P SupportPac, which implements WMQ classes for Microsoft .NET. Actually this SupportPac is only a .NET-compatible wrapper for an IBM library that provides native access to WMQ.

The WMQ transport implementation consists of the following parts:

- Transport – defines the means for transferring Soap messages between the client and the server.
- Channel – defines a channel that uses WMQ to transfer particular messages.
- Listener – allows the server to accept WMQ requests.
- Transactions – supports transactions.
- Configuration – allows binding of Web services to specified channels, using a configuration file.
- Performance counters – allows WMQ transport activity to be monitored.
- Installation classes – allows the WMQ transport library to be used as a separate installation unit or as a part of a bigger installation.

## TRANSPORT

The SoapWMQTransport class encapsulates details of the interaction with WMQ. It is inherited from the auxiliary SoapTransport class and implements the ISoapTransport interface.

Instances of this class are used by the infrastructure only; there is no need for users to deal directly with them. In order to use an alternative transport you need to implement the following methods and properties of the ISoapTransport interface:

- Scheme property.
- RegisterPort method.
- UnregisterAll method.
- UnregisterPort method.
- Send method.
- BeginSend method.
- EndSend method.

### Scheme property

The scheme property returns a scheme prefix. SoapWMQTransport uses the URI scheme provided by the property to identify Web services endpoints. An example of such an endpoint is: 'soap.WMQ:limits-service'. 'Soap.WMQ' refers to a transport protocol, as do http and ftp. The presence of the 'Soap.WMQ' scheme in URI determines that WSE will use WMQ-based transport. The service name 'limits-service', which appears after the scheme name, is used on the server-side to identify a targeted Web service, distinguishing it from other Web services that may be available at this server.

### RegisterPort method

The RegisterPort method binds the URI service to the service handler. This method is called on the server indirectly by the infrastructure. Use the configuration file to bind the service to Soap WMQ transport. For details of how to create and tune the configuration file look at the comments in the app.config file of the SoapWMQTransport project. You can also bind the URI to the service handler from your application, eg:

```
SoapWMQConfiguration.Initialize();
SoapWMQServiceBinding binding = new SoapWMQServiceBinding();
// Fill instance of binding here ...
SoapWMQConfiguration.Bindings.Add(binding);
```

After service binding is registered the server starts a listening channel associated with the service.

### UnregisterAll and UnregisterPort methods

UnregisterAll and UnregisterPort methods perform opposite functions to the RegisterPort method. Use UnregisterAll to release all listeners so requests will no longer be accepted. The UnregisterPort method is used to stop requests being accepted for a specified service. To remove transport binding for a service or all services manually, use the configuration classes shown in the following example, rather than the UnregisterPort and UnregisterAll methods.

```
// Removes service with name "limits-service".
SoapWMQConfiguration.Bindings.Remove("limits-service");
// Removes all services.
SoapWMQConfiguration.Bindings.Clear();
```

Usually there is no need to remove transport bindings manually.

### Send

Send is called by the infrastructure to send data to a server through the transport. Actually, the client never uses this method directly. A descendent of the SoapClient class (service proxy) usually invokes and passes data to the server:

```
/// <summary>
/// Defines proxy for limits requests.
/// </summary>
[SoapService(Message.DEMO_NS)]
public class LimitsProxy: SoapClient
{
/// <summary>Requests limits.</summary>
/// <param name="request">determines a request message.</param>
/// <returns>response as a LimitResponse instance.</returns>
[SoapMethod("limits")]
public LimitResponse GetLimits(LimitRequest request)
{
return (LimitResponse)SendRequestResponse("GetLimits", request).
```

```
GetBodyObject(typeof(LimitResponse));
}
}
```

Send creates an instance of the transport channel (SoapWMQChannel), which performs real work. It communicates with WMQ (fills the WMQMessage, puts into the appropriate output queue, and gets responses from the input queue). If a response is expected, Send calls the OnReceive private method to get a response.

### BeginSend and EndSend methods

These methods are used by the WSE infrastructure for calling the Send method asynchronously. Below, we describe several private methods that are not part of the ISoapTransport interface, but which are often used to enhance functionality.

### DoReceive method

DoReceive is a private method, which is used as an incoming request handler on the server side. DoReceive generally calls the OnReceive method; however, if it discovers that the operation is transactional it wraps an OnReceive invocation into the transaction.

### OnReceive method

OnReceive is a private method that converts a WMQ message into the soap envelope and then calls the WSE service dispatcher to process a request (or response). The following code sample shows how the Send, OnReceive, and DoReceive methods collaborate.

```
class SoapWMQTransport
{
void ISoapTransport.Send(SoapEnvelope envelope, Uri to)
{
SoapWMQChannel channel = GetChannel(to,envelope.Context.MessageID);
try
{
channel.Send(envelope);
if (channel.MessageType == WMQC.WMQMT_REPLY)
OnReceive(channel);
}
```

```
finally
{
channel.Close();
}
}
```

As you see, SoapWMQTransport recognizes request/response messages by their message type. When the client sends a request message the OnReceive method is called. Since a SoapWMQChannel instance encapsulates real communication with WMQ it is important to note that the channel is closed each time a message is processed. This happens because the Send method could be called from different threads and you cannot create a connection to an WMQ queue manager in one thread and use it in another.

```
/// <summary>This auxiliary class is used to call methods in
transactional context.</summary>
private class ReceiveProcessor
{
internal SoapWMQListener listener;
internal WMQMessage message;
internal SoapWMQServiceBinding binding;
internal SoapWMQChannelDescription descr;
internal SoapWMQTransport transport;
internal void Process()
{
SoapWMQChannel channel = transport.GetChannel(descr, message,
binding.Transactional);
try
{
//...
transport.OnReceive(channel);
if (binding.Transactional)
{
// at this point we decide to commit or rollback transaction
// depending on channel state
if (channel.IsFault)
ContextUtil.SetAbort();
else
ContextUtil.SetComplete();
}
// ...
}
finally
{
channel.Close();
}
```

```
}
}
private void DoReceive(SoapWMQListener listener,
SoapWMQServiceBinding binding, WMQMessage message)
{
SoapWMQChannelDescription descr =
SoapWMQConfiguration.Channels[binding.Channel];
ReceiveProcessor processor = new ReceiveProcessor();
processor.message = message;
processor.binding = binding;
processor.descr = descr;
processor.transport = this;
processor.listener = listener;
if (binding.Transactional)
{
// if we need to perform a transaction we create for this purpose
// COM+ object that provides transactional context and supports
// distributed transaction
using(SoapWMQDTCTransaction context = new SoapWMQDTCTransaction())
{
context.Process(new SoapWMQDTCMethod(processor.Process));
}
}
else
{
processor.Process();
}
}
private void OnReceive(SoapWMQChannel channel)
{
SoapEnvelope envelope = channel.Receive();
if (envelope != null)
{
// at this point we delegate control to WSE
SoapReceiver.DispatchMessage(envelope);
         }
     }
}
```

ReceiveProcessor is an auxiliary class that is used to support transactions. The DoReceive method is called by the infrastructure on the server-side only, whereas OnReceive is called by WSE on both sides.

## Channel

Channel encapsulates WMQ-related aspects of communication. It sends and receives WMQ messages and converts them to soap

envelopes. The main communication with WMQ is carried on inside this class. Two crucial methods are Receive and Send:

```
/// <summary>
/// Creates instance of soap channel through WMQ.
/// </summary>
public class SoapWMQChannel: SoapChannel
{
private WMQMessage message; // message is initialized in
constructor
// ...
/// <summary>
/// Receives message from the input queue.
/// </summary>
/// <returns>Soap envelope.</returns>
public override SoapEnvelope Receive()
{
WMQGetMessageOptions options=new WMQGetMessageOptions();
switch(message.MessageType)
{
case WMQC.WMQMT_REPLY: // client received response from server.
options.Options = WMQC.WMQGMO_WAIT | WMQC.WMQGMO_NO_SYNCPOINT;
options.MatchOptions = WMQC.WMQMO_MATCH_CORREL_ID;
options.WaitInterval = timeout;
break;
case WMQC.WMQMT_REQUEST: // server received request from client.
case WMQC.WMQMT_DATAGRAM:
options.Options=WMQC.WMQGMO_NO_WAIT |
(IsTransactional ? WMQC.WMQGMO_SYNCPOINT :
WMQC.WMQGMO_NO_SYNCPOINT);
options.MatchOptions = WMQC.WMQMO_MATCH_MSG_ID;
break;
}
inQ.Get(message, options);
SoapEnvelope envelope;
using (SoapWMQStream stream = new SoapWMQStream(message))
envelope = DeserializeMessage(stream);
if (message.MessageType == WMQC.WMQMT_REQUEST)
{
message.MessageType = WMQC.WMQMT_REPLY;
message.CorrelationId = message.MessageId;
}
if (envelope != null)
envelope.Context.Channel = this;
return envelope;
}
/// <summary>
/// Send the specified SOAP envelope to the output queue.
/// </summary>
/// <param name="envelope">Determines the envelope.</param>
```

```
public override void Send(SoapEnvelope envelope)
{
using(SoapWMQStream stream = new SoapWMQStream(message))
SerializeMessage(envelope, stream);
Timestamp timestamp = envelope.Context.Timestamp;
if (timestamp != null)
{
long ttl = timestamp.Ttl;
if (ttl > Ø)
message.Expiry = (int)(ttl / 1ØØ);
}
switch(message.MessageType)
{
case WMQC.WMQMT_REQUEST: // Send request from client
message.ReplyToQueueName = inQ.Name;
message.Report=WMQC.WMQRO_COPY_MSG_ID_TO_CORREL_ID;
outQ.Put(message);
message.MessageType = WMQC.WMQMT_REPLY;
message.CorrelationId = message.MessageId;
break;
case WMQC.WMQMT_DATAGRAM: // Send one way request from client
case WMQC.WMQMT_REPLY: // Send response from server
outQ.Put(message);
break;
}
}
}
```

### *Receive method*

Receive is called by the transport to receive a request or response and deserialize it into a soap envelope. This method is the starting point for distributed transactions. On a server this method is triggered by a listener, which monitors WMQ. On a client, Receive is invoked after sending a message and when WSE expects a response.

### *Send method*

Send serializes the soap envelope and sends a request to a client or a response to a server. It also manages the expiration time of messages. The Send method delivers a one-way message and, therefore, doesn't return anything to the caller. However, if the sender wants a response from the receiver it has to implement a SoapReceiver of its own and indicate where the original receiver should send the response messages.

## Listener

SoapWMQListener monitors incoming messages. Listener is used only on the server side. For each input queue on the server side there is a listener object that monitors it. The methods of this class are Start, Browse, Listen, and Receive event.

### *Start method*

When the transport needs to monitor a specific queue it creates an instance of SoapWMQListener and calls the Start method over this listener. The method creates a new monitoring thread.

### *Browse method*

Browse waits for incoming messages; usually it's sleeping.

### *Listen method*

Listen performs a thread cycle, which calls Browse, and, if there is an incoming message, gets a service binding for the message and passes the message to a handler (SoapWMQTransport.DoReceive).

### *Receive event*

This is an event handler that the transport attaches when a SoapWMQListener instance is created.

Now it's time to define how the channel and listener identify a destination service. For identification purposes the message-ID and correlation-ID WMQ message fields are used.

- When the client sends a request to a server a new message identification is generated and assigned to the message-ID. The correlation-ID is given the value of a service-ID, which is defined in the configuration.
- When the server accepts a request from a client the correlation-ID of the message is used to identify the service handler. The message-ID is used to identify the client that has sent the request.

- When the server sends a response to the client the server uses the message-ID of the request message, which identifies the client and puts it into the correlation-ID of the response message.
- When the client gets a response from the server it selects a message with the correlation-ID that is equal to the message-ID of the request message.

This technique avoids having to get a whole message from a queue in order to identify its destination. It's especially important for transactions, where the transaction needs to be started before a message is got from the queue. Use the configuration file to define the service binding or SoapWMQConfiguration to bind the service manually:

```
<configuration>
  <configSections>
    <section name="nesterovsky-bros.web.services.WMQ"
type="NesterovskyBros.Web.Services.Messaging.Configuration.SoapWMQConfiguration,
SoapWMQTransport" />
  </configSections>
  <nesterovsky-bros.web.services.WMQ>
    ...
    <!--
     Define limit service. This services uses following channel
name: "server-channel".
     Service handler is:
"NesterovskyBros.WMQTransport.Demo.Server.LimitsService".
     As a correlationID of outgoing messages the following id will
be used:
"Ø9cbbe5f-a8d2-4f8e-99fa-c53d7347afa1". Actually this id is used
to bind messages
to the service.
    -->
    <service name="limits-service"
       channel="server-channel"
  type="NesterovskyBros.WMQTransport.Demo.Server.LimitsService,
Server"
  id="Ø9cbbe5f-a8d2-4f8e-99fa-c53d7347afa1" />
  </nesterovsky-bros.web.services.WMQ>
</configuration>
```

See the app.config files of the Client and Server projects for more details of how to configure channels on the client and server-side respectively.

## TRANSACTIONS

Our implementation of WMQ transport supports distributed transactions. This means that the whole process of getting a message from the queue on the server, processing it, and sending a response, may be considered as a single unit of work, which either succeeds or is rolled back. The Distributed Transaction Coordinator (DTC) is used to manage transactions. See SoapWMQTransport.DoReceive method above for details of how the transport deals with transactions.

To configure a service to use transactions use the 'transactional' attribute in the 'service' element of the transport configuration, or configure a service binding manually:

```
<configuration>
  <configSections>
    <section name="nesterovsky-bros.web.services.WMQ"
type="NesterovskyBros.Web.Services.Messaging.Configuration.SoapWMQConfiguration,
SoapWMQTransport" />
      </configSections>
      <nesterovsky-bros.web.services.WMQ>
    ...
    <service name="charge-service"
      channel="server-channel"
      type="NesterovskyBros.WMQTransport.Demo.Server.ChargeService,
          Server"
      id="8af1d7fe-4f63-4c64-937c-1a5ffd497fc5"
      transactional="true"/>
  </nesterovsky-bros.web.services.WMQ>
</configuration>
```

It is important to note the following:

- A 'transactional' attribute has meaning only for the server, not for the client application.
- WMQ supports distributed transactions for server connections only (binding connection to a local queue manager), not for remote connections.

## CONFIGURATION

Soap WMQ transport supports a rich variety of configuration options. The main tasks are configuring channels and services. To

perform a configuration a transport section handler should be added to the 'configSections' of the configuration file and a *nesterovsky-bros.web.services.WMQ* section should be created.

```
<configuration>
  <configSections>
    <section name="nesterovsky-bros.web.services.WMQ"
type="NesterovskyBros.Web.Services.Messaging.Configuration.SoapWMQConfiguration,
 SoapWMQTransport" />
  </configSections>
  <nesterovsky-bros.web.services.WMQ>
    ...
  </nesterovsky-bros.web.services.WMQ>
</configuration>
```

Channels comprise a named set of parameters that transport uses to discover how to send and receive data for a particular service. It's possible to define many channels using the following template:

```
<nesterovsky-bros.web.services.WMQ>
    <!--
      Channel definition for the Soap WMQ transport.
    All services that use Soap WMQ transport are bound to
    channels.
    At least one channel required.
    Note that name is unique through all channels name.
    It's using for service binding.
    Note:
      In case of remote connection to queue manager the
      connection name must include a host name and port, if
      any. Also you have to specify a client channel name (eg
      CLNT.CONN).
-->
<channel name="{channel name}" queue-manager="{queue manager name}"
 input-queue="{optional input queue name}" output-queue="{optional
 output queue name}"
 dead-messages="{optional dead messages queue name}"
 client-channel-name="{client channel name in case of remote queue
 manager}"
 connection-name="{connection name in case of remote queue manager}"
 />
...
</nesterovsky-bros.web.services.WMQ>
```

Service elements define how services and clients are bound to channels, whether the service is transactional, a timeout option, and more.

```
<nesterovsky-bros.web.services.WMQ>
```

```
    ...
    <!--
      Service binding.
      There are two type of service binding: server and client.
      If an element contains a type attribute then the service element
      is considered as server binding, otherwise it's considered as
      client binding.
      The server uses the correlation-id to differentiate destinations
      of messages. A guid value of id attribute is used for these
      means. It's a good idea to use the TModel guid of service in UDDI
      as such an id.
      Service is bound to URI: soap.WMQ:{service name}.
      Service name should be unique through all services.
      There could be many service bindings.
    -->
    <service name="{service name}"
      channel="{name of channel to use}"
      type="{fully qualified type name on the server to process
      messages}"
      id="{guid that identifies service type}"
      transactional="{optional boolean value that determines whether
      service is transactional}"
      timeout="{optional value of client timeout in milliseconds}"
      try-count="{optional number of attempts to process transactional
      one-way method}" />
  </nesterovsky-bros.web.services.WMQ>
```

It is important to note that:

- In order to define a service binding you have to specify a type (type attribute), which will handle requests; otherwise, binding is considered to be a client of a service.
- The ‘transactional’ and ‘try-count’ attributes have meaning only to a server so it’s only the client that uses the ‘timeout’attribute .

It’s possible to limit the maximum allowable number of simultaneously processing requests:

```
  <nesterovsky-bros.web.services.WMQ>
    <!--
      Parameters that determine dynamic characteristics of the Soap WMQ
      transport max-requests - maximum of requests to process at a
      time.
    -->
    <process-model max-requests="5" />
    ...
  </nesterovsky-bros.web.services.WMQ>
```

There is an option to perform all configurations manually, using the SoapWMQConfiguration class and its related classes:

- SoapWMQChannel.
- SoapWMQChannelCollection.
- SoapWMQServiceBinding.
- SoapWMQServiceBindingCollection.

To ensure that the configuration is loaded, invoke the SoapWMQConfiguration.Initialize method.

## PERFORMANCE COUNTERS

When we have implemented WMQ transport the following questions require consideration:

- What is the performance of the Web services layer?
- How much time does WMQ transfer take?
- In what way is performance dependent on transactions?
- How much time does WSE consume?
- How does the number of simultaneously processing requests affect speed?
- How can we speed up the implementation?

In order to address these questions we use performance counters, which means the Soap WMQ transport library has to be installed. During installation a 'SOAP WMQ Transport' performance category is created, which contains the following counters:

- Client time – counts the average time (in milliseconds) that the client spends processing a request. The counting starts right before sending data through WMQ and ends immediately after receiving a WMQ response.
- Server time – counts the average time (in milliseconds) that the server spends processing a request. The counting starts

right after receiving data through WMQ and ends immediately after sending WMQ a response.

- WMQ time – counts the average time (in milliseconds) that is consumed by WMQ operations during a request/response cycle. The value is counted both on a client and on a server.
- Listener threads – an instantaneous counter that shows the number of threads on a server that listen for requests. This number of listening threads depends on the number of different input queues in channel bindings.
- Requests processing – an instantaneous counter that shows the number of requests being processed at any one time. The value is limited by the property 'SoapWMQConfiguration.Max WorkingRequests'.
- Requests total – counts the total number of requests that the server accepted for processing.
- Requests succeed – counts the total number of requests that were processed successfully.
- Failed requests – counts the total number of requests that failed during processing.
- Requests/sec – counts the average speed of request processing (requests per second).
- Transactions – counts the total number of transactional requests that the server accepted for processing.
- Transactions pending – an instantaneous counter that shows the number of transactional requests in processing at a time.
- Transactions committed – counts the total number of transactional requests that were committed.
- Transactions aborted – counts the total number of transactional requests that were aborted.
- Transactions/sec – counts the average processing speed of transactional requests (transactional requests per second).

The main reason for installation is to register and remove performance counters. We consider, however, that performance counters are a justified feature of the transport library because they help significantly in tracking the health of the application using Soap WMQ transport.

The SoapWMQInstaller class performs automatic installation of the performance counters. There are several installation options:

- Invoke the install utility to perform a registration process.
- Create an install project and add primary output custom action for SoapWMQTransport.
- Add primary output custom action for SoapWMQTransport as a part of a bigger installation.

## SAMPLE

From the perspective of the 'final' user, who creates Web services and writes application logic, transport is only one of the services plugged into the WSE. To demonstrate how to use our WMQ transport in real life we have created two demo applications, Client and Server. The demo project was created in C# and provided as a Visual Studio .NET 2003 solution. In order to compile and test demo projects the following programs should be installed:

- Visual Studio .NET 2003.
- WMQ V5.3 or higher.
- IBM's MA7P SupportPac, which implements WMQ classes for Microsoft .NET.
- WSE 2.0.

Let's look at the functionality of both applications.

### Client project

For simplicity, it performs two operations only:

- Request limits (whatever they are, eg money): the Client sends this request and awaits a response from the server.

- Charge request: in these situations the client sends a one-way message.

### Server project

The server project also supports two operations:

- It processes limit requests and returns the actual limits or a fault message as a response.
- It processes charge requests and changes corresponding limits without sending any response.

The charge and limit operations are implemented as two different services. The rationale of this decision is that our transport implementation considers transaction options on a per service basis; therefore, all service methods can be either transactional or non-transactional.

Whether or not the service is transactional is a configuration option. By 'transaction' we mean the following set of operations is carried out on the server: get a request, process it, and send a response. If the transaction succeeds the request message is deleted from the queue, otherwise, depending on the options, the message is either moved to a dead message queue or postponed for subsequent processing.

Charge operation is transactional because it changes the database state. On the other hand, the limits operation may be non-transactional because it just gets information.

In order for the service to work, WMQ transport bindings have to be configured. These are channel and service declarations. Note also that WMQ itself should be appropriately configured.

The essential part is the SoapMQTransport project. The Client and Server projects are included to demonstrate how to use the library.

```
public decimal GetLimit(string cardId)
{
// ...
// Create service proxy. Uri is transport (protocol) specific.
LimitsProxy proxy = new LimitsProxy(new Uri("soap.WMQ:limits-
```

```
service"));
// ...
}
```

Another transport specific code is transport initialization:

```
static void Main(string[] args)
{
SoapWMQConfiguration.Initialize();
// ...
}
```

If a user decides to change the background transport, eg to TCP/IP, only these parameters need to be changed.

*Arthur and Vladimir Nesterovsky (Israel)* 

## Using WMQ with the Microsoft.Net platform

You can use WMQ in your Microsoft.Net programs. This article is written from the perspective of Visual Basic.Net, but most of it applies to the other .Net languages as well. You have three options available, each with its own set of benefits and limitations:

- WMQ classes for .Net.
- WMQ automation classes for ActiveX.
- IBM Message Queue Interface (MQI).

### .NET CLASSES FOR WMQ

IBM has recently released a set of .Net classes for WMQ as part of ServicePac CSD05 for WMQ V5.3. If you use an older version of WMQ you can't use these classes. The DLL containing the classes is *amqmdnet.dll* in the *\Bin* folder of your WMQ client. You can download this ServicePac from *http://www-3.ibm.com/software/integration/mqfamily/support/summary/wnt.html*.

The WMQ classes for .Net support major WMQ application objects with the classes listed below. There are other minor

 

classes not listed below, plus an MQAX namespace, which is a .Net wrapper for the old WMQ COM object. You can use that when you want to upgrade your programs to .Net without changing the program logic.

- **MQC** – contains all the familiar WMQ constants.
- **MQEnvironment** – controls the network settings for connecting to the queue manager.
- **MQQueueManager** – contains the standard queue manager properties and methods. You connect via the constructor.
- **MQQueue** – contains the standard (MQOD) queue properties plus methods to Get, Set, Put, and Inquire. You open queues via the constructor.
- **MQMessage** – contains the standard (MQMD) message properties. There is no MessageData property any more. To access the message buffer you must use one of the many Read*XXXX* and Write*XXXX* methods.
- **MQGetMessageOptions** – contains settings that control the MQQueue.Get() operation.
- **MQPutMessageOptions** – contains settings that control the MQQueue.Put() operation.
- **MQException** – a .Net exception class with extra properties for ConditionCode and ReasonCode.

The code sample that follows shows an example of a .Net Class.

```
Imports IBM.WMQ
Module Module1
    Sub Main()
        Try
            Dim QMgr As New MQQueueManager("MyQueueManager")
            Dim Q As New MQQueue(QMgr, "My.Queue.Name", _
                                 MQC.MQOO_INPUT_AS_Q_DEF, Nothing, _
                                 Nothing, Nothing)
            Dim Msg As New MQMessage()
            Dim Gmo As New MQGetMessageOptions()
            Gmo.MatchOptions = MQC.MQMO_NONE
            Gmo.Options = MQC.MQGMO_WAIT
            Gmo.WaitInterval = 10000
```

```
            Do
                Try
                    Q.Get(Msg, Gmo)
                Catch ex As MQException
                    If ex.ReasonCode = MQC.MQRC_NO_MSG_AVAILABLE Then
                        Exit Do
                    Else
                        Throw ex
                    End If
                Catch ex As Exception
                    Throw ex
                End Try

                ' Process the message.
                Console.WriteLine(Msg.ReadString(Msg.MessageLength))
            Loop

            Q.Close()
            QMgr.Disconnect()
        Catch ex As MQException
            Console.WriteLine(ex.ToString)
        Catch ex As Exception
            Console.WriteLine(ex.ToString)
        Finally
            Console.WriteLine("Press ENTER to continue.")
            Console.ReadLine()
        End Try
    End Sub
End Module
```

## WMQ classes for .Net – advantages

- The product WMQ classes for .Net is the only commercially supported .Net class for WMQ.
- It has the advantage of being managed code.
- Support will diminish for the MQI and ActiveX interfaces over time.

## WMQ classes for .Net – disadvantages

- It requires a specific WMQ version.
- Documentation is superficial and the examples are exclusively in C#.
- This set of classes seems to be a step backwards from the old

ActiveX classes. There is no longer a distribution list class, nor do you have the ReasonName or MessageData properties. My first impression is that this is a stripped-down, no frills, hastily thrown together set of classes.

## AUTOMATION CLASSES FOR ACTIVEX (MQAX200)

The ActiveX classes, which were available for VB6, can still be used in .Net programs in pretty much the same way as they were used in VB6.

### Add the reference to your project

- Right-click the References folder and select Add Reference.
- In the References dialogue, select the COM tab for a list of COM object references.
- Locate IBM WMQ Automation Classes for ActiveX. Select and add it to your references.

### Import the name space

Add this line to the top of your code:

```
Imports MQAX200
```

While importing namespaces is not strictly necessary, if you don't do it you will have to prefix every MQ class reference with 'MQAX200.'.

### Work with the objects

Here is a VB.Net snippet for reading a queue, which shows the differences between VB6 and .Net:

```
Dim qm As New MQQueueManager
qm.Name = "QmgrName"
qm.Connect()
Dim q As New MQQueue
q.Name = "QueueName"
q.ConnectionReference = qm
q.OpenOptions = MQ.MQOO_OPEN_AS_Q_DEF
q.Open()
```

```
Dim gmo As New MQGetMessageOptions
gmo.MatchOptions = MQ.MQMO_NONE
Dim msg As New MQMessage
Do
     Try
          q.Get(msg, gmo)
     Catch ex As Exception
          Exit Do
     End Try
     ' Process the message
     ' here.
Loop
q.Close()
qm.Disconnect()
msg = Nothing
gmo = Nothing
q = Nothing
qm = Nothing
```

## Garbage collection issues

Objects in the .Net Framework tend to stay in memory until the garbage collector routine executes, and this usually happens at the discretion of the Common Language Runtime. You can put clean-up logic for your .Net classes in a reserved, protected subroutine named Finalize().

MQAX200 objects do not run as managed code and they won't be available to you in the Finalize() routine. So any clean-up you wish to do with them, such as closing queues, disconnecting from queue managers, or destroying objects, must be done before garbage collection happens. Since the .Net garbage collection resources do not work on unmanaged code you must clean them up properly yourself, or you will have memory leakage.

## .Net differences

- WMQ constants are defined in the MQAX200.MQ namespace. To reference them, just add the namespace prefix 'MQ.' to the front of the constant name. You could also just import the MQAX200.MQ namespace, but I like having Visual Basic's Intellisense pop-up the list of constants when I type the namespace prefix.

- It's OK to declare MQ objects with the new keyword, since there are no longer any performance penalties for doing so.
- There is little use any more for the MQSession class. In VB6 it was used for manipulating the ExceptionThreshold property and as an object factory for the other MQ classes. In VB.Net there are better ways to handle exceptions and you can easily declare MQAX200 objects directly without the need to invoke MQSession's object access methods.
- The Try / Catch structure makes end-of-queue processing much easier. The MQQueue class throws an exception when you read past the end of a queue (an over-reaction if there ever was one). VB6 required clumsy error handling to deal with this, or turning off MQ error handling altogether by setting the MQSession.ExceptionThreshold property higher than 2. Of course, built-in error handling is there for a reason and if you turned it off you had to take on the responsibility for doing your own error checking. Wrapping a Try / Catch around the MQQueue.Get() method makes the whole issue a lot neater.

### MQAX200 – advantages

- Object-oriented interface.
- Easiest option for older WMQ versions.
- Works as well as it did for pre-.Net platforms.

### MQAX200 – disadvantages

- You are making calls to unmanaged code.
- Advanced OO functionality, such as inheritance, overloading, and parameterized constructers, isn't available to MQAX200 classes.
- Poor support for PCF and MQAI.

## IBM MESSAGE QUEUE INTERFACE (MQI)

The old API calls still work in .Net, but you need to handle the API declarations and the structure definitions with care. If you just

paste them into your program from CMQB.Bas they won't work. The problem is that VB.Net changed the default mode for passing parameters from ByRef to ByVal, therefore, the old calls fail because they don't pass parameters properly. Fix this by specifying ByRef/ByVal explicitly for each API parameter. The .Net upgrade wizard can automate this chore for you.

As an example, let's look at the MQConn call. The IBM-supplied syntax for this call was written for Visual Basic 4.0 and looks like this:

```
Declare Sub MQCONN Lib "MQIC32.DLL" Alias "MQCONNstd@16" (ByVal
QMgrName As String * 48, Hconn As Long, CompCode As Long, Reason
As Long)
```

Remove the maximum string length from the QMgrName parameter definition to make it syntactically correct for VB.Net. However, when you try to execute the call it still won't work.

```
Declare Sub MQCONN Lib "MQIC32.DLL" Alias "MQCONNstd@16" (ByVal
QMgrName As String, Hconn As Long, CompCode As Long, Reason As
Long)
```

To fix this declaration, look at the parameter definitions. Make sure that each parameter is defined with either the keyword ByRef or ByVal. This is because parameters were passed by reference as the VB6 default, but now they are passed by value as the .Net default. This small change makes all the difference between your code working or not working.

```
Declare Sub MQCONN Lib "MQIC32.DLL" Alias "MQCONNstd@16" (ByVal
QMgrName As String, ByRef Hconn As Long, ByRef CompCode As Long,
ByRef Reason As Long)
```

## Structures are different from types

The user-defined data type, or UDT, has been with Visual Basic for as long as most programmers can remember. It is used to create a data structure consisting of fields that are stored contiguously in memory. In other words, suppose you have a UDT consisting of two fields – FirstName and LastName. You declare a variable named 'Person' of this type, and assign the values 'Agent' and 'Smith' to the two fields, respectively. The memory area occupied by the Person variable will look like this: 'AgentSmith'. This

contiguous storage makes it possible to pass UDT variables to C-style APIs as parameters.

The UDT is no longer supported in .Net, having been replaced by the Structure. Structures do not store their fields contiguously by default so you cannot pass them to C-style APIs without a lot of serious persuasion. This 'persuasion' is done via .Net marshalling attributes. To use marshalling attributes you should first import the InteropServices namespace, using this import statement:

```
Imports System.Runtime.InteropServices
```

Next, add the StructLayout attribute as a prefix to each of your structure definitions. This attribute forces the common Language Runtime to store all fields in the structure contiguously.

```
<StructLayout(LayoutKind.Sequential, CharSet:=CharSet.Ansi)> _
Structure MQOD
    Public StrucId As String
    ...
End Structure
```

Because many of the WMQ UDT fields are fixed length strings you have another problem – VB.Net does not support fixed-length strings. This is solved with another marshalling attribute, MarshalAs. You control the string length with the SizeConst parameter. Convert the data type from String to a Char array. Each such Char array must be padded to the exact length prior to using the structure in an API call.

```
<StructLayout(LayoutKind.Sequential, CharSet:=CharSet.Ansi)> _
Structure MQOD
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=4)> _
    Public StrucId() As Char
    ...
End Structure
```

A word of warning: the Visual Studio Upgrade Wizard is not very helpful for the WMQ Structures. It has a bug that causes it to use an incorrect attribute for the fixed-length strings. Also, it does not apply the StructLayout attribute.

Once you have added all your marshalling attributes you must add initialization logic. It's really up to you as the programmer where you initialize the fields, but in my example I added an initialization

method to my structures. Yes, VB.Net structures support methods, just like objects do. Here is an example of my finished MQOD structure – UDT converted to a .Net version structure.

```
<StructLayout(LayoutKind.Sequential, CharSet:=CharSet.Ansi)> _
Structure MQOD
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=4)> _
    Public StrucId() As Char
    Dim Version As Integer
    Dim ObjectType As Integer
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=48)> _
    Public ObjectName() As Char
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=48)> _
    Public ObjectQMgrName() As Char
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=48)> _
    Public DynamicQName() As Char
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=12)> _
    Public AlternateUserId() As Char
    Dim RecsPresent As Integer
    Dim KnownDestCount As Integer
    Dim UnknownDestCount As Integer
    Dim InvalidDestCount As Integer
    Dim ObjectRecOffset As Integer
    Dim ResponseRecOffset As Integer
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=32)> _
    Public ObjectRecPtr() As Char
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=32)> _
    Public ResponseRecPtr() As Char
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=40)> _
    Public AlternateSecurityId() As Char
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=48)> _
    Public ResolvedQName() As Char
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=48)> _
    Public ResolvedQMgrName() As Char
    Sub Initialize()
        Me.StrucId = "OD  "
        Me.Version = MQ.MQOD_CURRENT_VERSION
        Me.ObjectType = MQ.MQOT_Q
        Me.ObjectQMgrName = New String(" ", 48)
        Me.DynamicQName = "AMQ.*" & New String(" ", 43)
        Me.AlternateUserId = New String(" ", 12)
        Me.AlternateSecurityId = New String(" ", 40)
        Me.ObjectRecPtr = New String(Chr(0), 32)
        Me.ResponseRecPtr = New String(Chr(0), 32)
        Me.ResolvedQMgrName = New String(" ", 48)
        Me.ResolvedQName = New String(" ", 48)
    End Sub
End Structure
```

Putting it all together, here is a sample of the Main() subroutine for a program using the MQI API, with as much 'fluff' compressed out as possible.

```
' I put my MQ constants here.
Private Enum MQ
    MQCC_OK = Ø
    MQCC_FAILED = 2
    MQOD_CURRENT_VERSION = 3
    MQOO_INPUT_AS_Q_DEF = 1
    MQOT_Q = 1
    MQRC_NO_MSG_AVAILABLE = 2Ø33
    MQMO_NONE = Ø
End Enum

Declare Sub MQCONN Lib "MQIC32.DLL" Alias "MQCONNstd@16" _
    (ByVal QMgrName As String, ByRef hConn As Integer, _
     ByRef CompCode As Integer, ByRef Reason As Integer)
Declare Sub MQOPEN Lib "MQIC32.DLL" Alias "MQOPENstd@24" _
    (ByVal hConn As Integer, ByRef ObjDesc As MQOD, _
     ByVal Options As Integer, ByRef Hobj As Integer, _
     ByRef CompCode As Integer, ByRef Reason As Integer)
Private Declare Sub MQGETX Lib "MQIC32.DLL" Alias "MQGETstd@36" _
    (ByVal hConn As Integer, ByVal Hobj As Integer, _
     ByRef MsgDesc As MQMD, ByRef GetMsgOpts As MQGMO, _
     ByVal BufferLength As Integer, ByVal Buffer As String, _
     ByRef DataLength As Integer, ByRef CompCode As Integer, _
     ByRef Reason As Integer)
Declare Sub MQDISC Lib "MQIC32.DLL" Alias "MQDISCstd@12" _
    (ByRef hConn As Integer, ByRef CompCode As Integer, _
     ByRef Reason As Integer)
Declare Sub MQCLOSE Lib "MQIC32.DLL" Alias "MQCLOSEstd@2Ø" _
    (ByVal hConn As Integer, ByRef Hobj As Integer, _
     ByVal Options As Integer, ByRef CompCode As Integer, _
     ByRef Reason As Integer)
Public Sub Main()
    Dim hConn As Integer ' Connection handle.
    Dim hQueue As Integer ' Queue handle.
    Dim CompCode, Reason, intMsgLen As Integer
    Dim OD As MQOD
    Dim MD As MQMD
    Dim GMO As MQGMO
    Dim strMsg As String
    ' Connect to queue manager.
    MQCONN("MyQMgrName", hConn, CompCode, Reason)
    If CompCode = MQ.MQCC_FAILED Then
        Console.WriteLine("MQCONN failed. RC(" & Reason & ")")
        Exit Sub
    End If
```

```
    ' Open the queue.
    OD.Initialize()
    OD.ObjectName = Left("My.Queue.Name" & New String(" ", 48),48)
    MQOPEN(hConn, OD, MQ.MQOO_INPUT_AS_Q_DEF, hQueue, _
           CompCode, Reason)
    If CompCode = MQ.MQCC_FAILED Then
        Console.WriteLine("MQOPEN failed. RC(" & Reason & ")")
        Exit Sub
    End If
    ' Initialize the message and GMO.
    MD.Initialize()
    GMO.Initialize()
    GMO.MatchOptions = MQ.MQMO_NONE
    strMsg = New String(Chr(Ø), 1ØØØ)
    ' Process the queue.
    Do Until Reason = MQ.MQRC_NO_MSG_AVAILABLE
        MQGETX(hConn, hQueue, MD, GMO, strMsg.Length, _
               strMsg, intMsgLen, CompCode, Reason)
        If CompCode < MQ.MQCC_FAILED Then
            ' Do something with the message.
        Else
            Exit Do
        End If
    Loop
    MQDISC(hConn, CompCode, Reason)
    MQCLOSE(hConn, hQueue, Ø, CompCode, Reason)
End Sub
```

## Return to 'DLL hell'

If you check the opening comments in CMQB.BAS you will see that you must use different compile options depending on whether you run the code on an WMQ client or a queue manager machine. This is because the API functions are defined in differently-named DLLs depending on whether it's a client or server machine. This can be especially annoying if you are developing programs on a client machine that are intended to run on a server machine. This problem has at least three solutions:

1 Set compiler directives to compile the calls to either the client DLL or the server DLL as needed. This is the recommended way in CMQB.BAS.

2 Fool the system into thinking the expected DLL is on the target machine. On a client machine, locate file MQIC32.DLL. Make

a copy of the file and rename it MQM.DLL. Now most API calls coded against MQM.DLL will work on this machine. Reverse the process for a queue manager machine by copying MQM.DLL and renaming it MQIC32.DLL. Okay, I'll admit this is an ugly hack, but it works for most functions.

3 Use polymorphism. Declare an abstract class with all the required MQ functions prototyped. Create two derived classes – one implemented for a client and the other implemented for a server. In your program, create an object of the class type you need, and then assign it to an instance of the base class. All subsequent calls to the base class methods will now actually call the correct derived class implementation.

### MQI – advantages

- Supports all WMQ functionality, including PCF and MQAI.
- May run marginally faster than ActiveX.
- If you are upgrading VB6 programs that used this API you may find it easier to keep it than convert to another API.
- If you spend any amount of time in this API you will feel like a real programmer and will be fully justified in looking down your nose at lesser programmers with their silly object-oriented interfaces. Of course, they will get twice as much done as you.

### MQI – disadvantages

- You are calling unmanaged code.
- It has a less intuitive interface.
- VB.Net structures pass data differently from VB6 user-defined data types, and extra code is needed to resolve it.
- This API is very delicate. Any failure to properly initialize your structures will cause program exceptions.
- You must declare the API functions with different DLLs depending on whether the application runs on a client or queue manager machine.

## CONCLUSION

At the company where I work we have a large investment in VB6 code but we are developing new systems in VB.Net. Because the WMQ classes for .Net have been available for such a short time we are still using the ActiveX classes for most things and MQI where we absolutely have to. We have WMQ V5.2.1 and we probably won't upgrade just for the .Net support given the disappointing quality of those classes.

*Mills Perry*
*ZyQuest (USA)*


# Statistics and accounting in WBI MB

In May 2003 IBM announced the release of WebSphere Business Integration Message Broker Version 5.0 (WBI MB), which is the successor to WMQ Integrator Broker V2.1. With the new version the product introduced the new statistics and accounting functionality. This article discusses the capabilities and application of that functionality.

## STATISTICS AND ACCOUNTING INFORMATION

The statistics and accounting function gathers dynamic information about message flow use, and processing details about the nodes involved in message processing. Accounting information provides a customer with data on broker usage, which enables chargeback for invoking message flows. These statistics can be used for monitoring and performance analysis.

A WBI MB broker can provide the following different statistics:

- **Message flow statistics**. This is the highest level of information and includes data on the names of the flow, the execution group a message flow is running in, the broker, and the timeframe of the data-gathering period. Additionally, it contains the following information:

- the number of messages processed in the measurement interval.
- the number of commits and rollbacks used to process the messages.
- the total, minimum, and maximum length of the messages processed.
- the total, minimum, and maximum CPU time used to process the messages.
- the total, minimum, and maximum elapsed time used to process the messages.
- thread statistic information about the number of threads and the maximum number of parallel processing threads for this message flow.

In most cases the invocation of a message flow is used as an accounting unit. The functionality of the message flow is associated with a price, so different message flows could have different prices.

Some customers measure broker usage by the resources used by message flows. For this kind of accounting the amount of CPU time or the size of the messages processed form useful accounting criteria.

- **Message flow thread statistics**. Information similar to that collected for a message flow is also gathered for each thread that is processing for the message flow. Each thread is identified by a thread number. This thread number does not correspond to any thread ID within the processing of the flow, eg the one in the user trace or the one that can be retrieved by a plug-in node.
- **Node statistics**. This statistic contains information about all nodes within a message flow. For each node the number of input and output terminals and the node type (eg MQInputNode or ComputeNode) are included. The following processing information is also included:

- the number of times the node was invoked.
- total, minimum, and maximum elapsed time spent in this node.
- total, minimum, and maximum CPU time used for this node.

In the previous version of the broker you could use the user trace only to get information about the elapsed time for a node. When comparing the results of a user trace with the results of the new node significant differences appear for some nodes. The accuracy of the statistics and accounting information seems to be much higher, so the node statistic can be used for performance analysis, for example, to get information about which nodes of a message flow are using most of the elapsed and processing time. This provides a good indication of where to start investigating possible performance improvements.

For some nodes the statistic shows significant differences between the elapsed time and the CPU time. In most cases this is because the nodes are performing an I/O operation, such as a database update.

- **Terminal statistics**. For each terminal of a message processing node the terminal statistic contains information on the name of the terminal, whether it is an input or output terminal, and the number of invocations for this terminal.

  If a message flow contains a filter node that checks for the correctness of the message and the false terminal indicates incorrect messages, this statistic could be used to get information about the number of incorrect messages that are processed by the flow. This can be achieved by counting the false terminals of a filter node that checks for message correctness.

  Another use for this statistic could be to count processed elements. It could be achieved if messages are processed which contain an arbitrary number of records. If one node

splits up the message and forwards each record separately on its out terminal the number of invocations of the out terminal can be used to count the number of records that have been processed.

You need to enable the statistics and accounting information function to gather this information; it is disabled by default. Collection of the data is integrated in the broker. The broker must not be changed to get the information; redeploying message flows is not required. A simple command can be used to turn the statistics on and off.

## TYPES OF INFORMATION

WBI MB provides two different types of statistics information – snapshot and archive – which are used for different time periods. Each time period or type of statistic can separately be enabled and disabled, and it is possible to enable both at the same time. When snapshot statistics are enabled the broker collects all information for a predefined brief time interval and makes this information available. The time interval is approximately 20 seconds. It cannot be altered by the user.

The snapshot statistics can be used to monitor the broker's processing activities. A monitoring program could retrieve the snapshot statistic data and display it as an indication of health , eg if a flow is processing messages this could be used as a sign that everything is OK. Snapshot statistics can also be useful in production environments to gather performance data.

Archive statistics data is intended for accounting and chargeback purposes and is collected over a longer period. The default period is one hour, but it can be adjusted by a user within the range of 10 minutes to 10 days.

## GETTING THE STATISTICS AND ACCOUNTING INFORMATION

WBI MB can write the statistics and accounting information to different destinations. On all platforms the information can be

written to the WBI MB user trace and the broker can publish it on a subscription basis.

When writing the information to the WBI MB user trace it must be handled like any other user trace information. It can be retrieved using the standard commands **mqsireadlog** and **mqsiformatlog**. When using the user trace destination you should ensure that tracing is not active in parallel. Otherwise the statistics and accounting information may be overwritten by the trace information. That's why this destination should be used mainly for short-term statistical information, eg when using the data for performance analysis. Usually a batch of messages is processed and the data is retrieved for analysis purposes. Afterwards, the data is usually discarded.

In the user trace the broker uses message numbers BIP2380I to BIP2383I. An example message for a flow statistic is shown below. The statistical information is for message flow DNF_ILC_FIN_3140 in broker WFMGBRK.

```
2003-08-14 13:54:54.303344        3   UserTrace   BIP2380I: WMQI
message flow statistics.
ProcessID='66009',
Key='0',
Type='Archive',
Reason='Shutdown',
BrokerLabel='WFMGBRK',
BrokerUUID='a7b766b7-f600-0000-0080-fe0735107e6b',
ExecutionGroupName='MGTEST',
ExecutionGroupUUID='ad138ee2-f600-0000-0080-fe0735107e6b',
MessageFlowName='DNF_ILC_FIN_3140',
StartDate='2003-08-14',
StartTime='13:43:13.418564',
EndDate='2003-08-14',
EndTime='13:43:50.267868',
TotalElapsedTime='211785',
MaximumElapsedTime='44600',
MinimumElapsedTime='39836',
TotalCPUTime='121323',
MaximumCPUTime='26110',
MinimumCPUTime='23579',
CPUTimeWaitingForInputMessage='3344',
ElapsedTimeWaitingForInputMessage='36632372',
TotalInputMessages='5',
TotalSizeOfInputMessages='18478',
```

```
MaximumSizeOfInputMessages='4426',
MinimumSizeOfInputMessages='3418',
NumberOfThreadsInPool='1',
TimesMaximumNumberOfThreadsReached='5',
TotalNumberOfMQErrors='0',
TotalNumberOfMessagesWithErrors='0',
TotalNumberOfErrorsProcessingMessages='0',
TotalNumberOfTimeOuts='0',
TotalNumberOfCommits='5',
TotalNumberOfBackouts='0'.
```

For readability the message in the example has been split into multiple lines. The broker can also publish the statistics and accounting data. This is done using a pre-defined topic structure and the default subscription point. The topic structure looks like this:

```
$SYS/Broker/<broker name>/StatisticsAccounting/<type>/<EG name>/
<Flow name>
```

The placeholders in this structure are substituted by the corresponding value for the broker name, the type of accounting and statistic information – either SnapShot or Archive, the name of the execution group the message flow is running in, and the name of the message flow. Using this structure an accounting application can subscribe to the relevant information.

As can be seen from the fact that the topic name starts with $SYS, statistics and accounting data publications are broker publications. The broker is always allowed to publish these topics. If topic-

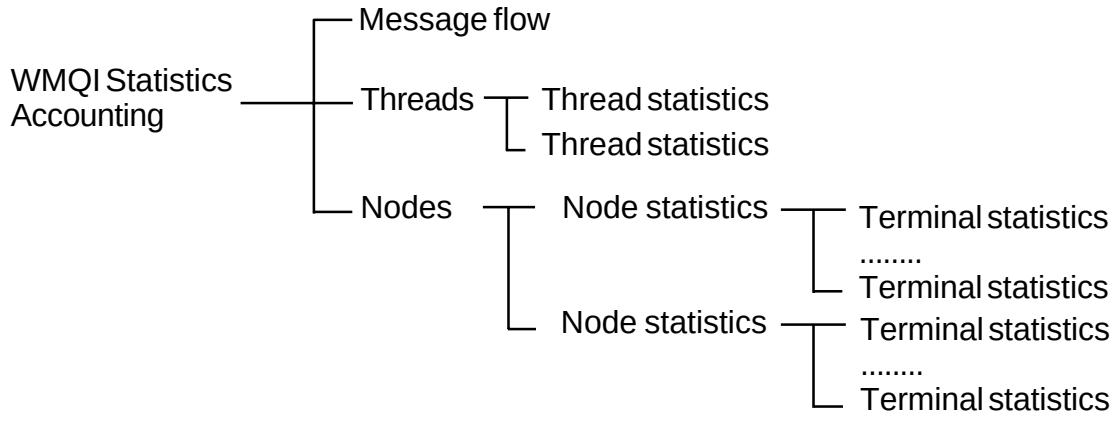


*Figure 1: XML structure of statistics and accounting data*

based security for the broker domain is used the user name that is running the application requesting the statistics and accounting information must be allowed to issue the subscription.

When publishing the information the broker issues a publication with an RFH2 header and the accounting and statistics information in the message body. The message body is in XML format. The information is structured in a folder as shown in Figure 1. The details of each statistic are available as attributes to the appropriate element in the tree.

Publish and subscribe allows an application to get the accounting and statistics information for a whole WBI MB broker domain.

On z/OS the WBI MB broker offers a third destination for the statistics and accounting data, the Systems Management Facility (SMF). SMF is a standard z/OS high-performance service that can be used to store and handle data, eg switching to a different dataset if one becomes full.

Statistics and accounting information is provided in SMF records of type 117, which has two subtypes. Subtype 1 contains message flow statistics and, if enabled, message flow thread statistics. Subtype 2 is used when node statistics are requested. This subtype also contains message flow statistics and, if enabled, terminal statistics.

The format of SMF type 117 is a dynamic, record-oriented C-structure, consisting of character and binary data. Further information on the data format in SMF records can be found in the header file BipSMF.h and in the WBI MB product documentation.

To write to SMF, the SMF destination in the broker must be enabled. The broker must be allowed to write the SMF data by enabling the broker's user ID access to the BPX.SMF facility class.

## CONTROLLING STATISTICS AND ACCOUNTING INFORMATION

With the new broker there are two additional commands available to define and control the settings for statistics and accounting information: **mqsichangeflowstats** and **mqsireportflowstats**.

**mqsichangeflowstats** is used to change the settings for the type of statistics and accounting information requested from a broker. The setting can be changed for a specific execution group or for all execution groups. The settings can address a specific message flow or all message flows in the specified execution group. Furthermore you can also specify the type of statistic requested and its destination. The broker allows different destinations for snapshot and archive statistics.

If the statistics and accounting function is turned on, message flow statistics are always gathered. Thread statistics and node statistics can be controlled separately. Terminal statistics can be included only if node statistics are also requested.

The new command **mqsireportflowstats** displays the current settings for statistics and accounting processing. The settings can be requested for one type of statistics and accounting information at a time. The requested information can be at the same granularity as for **mqsichangeflowstats**, either for a specific execution group or for all execution groups of the broker, and for a specific message flow or all message flows in the specified execution group.

Once enabled, the broker writes the statistics and accounting information to the specified destination at the set intervals. Other events can force the statistics and accounting information to be written, however, ie if message flows are redeployed or if the broker is shutting down.

## FURTHER DETAILS

The statistics and accounting functionality is available on all WBI MB-supported platforms and for all members of the broker family, including the WBI Event Broker. The information is collected for all built-in nodes except MQeInput and MQeOutput, SCADAInput and SCADAOutput, Real-timeInput and Real-timeOptimizedFlow. It is also collected for any customer-provided plug-in node.

With CSD2, WBI MB introduced another capability to the statistics and accounting functionality. It allows the statistic information to

be partitioned according to a customer-defined criterion known as the Accounting Origin. To allow for this partitioning a message flow has to be modified to provide the Accounting Origin. This is done by providing a value in a field in the message environment, eg by using a compute node, as shown here:

```
SET Environment.Broker.Accounting.Origin = 'MyDepartment';
```

The information in the environment field is recognized when the flow returns to its input node. There is no predefined value that has to be entered into this field, but usually partitioning is decided by the department or organizational unit to which the message belongs.

On all platforms archive statistics and accounting information can be gathered for intervals as defined by the WBI MB user. On z/OS there is an option to synchronize writing the statistics and accounting information with the same type of information of other z/OS subsystems, eg a WMQ queue manager. If the archive interval is set to 0 the broker listens to the Event Notification Facility (ENF). If the system issues an ENF event with event code 37 the broker writes to the archive statistics.

When enabling the data gathering for statistics and accounting purposes you should be aware that there may be an impact on performance. The more statistics are requested, the higher the impact. How much a flow is impacted depends on what the flow is doing. General rules to estimate the impact on the throughput and processing resources are not available.

## SUMMARY

WebSphere Business Integration Message Broker provides new functionality for gathering statistics and accounting information. It provides different kinds of information to suit different purposes, eg accounting message flow processing to allow chargeback of the costs for monitoring and performance analysis.

*Michael Groetzner*
*IBM (Germany)*

Symbol Technologies and IBM have recently announced the joint development of a new generation of hand-held wireless and scanning solutions, which are customized for specific industries.

The companies claim the technology will provide a new generation of mobile workers with rapid access to key business information, such as helping retailers track inventory and customer orders from the factory to the cash register more efficiently.

Symbol's ruggedized wireless mobile computers, running IBM's WebSphere Micro Edition, will use the messaging capabilities of MQ Everyplace to allow users access to Siebel, SAP, JD Edwards, or PeopleSoft applications, in addition to information from disparate databases, such as IBM's DB2e database software.

Symbol will integrate devices with embedded IBM mobile middleware so workers will be able to use their handhelds to rapidly access data from back-end computer systems running on IBM eServer x-Series servers for the enterprise.

*For more information contact your local IBM representative.*

* * *

Candle has recently announced six new PathWAI packages that are claimed to accelerate IBM WebSphere Java 2 Enterprise Edition (J2EE) and enterprise application integration (EAI) initiatives.

The new solutions are designed to complement the original PathWAI packages, which include architecture and development services, as well as application performance testing, tuning, and management packages for WebSphere Application Server and WebSphere MQ.

Amongst the new packages released is the PathWAI Tuning Workbench for WebSphere Business Integration, which is said to deliver powerful tuning and testing tools, including PathWAI Message Editor and monitors for WebSphere MQ and WebSphere Business Integration.

Candle claims these tools will enable organizations to accelerate problem resolution and gain assurance that their infrastructures will support business-critical applications once in production.

*For more information contact:*
Candle, 100 N Sepulveda Blvd, El Segundo, CA, 90245, USA.
Tel: +1 310 535 3600.
Fax: +1 310 727 4287.
Web:http://www.candle.com

Candle, 1 Archipelago, Lyon Way, Frimley, Camberley, Surrey, GU16 7ER, UK.
Tel: +44 1276 414 700.
Fax: +44 1276 414 777.

* * *