60

MQ update

June 2004

In this issue

Published by

Xephon Inc
PO Box 550547
Dallas, Texas 75355
USA

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

Editor

Trevor Eddolls
E-mail: trevore@xephon.com

Publisher

Nicole Thomas
E-mail: nicole@xephon.com

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 2000 issue, are available separately to subscribers for $33.75 (£22.50) each including postage.

Contributions

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

MQ Update on-line

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

Printed in England.

ESQL debugging in IBM WebSphere Business Integration Message Broker V5.0 toolkit

This article is aimed at people with some previous working experience and knowledge of using IBM MQSeries Integrator V2.1 and WebSphere MQ products. Additional knowledge of the Agent Controller and Visual Debugger is useful. WebSphere Business Integration Message Broker V5.0 will be referred to as Message Broker V5.0 in future in the article.

ESQL debugging is a new, previously unavailable, feature present in Message Broker V5.0. ESQL debugging is done in the Debug perspective. It is very useful especially for flows containing complex and large amounts of ESQL code. Various uses and features of the ESQL debugger will be discussed in the following sections.

The V5.0 BVTMain message flow is used in this article for illustrative purposes, with the assumption that it exists in the workplace. The flow contains a compute node with the following ESQL code:

```
*****************************************************************************************
CREATE COMPUTE MODULE "v5.ØBVTMain_Compute"
      CREATE FUNCTION Main() RETURNS BOOLEAN
      BEGIN
            DECLARE origSharePrice INTEGER 1;
            DECLARE I INTEGER 1;
            DECLARE J INTEGER CARDINALITY(InputRoot.*[]);
            WHILE I < J DO
                  SET OutputRoot.*[I] = InputRoot.*[I];
                  SET I = I+1;
            END WHILE;
            SET origSharePrice = InputRoot.XML.SP1.Price;
            SET OutputRoot.XML.SP1.Company='Non-Existent Ltd';
            SET OutputRoot.XML.SP1.Company.(XML.Attribute)ID='NEØØ1';
            SET OutputRoot.XML.SP1.Address1='1 Long Street';
            SET OutputRoot.XML.SP1.Address2='Bigtown';
            SET OutputRoot.XML.SP1.Address3='Greenshire';
            SET OutputRoot.XML.SP1.Address4.Country='UK';
      END;
END MODULE;
```

(Note: the code given above is for illustrative purposes only.)

REQUIREMENTS

The following set-up is required before proceeding:

1 A configuration manager created with its queue manager and listener running.

2 A broker created with its queue manager and listener running. (Less set up is required if the configuration manager and broker share the same queue manager and listener.)

3 A domain connection connected to the configuration manager and a broker to deploy to in the toolkit.

4 A deployed message flow to debug.

5 An agent controller installed and running on the machine where the broker is installed.

GETTING STARTED

To get started:

1 Once the message flow has been deployed, open the flow debug perspective – **Window/Open Perspective/Other**. Select **Flow Debug** from the list and click **OK**. By default a debug perspective will open along with the flow debug perspective (ESQL debugging is done in the debug perspective). If it does not, the debug perspective will open when the user steps into ESQL code in the message flow. The default behaviour can be changed from the preferences menu – select **Window/Preferences**. Select **Flow Debug** from the left pane. Here the user can choose whether to **Enable ESQL source debugging** or not by default.

2 Attach to the flow run-time engine of the deployed message flow. Click on the **Attach to Flow runtime** toolbar button.

Figure 1: Add a breakpoint and step into the compute node

Select **localhost** and click **Next**. Select the flow engine where the flow resides from the list, and click **Finish**.

3 Double-click the message flow in the Flow Debug pane to open the message flow in the flow editor pane if it is not already open.

4 Add a breakpoint before the compute node (or other nodes with ESQL) that contains the ESQL code in the message flow as shown in Figure 1.

5 Start the message flow if it is not already started and put a message on the queue of the input node (See *USEFUL TITBITS* section at the end of the article). When the message

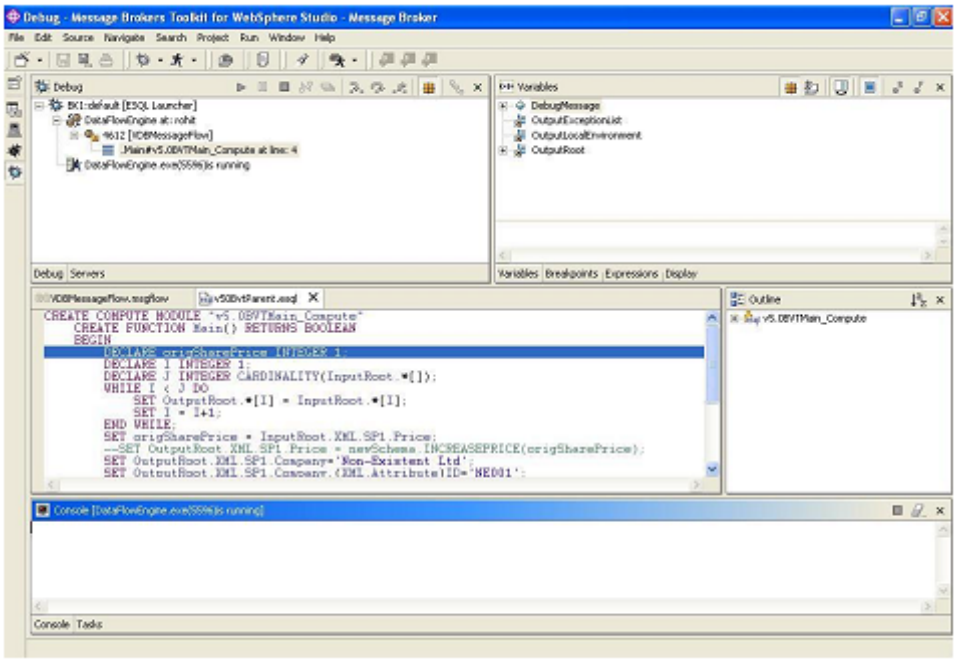

Figure 2: The debug perspective

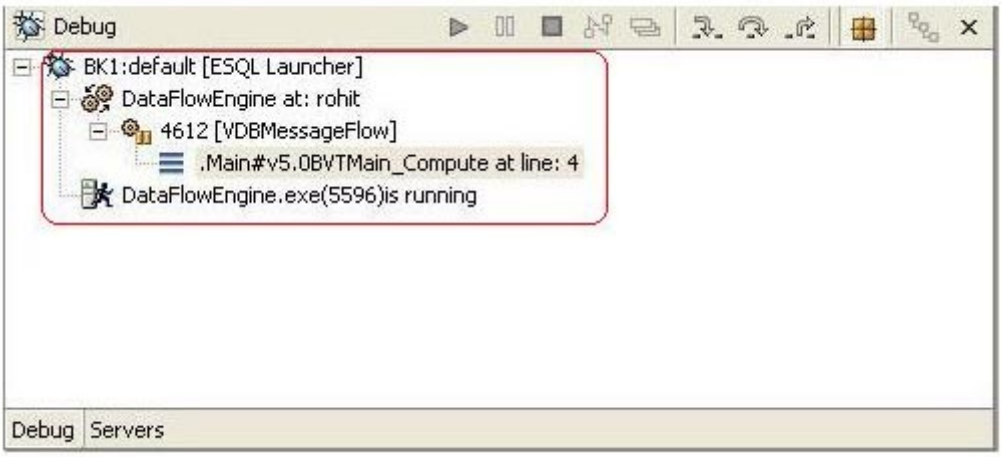

Figure 3: Debug pane

gets to the breakpoint before the compute node, the user can step into the compute node, as shown in Figure 1.

6 In the Flow Debug pane click the *Step Into Source Code* button. This will open the debug perspective if it is not already open and display the ESQL source code.

THE DEBUG PERSPECTIVE

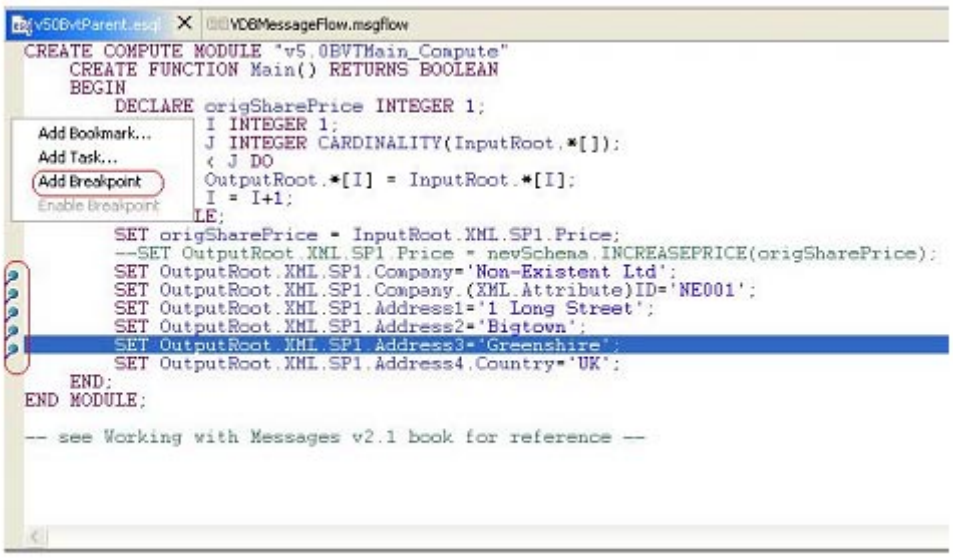

Figure 4: Columns and breakpoints

In the debug perspective (see Figure 2):

1 The Debug pane gives details about the Data Flow Engine process id, the broker, message flow, and the current position of the ESQL debugger in the ESQL code (see Figure 3).

2 The user can add breakpoints to the ESQL code by right-clicking on the column next to the ESQL statement where the breakpoint is desired and select **Add Breakpoint**. Alternatively a double-click can also be used to add a breakpoint. The column and breakpoints are shown in Figure 4.

3 Breakpoints can also be added to the ESQL source code when it is written and before attaching to the flow run-time engine. It is quite useful to be able to add the ESQL breakpoints dynamically during debugging, and it provides great flexibility and ease of use.

4 Once breakpoints have been added, the user can step through the ESQL source code with the debugger, stopping at each breakpoint. The user has similar actions to choose

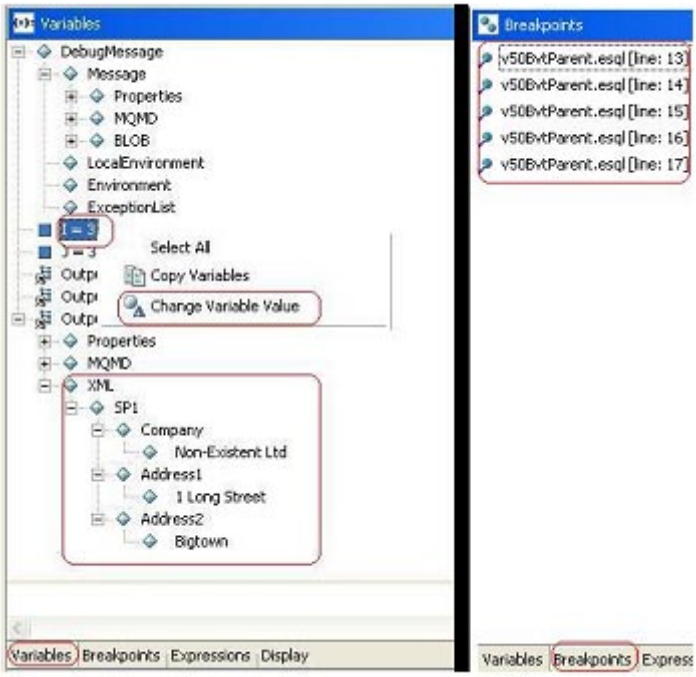

Figure 5: Variables and breakpoints

from in the right-hand corner of the Debug pane, which include *Play*, *Suspend*, *Terminate*, *Step Into*, *Step Over*, and *Step Return*.

5 The Variables pane shows and displays the construction of the message as it moves through the ESQL code breakpoints. The user is able to see the construction of variables used in the code, and can see the manipulation of the message. Perhaps the most important feature is that the user has the ability to modify the value of a variable in debug mode. As shown in Figure 5, the user can right-click on a variable in the **Variables** tab and select **Change Variable Value**. This is quite useful because different values can be used during debug mode to check the output. This saves the user having to run another debug cycle to test different values. The **Breakpoints** tab displays a list of the breakpoints and their position in the code, as shown in Figure 5.

6 Once the message has passed through the breakpoints in the ESQL code, the message will exit the compute node displaying the message flow in the flow debug perspective and move to the next breakpoint in the message flow, if there are any.

7 The Outline pane gives an overview of where the user is in the ESQL code.

8 The Console pane is standard with the debug perspective and allows the user to interact with the running program. (Note: this is not within the scope of this article.)

ESQL debugging is a very powerful and dynamic feature in Message Broker V5.0. The ability to be able dynamically to debug large ESQL code on the spot by adding breakpoints and altering variable values in the code is probably one of the most significant uses of the ESQL debugger. It provides a major capability for visual debugging as a whole.

USEFUL TITBITS

It can be useful to know that:

1 The underlying IBM WebSphere MQ program provides a useful command called **amqsput.exe**, which can be run from the command line. The user can use this to put a message on a queue.

2 ESQL written in the ESQL editor follows a colour scheme (not visible in this article). There is a separate colour for comments, keywords, user code, etc. This makes it easier to read the ESQL code.

3 The ESQL editor also has another feature, called Content Assist. While typing in the ESQL editor, the user can stop halfway through a word and press *Ctrl* and *Space*. This displays a list of all possible keywords to select from, based on the incompletely typed word.

Rohit Bhasin
Software Engineer, WebSphere MQI GUI Test Team
IBM Hursley (UK)

Message warehousing with WebSphere MQ Integrator Broker and DB2 UDB

INTRODUCTION

WebSphere MQ is IBM's messaging product. This product features an asynchronous communication technique, via messages, that decouples the application sending the messages from the application that receives and processes the information. Between those applications there can be a message broker like WebSphere MQ Integrator Broker that routes or reformats the messages that are interchanged between the sending and the receiving application. The broker may even translate the protocols between these applications. In such a system, as shown in Figure 1, the broker is usually needed for both directions – from the sending application to the receiving application and also when the receiving application replies to the request message.

If the sending application and the receiving application belong to different organizations, it is necessary to be able to determine which messages have passed the broker and the status of the messages. To determine which messages passed the broker, auditing capabilities are required. Depending on the complexity of the processing in the broker and the kind of messages that are interchanged between both applications, it must be possible to determine the actual state of a message. One example where this is required is the WebSphere Business Integration for Financial Networks product (abbreviated to WebSphere BI for FN) – for further details see http://www.ibm.com/software/integration/wbifn. This product consists of a base part that contains common functionality required on top of WebSphere MQ Integrator Broker to deliver extensions to the market. One such extension is the Extension for SWIFTNet (ESN). This extension allows applications to connect to the Secure Internet Protocol Network (SIPN) provided by SWIFT (Society for Worldwide Interbank Financial Telecommunication). Messages, for example SWIFT FIN messages, can be exchanged across this network. The exchange of FIN messages is based on a complex session protocol.

A simplified overview of the SWIFT FIN processing over the SIPN is shown in Figure 2. In this Figure, it can be seen that SWIFT FIN messages arrive at a component called Interface Layer Client. This component stores the message data in a database. The reason for not forwarding the message directly is that the access to the SIPN is twofold:

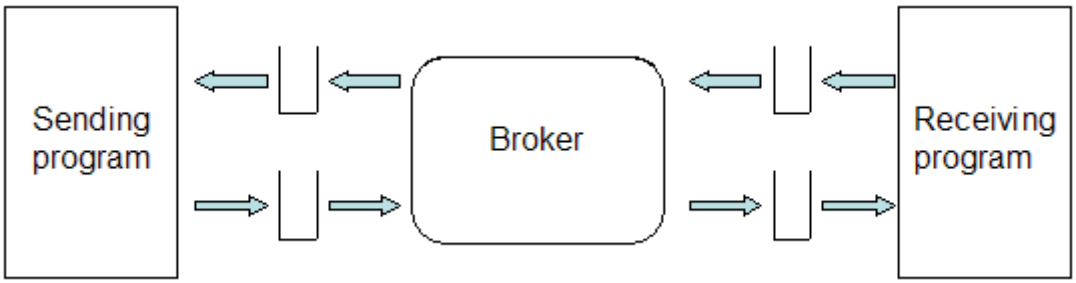

Figure 1: Message broker

- Sending FIN messages is allowed only if a logical session, represented by a SWIFT logical terminal (LT), is established.
- If the logical session is established, only a defined number of messages can be sent to the SIPN while the acknowledgements (ACKs) are still outstanding.

In addition to the translation of the message formats, Figure 2 also shows that a protocol conversion is required. Access to the SIPN is based on a request/reply model while the FIN processing is based on asynchronous correlated request messages. For further details on the SWIFT FIN processing on the SIPN see either information from SWIFT or the documentation about WebSphere BI for FN ESN.

The messages interchanged between the financial application and the SWIFT network represents, for example, high-value payments. For a bank it is essential to know whether the payment is already accepted and stored in the database, whether it is already sent to the SIPN, or whether the message is already acknowledged by the network.

When searching for a specific message, a user usually doesn't know technical criteria like the WebSphere MQ message id that identifies the message. Instead, the user wants to search using a criterion that is part of the message, for example, a specific

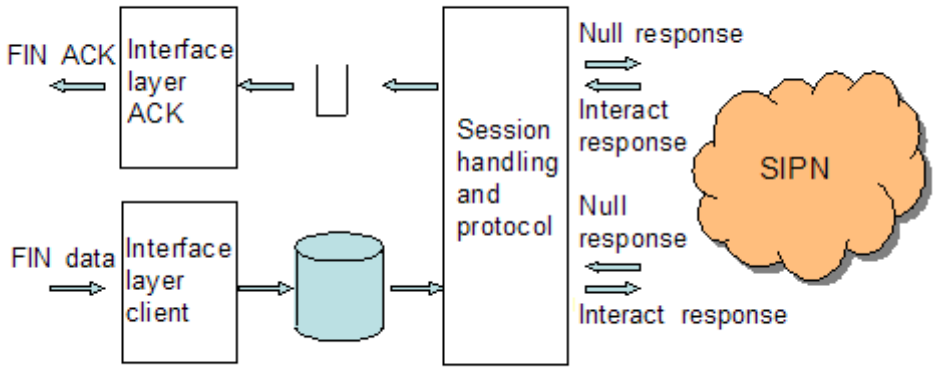

Figure 2: WebSphere BI for FN ESN FIN processing

amount, the currency, or the addressee. To allow full flexibility the search criteria should be definable by the user.

To determine which messages have been processed or in which state the messages are, all messages should be stored in a message warehouse with their corresponding state. In the message warehouse, a user must be able to search whether a message that matches certain criteria has already been processed or whether the message is currently being processed – for example accepted by the broker, sent to the external network, acknowledged by the external network, or similar.

There are different techniques that allow for the building of such a message warehouse. This article describes different ways to achieve this with WebSphere MQ Integrator Broker and DB2 UDB, and discusses advantages and disadvantages of the different methods.

DIRECT METHOD

To allow queries for specific information about messages, the simplest method is to create a database table that contains predefined search criteria.

The message flow that is processing the messages is extended with a WebSphere MQ Integrator Broker database node that extracts the relevant information from the message and stores it as a row in the database table. This row can then be updated as the message continues processing through the various stages in the broker. To allow this, the row may require some technical criteria in order to identify the correct row in the database table. Such a technical criterion could be the WebSphere MQ message id from the message descriptor (MQMD) of the first incoming message. It must be passed through the complete processing chain.

When searching for a message, a user can specify search arguments based on the fields stored in the database table. An application can take these search arguments and execute the necessary SQL SELECT statement that checks for entries with

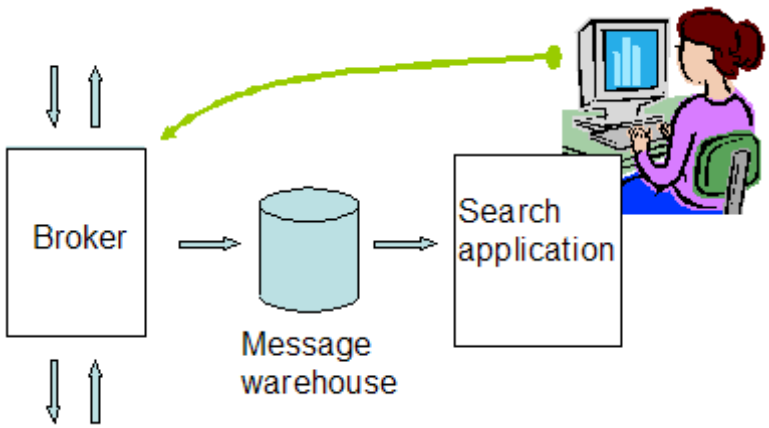

Figure 3: Message warehouse

the appropriate values. Depending on the results the user can identify the stage of the processing of his individual message and may take some action, if required. Such a process is shown in Figure 3.

There exist some variations in how and when the data is stored, perhaps with some transformations of the values in the message and the format that these values are stored in the message warehouse.

The direct method is easy to understand and very easy to implement. It requires either that the message format be defined in the message dictionary, for example in the WebSphere MQ Integrator Broker Message Repository Manager (MRM), or that the message be in a self-defined format, for example Extensible Mark-up Language (XML). This is necessary to be able to parse the messages flowing through the broker and to access the appropriate fields of the message that need to be stored in the database table. Parsing the message can add processing time and a processing delay if the message does not need to be parsed.

Storing only the relevant parts used for searching for messages in such a database table is very efficient in terms of the amount of data and the time required for storing entries in the message warehouse. Also, searching for specific messages in such a

database table could be relatively cheap, given that indexes can be used for common search patterns.

Even if simple and optimized for performance, this method has some drawbacks. It is very inflexible if search criteria need to be added or the database table should be used to store different kinds of messages. When changing a search criterion, either by changing an existing one or adding a new one, everything must be changed, including:

- The message flow that inserts the data into the database table.
- The definition of the database table.
- The application used to search for specific messages.

Each of the search criteria is defined at development time. A user has no opportunity to use others fields at search time to define his selection criteria.

In addition, there are problems with entries already existing in the database table. These do not have any values for added criteria, and a transformation of the existing criteria to the new ones may not be possible. They can also not be rebuilt because the messages themselves no longer exist. There are ways to reprocess messages stored in a message audit database, but this usually requires additional programming.

If there is the need to store information for different kinds of message, the direct method requires either many optional fields to be added into the database table or new tables. This makes the design much more complex or leads to inconsistent search criteria. Then, there is no longer a common format for the data and the search arguments.

WEBSPHERE MQ INTEGRATOR BROKER WAREHOUSE NODE

Another way to provide message warehouse functionality is by using the Warehouse node provided by WebSphere MQ Integrator Broker. This node is a specialized database node that can be

used as is to store the whole message or parts of the message in a database table. Storing parts of the message corresponds to the method described in the previous section.

The whole message is stored by converting the current message into a stream of binary data. This binary data is then stored in a BLOB column in the table. Together with this binary data, a timestamp can be stored. This method is relatively fast because the broker does not need to parse the message. There is also the advantage that the format of the message data does not have to be defined to the broker.

This process adds a large amount of unnecessary data to the database table. Searching within this data is not trivial, because searching within BLOB data itself is not supported by DB2. Therefore, any search can be done only by using a separate program that post-processes the messages in the message store. This structure is shown in Figure 4.

After the data is inserted into the message store, a second process is required that regularly scans the message store. It retrieves and processes the new records. Processing means that the program needs to parse the message data, extract the fields that should be used as search criteria, and insert them into a message warehouse database table as in the first method. The

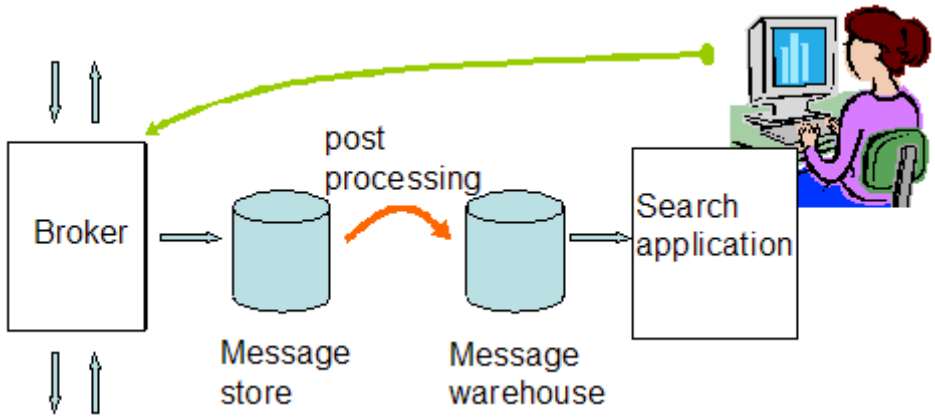

Figure 4: Message warehouse using post-processing

processing can be decoupled from the actual processing of the message, eg at off-peak times when sufficient processing capacity is available. Similar to the direct method this method is easy to implement and searching the post-processed data can be very effective.

Having the complete data in the database allows the search criteria to be altered, for example by reprocessing the message data in the database table. Changes to the database layout must be synchronized with the changes to the search application. The message flow is not affected by any such change.

With this method there are also some drawbacks. The most important one is that there is a time delay until the information is available for searching. Another disadvantage is that the post-processing program must be capable of parsing the message. If this is also required in the WebSphere MQ Integrator Broker environment, there are potentially two different locations where the message format has to be defined. If different types of messages are stored using this method, the post-processing program must be able to parse different message formats.

COMBINED METHOD

It is possible to combine the WebSphere MQ Integrator Broker Warehouse node method with the direct method described in the previous section. Using this method, the message flow already extracts search criteria from the message and stores them together with the complete message data in a database table. This combines some of the advantages, but also some of the disadvantages, of both methods.

This method generates good access performance to the search criteria and is easy to implement. If a new search criterion is needed, the message data is still available to be reprocessed.

On the other hand, changes to the search criteria require that changes to the message flow, the search application, and the database layout have to be synchronized. If there are different message types that need to be stored in the message warehouse

table, different search criteria are usually needed. This can result in a lot of unused fields in a row. The search criteria are already pre-defined to the fields extracted by the message flow before the entry is added to the table. To be able to extract the values, the message layout must be defined to the broker.

The amount of data that is stored in the message warehouse is much more than in the direct method because the complete message data is stored in addition to the search criteria.

USING DB2 XML EXTENDER

In contrast to the methods above, WebSphere BI for FN defines a common message format for its message warehouse. All messages of any message type have to be converted into this common message format and will then be stored in the message warehouse database table. The format used is the Extensible Mark-up Language (XML).

XML is a simple, very flexible, text format to store structured data like financial transactions, addresses, and so on. It is easy to learn even for people who are not programmers. Although XML is not meant to be read by humans, people can read it using their favourite text editor. Take a look at the following simple example. It shows how XML can be used to store people's addresses:

```
<address>
      <forename>Mike</forename>
      <surname>Smith</surname>
      <street>Oak Avenue 2</street>
      <postal>12345</postal>
      <city>Oakville</city>
      <state>Oakland</state>
</address>
```

The XML syntax uses tags to represent the structure and values of the data. In the example, an address consists of the forename, surname, street, postal, city, and state information. The value of the forename is Mike, because it is enclosed by the start tag <forename> and the end tag </forename>.

WebSphere BI for FN uses XML for its common message format

because it is standardized, easy to use, and many people know it today. Additionally, many off-the-shelf tools are available to handle this format, giving customers the choice of tool with which to accomplish their tasks.

Nevertheless, to provide customers with a running solution out-of-the-box, WebSphere BI for FN uses the off-the-shelf tool DB2 XML Extender (for more details see http://www.ibm.com/software/data/db2/extenders/xmlext/index.html) to offer the search functionality for user-defined search criteria.

The message warehouse component of WebSphere BI for FN consists mainly of three components – a message flow to convert the incoming message into XML, a database table to store the converted message, and the DB2 XML Extender, or other XML tools, to search for specific information within the stored messages.

The message flow uses the ability of the WebSphere MQ Integrator Broker to convert messages from any parser domain into another parser domain. This is done within a compute node by changing the parser domain of the output message to XML using the following two ESQL statements:

```
SET OutputRoot.MQRFH2.Format = 'XML';
SET OutputRoot.MQRFH2.mcd.Msd = 'xml';
```

Each child of the input message, represented in WebSphere MQ Integrator Broker as InputRoot, can have its own parser, for example the message descriptor (MQMD), request, and the formatting header (MQRFH2). To ensure that each child element of InputRoot is correctly copied to the XML part of the output message OutputRoot.XML.Msg, each child element must be copied separately. This can be done with the following ESQL statements:

```
DECLARE I INTEGER;
SET I = 1;
DECLARE C INTEGER;

SET C = CARDINALITY(InputRoot.*[]);
WHILE I <= C DO
  SET OutputRoot.XML.Msg.*[I] = InputRoot.*[I];
  SET I=I+1;
END WHILE;
```

If the parser domain of a child element is not known by WebSphere MQ Integrator Broker, it cannot be converted into XML. Such child elements are copied as BLOB elements into the output message.

By using this approach it is possible to provide a message warehouse that is independent of the message types that should be stored. It also means that an Extension of WebSphere BI for FN needs to provide its own parser or MRM definitions if it handles message types not basically known by WebSphere MQ Integrator Broker.

After the message is converted into XML format it is stored in a database table. This database table contains an XMLCLOB column to support the DB2 XML Extender. XMLCLOB is a user-defined data type provided by the DB2 XML Extender.

Storage of the message in the database table is done in the compute node. The XML part of the message is put into the database table. Before this can be done, that part of the message must be converted into a flat text string. This can be accomplished, for example, with the following ESQL statements:

```
DECLARE chXmlMsg CHARACTER;

SET chXmlMsg = CAST( BITSTREAM( OutputRoot.XML ) AS CHARACTER
                     CCSID 1208
                     ENCODING MQENC_NATIVE );
```

Now, the field chXmlMsg contains the whole original message converted into XML surrounded by the start tag <Msg> and the end tag </Msg>. Note that the surrounding XML tag is necessary, because the XML specification requires that an XML document be allowed to have only one root element. The data can then be stored in the XMLCLOB column using an INSERT statement.

The great benefit of this approach is that all messages are stored in one common message format and the user is able to use the DB2 XML Extender to search for messages containing any kind of search criterion. The following example shows how many messages were originated by user Mike in the Message Warehouse database table DNI.DNI_MWH_OU1:

```
SELECT COUNT(*) FROM DNI.DNI_MWH_OU1
WHERE DB2XML.extractChar(MWH_XML_MSG, '/Msg/MQMD/UserIdentifier') =
      'Mike'
```

Using this method, a user can define searches on any field in the message. The search criteria do not have to be pre-defined at development time. To allow for this, the user must know about the structure of the messages he is searching for.

Furthermore, no additional programming logic is needed to query the message warehouse for specific information. All queries can be performed using just SQL.

The drawback with this method is that additional processing time is needed to convert the messages into XML format, and that no dedicated database table columns exist for a fast search. The latter problem can be solved using side table definitions for the DB2 XML Extender at the cost of insert and update performance. Side tables are additional database tables that the XML Extender creates to improve performance when searching elements or attributes in an XML column. A Document Access Definition (DAD) file describes which XML elements have to be extracted and which columns they are stored in in the side tables. The search performance can further be improved by indexing the most frequently-used columns of the side tables. This allows users to optimize the search performance for their specific search criteria.

Another disadvantage of this method compared with the previous methods is that the amount of data stored is much greater. The main reason is that most data used for messages is in a structure that is very compact. XML usually requires much more space because each field is surrounded by a start tag and an end tag.

COMMON CHARACTERISTICS

The message warehouse methods described above for building the message warehouse for messages processed in WebSphere MQ Integrator Broker consider just storing the information in the message warehouse and searching in the message warehouse. To search for entries within the message warehouse, users need

to provide their own program. After the search is complete a user may want to perform actions to resolve a problem. An example could be when a user detects that his message has not yet arrived at the required destination, for example because the session to that destination is not available. In this case the user may decide to use message administration functions to route the message to the destination via a different path . In case of a payment, the message may be sent using a fax instead of the SWIFT network. Such types of application need to be interfaced with the search program for the message warehouse. The interface between such programs usually uses some technical attributes. Therefore, such attributes, perhaps with some state information, have to be stored in the message warehouse independently of the chosen solution.

Another common issue for all solutions is that the amount of data is continually growing. Specific messages are usually looked for shortly after the message has been sent. After a defined time, the data in the message warehouse is usually no longer used to find the status of individual messages. Instead the data may be used to get statistical data, such as the total amount of money transferred to a specific addressee. Depending on the utilization of the information in the message warehouse, some maintenance needs to be carried out on the data. This requires, for example, archiving the message warehouse data for later use or deleting data that is no longer used for searching.

SUMMARY

There are different methods for building a message warehouse in the WebSphere MQ Integrator Broker environment and using DB2 UDB. Each of them has its own advantages and disadvantages. A user can make the choice between these different methods based on his requirements for performance, flexibility, and associated costs. As it stores only the minimum required information, the direct method is the fastest. The method provided by WebSphere BI for FN's message warehouse nodes using DB2 XML Extender is the most flexible.

Michael Groetzner
IBM (Germany)

MQJavaRoundTrip: a Java-based performance tester

WHY USE IT? (PROBLEMS SOLVED)

When rolling out a new message queueing deliverable or prototype that involves the movement of messages between queues, it would be advantageous to be able to both performance test and stress test the new system or prototype. This could be particularly useful when implementing a WebSphere Business Integration Message Broker solution on top of WebSphere MQ. When using a message flow within WBIMB, which reads a message from an input queue, performs some operation(s) on that message, and then places it onto an output queue, it is often difficult to predict exactly how quickly messages can be processed.

The performance of such flows is often a key requirement, and being able easily to measure performance outside of the production environment is a major enabler of a good estimate.

The MQJavaRoundTrip application is a simple Java-based tool that connects to WMQ, puts a message in a queue, and waits for a reply on another queue, while reporting throughput during the test.

MQJavaRoundTrip will, therefore, allow a more scientific figure of potential throughput to be obtained.

HOW TO USE IT

Before executing MQJavaRoundTrip, set-up information needs to be obtained, such as the name of the queue manager your application is connected to, the queues messages are put to and got from by the client, and the input message to be used. Note: some systems may have several different input messages – MQJavaRoundTrip uses one message only, and is intended to be used to obtain ballpark figures. It may be necessary to run

MQJavaRoundTrip several times, specifying a different input message each time in order to obtain a more accurate throughput figure.

Note: if a usable MQ message is not yet available, for example the raw data is available but required message headers (MQRFH2) have not been set, a useful tool to allow the simple construction of an MQ message is RFHUTIL – available from IBM's WebSphere MQ SupportPac Web site at http://www-306.ibm.com/software/integration/support/supportpacs/category.html#cat2 – IH03.

MQJavaRoundTrip is run from the command line and a number of arguments are required. A detailed breakdown of the arguments can be found in the section *ARGUMENT BREAKDOWN*.

When running MQJavaRoundTrip, a number of decisions need to be made. MQJavaRoundTrip is a multithreaded application in that it can spawn several client threads capable of producing messages and putting them in the input queue and retrieving the subsequent reply from the output queue. Deciding on the number of threads to run depends largely on the desired findings. If the maximum possible throughput is sought, it may be necessary to run MQJavaRoundTrip several times with increasing numbers of threads until the throughput rate doesn't increase. This will be the point at which the application being tested is running at its full potential.

It is also necessary to decide on how to connect to the queue manager: whether to connect directly to the API using binding connections or whether to connect as an MQ client.

The persistence of the message, along with other message specifics such as the character set of the character data within the message and the encoding of the numeric data within the message, needs to be specified.

These decisions should be made based on what a fully-working version of the system would use. For example, if within the end system the connection to the queue manager would be made via an MQSeries Java Client, then a client connection should be used for MQJavaRoundTrip.

Once MQJavaRoundTrip has been executed it will print out confirmation of the connection to the queue manager and queues. Depending on the interval argument specified, MQJavaRoundTrip will then begin intermittently to print out the message rates per second. Once the test has completed, a final printout of average throughput will be produced. Note: this doesn't take into account warm-up of the messaging transport.

ARGUMENT BREAKDOWN

Running MQJavaRoundTrip with no arguments specified displays the following output. This details all of the possible command line arguments:

```
usage: java MQJavaRoundTrip <args>
 Controller properties:
 -n <clients> (parallel threads)
 -d <duration> (in seconds - 0=never stop)
 -z (quiet, minimal output)
 -v <interval> (statistics interval)
 Queue manager properties:
 -qm <qm> (queue manager name)
 -i <queue> (put requests to)
 -o <queue> (get replies from)
 Connection properties for Bindings :
 -b (use local bindings connection)
 -t (use trusted/fastpath bindings, must use -b too)
 Connection properties for TCP/IP client :
 -h <hostname> (QM machine)
 -c <channel> (QM svrconn - defaults to SYSTEM.DEF.SVRCONN)
 -p <port> (QM listener port - defaults to 1414)
 Message properties:
 -f <file> (user data to send)
 -x (use transactions one-per-message)
 -s (use persistent messages)
 -mc (message character set)
 -me (message encoding)
 -mf (message format)
 -id (use get-correlation-id from sent-mesg-id)
 -gid (fixed correlation id per thread)
 -w <timeout> (message wait timeout in seconds)
```

where:

- -n <clients> allows the user to specify the number (the

default is 1) of client threads that the application should spawn. Each thread will execute its put-get loop in parallel.

- -d <duration> allows the user to specify in seconds how long the test should run. Once the duration has passed, the application will terminate and remove any handles on queues and queue managers. By specifying a value of 0, the test will run indefinitely.
- -z allows the user to specify that only the final average throughput rate should be output.
- -v <interval> allows the user to specify in seconds the interval at which the throughput rate is output. The default value is 10 seconds.
- -qm <qm> allows the user to specify which WebSphere MQ queue manager to connect to.
- -i <queue> allows the user to specify the name of the queue, in the specified queue manager, to which MQJavaRoundTrip will put its requests.
- -o <queue> allows the user to specify the name of the queue, in the specified queue manager, from which MQJavaRoundTrip will wait for replies.
- -b allows the user to specify whether to connect to the queue manager through local bindings (the default is via TCP/IP client connections). This gives faster results but is only applicable if the client is on the same hardware as the queue manager.
- -t allows the user to specify whether the binding connection being used is to be trusted, also known as fastpath bindings. This option will increase the performance at the cost of reduced safety checking. It is not usually recommended for a production system, but may help with load generation from a performance testing client.
- -h <hostname> allows the user to specify the name/IP address of the machine that is hosting the queue manager.

This is relevant only for client connections. Note: if the –b flag is specified all client connection settings are ignored.

- -c <channel> allows the user to specify the channel to be used when connecting to the queue manager (the default is SYSTEM.DEF.SVRCONN). This is relevant only for client connections. Note: if the –b flag is specified all client connection settings are ignored.
- -p <port> allows the user to specify which port to use when connecting to the queue manager (the default is 1414). This is relevant only for client connections. Note: if the –b flag is specified all client connection settings are ignored.
- -f <file> allows the user to specify the full path of the file to be used as the basis of the request message. This should be in the format of x:\temp\filename.msg.
- -x allows the user to specify that messages should be put and got transactionally (the default is to send non-transactionally). If transactional, the client loop will be: put, commit, get, commit.
- -s allows the user to specify that sent messages should be persistent (the default is to send non-persistent).
- -mc <ccsid> allows the user to explicitly specify which character set identifier (CCSID) is set on sent messages (the default is to use the CCSID of the queue manager). See the WebSphere MQ *Using Java* manual for details if this applies to your scenario.
- -me <??> allows the user to specify which representation should be used for numeric values within the message data. See the WebSphere MQ *Using Java* manual for details if this applies to your scenario.
- -mf <format> allows the user to specify to the receiving application (the application reading the message from the input queue) the nature of the data within the message. Typical examples are MQRFH2, MQSTR, or XML. See the

WebSphere MQ Using *Java manual* for details if this applies to your scenario.

- -id informs the tool that replies will have their correlation-id set to the message-id of the request message. In order for this to work the server process under test must adhere to the same policy (ie it must copy the correlation-id into place from the message-id). The default is to do nothing with correlation identifiers.
- -gid informs the tool to generate a correlation-id for each thread and specify it in the request message. Reply messages must then have the same correlation-id. In order for this to work the server process under test must adhere to the same policy (ie it must duplicate the correlation-id of the request message).
- -w allows the user to specify how long the tool should wait for a reply to the message it originally put in seconds. If the application being tested is likely to take 10 seconds to process a single message the wait time should be specified as ~10 seconds. (the default is 5 seconds).

An example of how to use MQJavaRoundTrip:

```
java MQJavaRoundTrip -n 10 -d 60 -qm TEST_QM -i IN.QUEUE -o OUT.QUEUE -h
127.0.0.1 -c JAVA.CHANNEL -p 1414 -f inputmessage.xml -x -s -id
```

This will run for 60 seconds with 10 threads connecting via TCP/IP port 1414 to TEST_QM (hosted on 127.0.0.1). They will put persistent, transacted messages onto IN.QUEUE and wait for replies from OUT.QUEUE. They are expecting those replies to have a correlation-id matching the message-id of the request

CAVEAT

This program is an excellent starting place for investigating the potential throughput of WebSphere MQ applications that move messages from one queue to another. Take care that the settings of this tool do match the scenario you are expecting. Incorrectly specifying message persistence, bindings connections or suchlike can give markedly different performance, both higher and lower.

CODE

MQJavaRoundTrip consists of two classes. The first, MQJavaRoundTrip, is the main class and as such contains the main method required for executing a Java application.

This class contains all of the code for setting up the connections to the queues and queue manager etc. The second class, ClientThread, is instantiated by the MQJavaRoundTrip class *x* number of times, where *x* is the number of threads specified by the user.

It is then the individual instances that do the actual putting and getting to the queues.

MQJavaRoundTrip

```
import com.ibm.mq.*;
import java.io.*;
import java.util.ArrayList;
import java.util.Properties;
import java.util.Iterator;
public class MQJavaRoundTrip extends Thread {
	private static MQJavaRoundTrip instance = null;
	private boolean shutdown = false;
	private final ArrayList workers = new ArrayList();
	private Thread stats = null;
	final Properties arguments = new Properties();
	private byte[] inputFileBytes = null;
	private final java.text.NumberFormat numberFormat =
		java.text.NumberFormat.getInstance();
	boolean verbose = true;
	private MQJavaRoundTrip() {
		super("MQJavaRoundTrip");
		numberFormat.setMinimumFractionDigits(2);
		numberFormat.setMaximumFractionDigits(2);
		numberFormat.setGroupingUsed(false);
	}
	public static MQJavaRoundTrip getInstance() {
		if (instance == null) {
			instance = new MQJavaRoundTrip();
		}
		return instance;
	}
	void signalShutdown() {
		shutdown = true;
		interrupt();
```

```
}
public static void main(String args[]) {
      getInstance().parseArguments(args);
      getInstance().start();
}
public void run() {
      try {
            {
            if ( verbose ) System.out.println("Testing settings");
                  MQQueueManager qm = getQueueManagerConnection();
         if ( verbose ) System.out.println("Connection to QM OK");
                  MQQueue putQueue =
                        qm.accessQueue(
                              arguments.getProperty("i"),
                              MQC.MQOO_OUTPUT
                                    | MQC.MQOO_INQUIRE
                                    | MQC.MQOO_FAIL_IF_QUIESCING);
                  MQQueue getQueue =
                        qm.accessQueue(
                              arguments.getProperty("o"),
                              MQC.MQOO_INPUT_SHARED
                                    | MQC.MQOO_INQUIRE
                                    | MQC.MQOO_FAIL_IF_QUIESCING);
     if ( verbose ) System.out.println("Connection to Queues OK");
                  getQueue.close();
                  putQueue.close();
                  qm.disconnect();
            }
            readDataFromFile();
            createClientThreads();
            startStatisticsThread();
            startClientThreads();
            // Wait for the end fo the test
int duration = Integer.parseInt(arguments.getProperty("d")) * 1000;
            if (duration <= 0) {
                  ((Thread) workers.get(0)).join();
            } else {
                  Thread.sleep(duration);
            }
      } catch (InterruptedException e) {
            // No-op
      } catch (Throwable e) {
            // Display all other errors
            System.out.println(e);
            e.printStackTrace();
      } finally {
            shutdown = true;
  if ( verbose ) System.out.println("Instructing threads to stop");
            if (stats != null)
                  stats.interrupt();
```

```
            // signal workers to die ....
            Iterator iter = workers.iterator();
            while (iter.hasNext()) {
                ClientThread worker = (ClientThread) iter.next();
                  worker.shutdown = true;
              worker.interrupt(); // wake them if they are waiting
            } // end while workers
         if ( verbose ) System.out.println("Waiting for threads");
            iter = workers.iterator();
            try {
                  while (iter.hasNext())
                         ((Thread) iter.next()).join();
            } catch (InterruptedException e) {}
            // Collect overall message rates
            iter = workers.iterator();
            double rate = 0;
            while (iter.hasNext()) {
                 ClientThread worker = (ClientThread) iter.next();
                 rate += (double) (worker.numberOfMessages * 1000)
                        / worker.runlength;
              worker.interrupt(); // wake them if they are waiting
            } // end while workers
            System.out.println(
                  "Total rt/sec=" + numberFormat.format(rate));
      } // end try/catch/finally
} // end run()
private void startClientThreads() throws Exception {
      Iterator iter = workers.iterator();
      while (iter.hasNext()) {
            ClientThread worker = (ClientThread) iter.next();
            Thread.sleep(250 + (int) (Math.random() * 500));
            worker.start();
      } // end while
} // end startClientThreads
MQQueueManager getQueueManagerConnection() throws MQException {
      int connectionoptions = MQC.MQCNO_NONE;
      boolean bindings = arguments.getProperty("b") != null;
      if (bindings) {
            boolean fastpath = arguments.getProperty("f") != null;
            if (fastpath) {
                  connectionoptions = MQC.MQCNO_FASTPATH_BINDING;
            } else {
                  connectionoptions = MQC.MQCNO_STANDARD_BINDING;
            } // End elseif fastpath bindings
      } else { // End if bindings
            // if client connections
            MQEnvironment.hostname = arguments.getProperty("h");
            MQEnvironment.channel = arguments.getProperty("c");
MQEnvironment.port = Integer.parseInt(arguments.getProperty("p"));
      }
```

```
        return new MQQueueManager(
              arguments.getProperty("qm"),
              connectionoptions);
  } // End getQueueManagerConnection
  private void createClientThreads() throws Exception {
        int nThreads = Integer.parseInt(arguments.getProperty("n"));
        for (int i = 1; i <= nThreads; i++) {
              String name = "Client" + i;
              if (arguments.getProperty("gid") != null) {
     byte addr[] = java.net.InetAddress.getLocalHost().getAddress();
                    byte bytes[] = new byte[24];
                   System.arraycopy(addr, 0, bytes, 0, addr.length);
                    bytes[20] = (byte) i;
                    bytes[21] = (byte) (i >> 8);
                    bytes[22] = (byte) (i >> 16);
                    bytes[23] = (byte) (i >> 24);
                    workers.add(new ClientThread(name, bytes));
              } else {
                    workers.add(new ClientThread(name));
              }
        } // end foreach thread
  } // end createClientThreads
  private void readDataFromFile() throws Exception {
        File theFileToRead = new File(arguments.getProperty("f"));
        inputFileBytes = new byte[(int) theFileToRead.length()];
        BufferedInputStream bIS =
        new BufferedInputStream(new FileInputStream(theFileToRead));
        bIS.read(inputFileBytes);
        bIS.close();
  } // End readDataFromFile
  // Create a new copy of the message
  MQMessage createMessage() throws Exception {
        MQMessage putMessage = new MQMessage();
        if (arguments.getProperty("mc") != null)
              putMessage.characterSet =
                    Integer.parseInt(arguments.getProperty("mc"));
        if (arguments.getProperty("me") != null)
putMessage.encoding = Integer.parseInt(arguments.getProperty("me"));
        if (arguments.getProperty("mf") != null)
              putMessage.format = arguments.getProperty("mf");
        putMessage.messageFlags = MQC.MQMT_REQUEST;
        if (arguments.getProperty("s") != null) {
              putMessage.persistence = MQC.MQPER_PERSISTENT;
        } else {
              putMessage.persistence = MQC.MQPER_NOT_PERSISTENT;
        }
        putMessage.write(inputFileBytes);
        return putMessage;
  } // end createMessage
  private void parseArguments(String[] args) {
```

```
            String currentkey = null;
            String currentvalue = "";
            arguments.setProperty("c", "SYSTEM.DEF.SVRCONN");
            arguments.setProperty("p", "1414");
            arguments.setProperty("n", "1");
            arguments.setProperty("d", "60");
            arguments.setProperty("w", "5");
            if (args.length == 0)
                  printUsage();
            for (int i = 0; i < args.length; i++) {
                  if (args[i].startsWith("-")) {
                        if (currentkey != null) {
                        arguments.setProperty(currentkey, currentvalue);
                        }
                        currentkey = args[i].substring(1);
                        currentvalue = "true";
                  } else if (currentvalue.equals("true")) {
                        currentvalue = args[i];
                  }
            } // end for
            if (currentkey != null)
                  arguments.setProperty(currentkey, currentvalue);
            if ( arguments.getProperty("z")!=null ) verbose = false;
      } // end parseArguments
      private void printUsage() {
            System.out.println(
                "usage: java " + this.getClass().getName() + " <args>");
            System.out.println(" Controller properties:");
            System.out.println(" -n <clients>     (parallel threads)");
    System.out.println(" -d <duration>    (in seconds - 0=never stop)");
        System.out.println(" -z                (quiet, minimal output)");
          System.out.println(" -v <interval>    (statistics interval)");
            System.out.println(" Queue manager properties:");
          System.out.println(" -qm <qm>          (queue manager name)");
            System.out.println(" -i <queue>       (put requests to)");
            System.out.println(" -o <queue>       (get replies from)");
            System.out.println(" Connection properties for Bindings :");
System.out.println(" -b               (use local bindings connection)");
            System.out.println(
  " -t               (use trusted/fastpath bindings, must use -b too)");
       System.out.println(" Connection properties for TCP/IP client :");
            System.out.println(" -h <hostname>    (QM machine)");
            System.out.println(
     " -c <channel>     (QM svrconn - defaults to SYSTEM.DEF.SVRCONN)");
            System.out.println(
             " -p <port>        (QM listener port - defaults to 1414)");
            System.out.println(" Message properties:");
            System.out.println(" -f <file>        (user data to send)");
            System.out.println(
                " -x               (use transactions one-per-message)");
```

```
  System.out.println(" -s               (use persistent messages)");
    System.out.println(" -mc              (message character set)");
        System.out.println(" -me              (message encoding)");
        System.out.println(" -mf              (message format)");
        System.out.println(
   " -id              (use get-correllation-id from sent-mesg-id)");
        System.out.println(
            " -gid             (fixed correllation id per thread)");
        System.out.println(
             " -w <timeout>     (message wait timeout in seconds)");
        System.exit(0);
  } // end printUsage
  private final void startStatisticsThread() throws Throwable {
        stats = new Thread("Stats") {
              private int prev[];
              private int curr[] = null;
              private void getValues() {
                    // Get a snapshot of current "total messages"
                    prev = curr;
                    curr = new int[workers.size()];
                    int i = 0;
                    Iterator iter = workers.iterator();
                    while (iter.hasNext() && i < curr.length) {
                   ClientThread worker = (ClientThread) iter.next();
                          curr[i++] = worker.numberOfMessages;
                    }
              }
              public void run() {
                    int diff;
                    int total;
final int interval = Integer.parseInt( arguments.getProperty("v") );
                    final StringBuffer sb = new StringBuffer();
                    getValues();
                    while (!shutdown) {
                          sb.setLength(0);
                          try {
                                Thread.sleep(interval * 1000);
                                getValues();
                                total = 0;
                         // Calculate difference from last snapshot
                              for (int j = 0; j < curr.length; j++) {
                                      diff = curr[j] - prev[j];
                                      total += diff;
                                }
                                sb.append("rt/sec=").append(
                     numberFormat.format((double) total / interval));
                                System.out.println(sb);
                          } catch (InterruptedException e) {}
                    } // end while
              } // end run
```

```
            }; // end inner stats class
            stats.setDaemon(true);
            stats.start();
      } // end startstatisticsthread
} // End of file
ClientThread:
import java.util.Date;
import com.ibm.mq.*;
public class ClientThread extends Thread {
      private byte[] fixedCorrelId = null;
      private final static MQJavaRoundTrip parent =
MQJavaRoundTrip.getInstance();
      int numberOfMessages = 0;
      boolean shutdown = false;
      long runlength = 0;
      public ClientThread(String name) {
            super(name);
      }
      public ClientThread(String name, byte[] correlId) {
            super(name);
            this.fixedCorrelId = correlId;
      }
      public void run() {
            MQQueueManager queueManager = null;
            MQQueue putQueue = null;
            MQQueue getQueue = null;
       if ( parent.verbose ) System.out.println(getName() + " : START");
         boolean transacted = parent.arguments.getProperty("x") != null;
            boolean useCorrelId =
                  (parent.arguments.getProperty("id") != null)
                        || (fixedCorrelId != null);
            boolean copyCorrelId =
                  (parent.arguments.getProperty("id") != null)
                        && (fixedCorrelId == null);
            // SETUP MESSAGES AND OPTIONS
            MQPutMessageOptions pmo = new MQPutMessageOptions();
            MQGetMessageOptions gmo = new MQGetMessageOptions();
            gmo.waitInterval =
             Integer.parseInt(parent.arguments.getProperty("w")) * 1000;
            gmo.matchOptions =
                 useCorrelId ? MQC.MQMO_MATCH_CORREL_ID : MQC.MQMO_NONE;
            gmo.options = MQC.MQGMO_WAIT | MQC.MQGMO_FAIL_IF_QUIESCING;
            gmo.options |= transacted
                  ? MQC.MQGMO_SYNCPOINT
                  : MQC.MQGMO_NO_SYNCPOINT;
       pmo.options = MQC.MQPMO_FAIL_IF_QUIESCING | MQC.MQPMO_NEW_MSG_ID;
            pmo.options |= transacted
                  ? MQC.MQPMO_SYNCPOINT
                  : MQC.MQPMO_NO_SYNCPOINT;
            long startTime = 0;
```

```
            try {
                  MQMessage outMessage = parent.createMessage();
                  MQMessage inMessage = new MQMessage();
                  if (fixedCorrelId != null) {
                        outMessage.correlationId = fixedCorrelId;
                        inMessage.correlationId = fixedCorrelId;
                  }
                  // Make connections to WMQ resources
                  queueManager = parent.getQueueManagerConnection();
                  putQueue =
                        queueManager.accessQueue(
                              parent.arguments.getProperty("i"),
                          MQC.MQOO_OUTPUT | MQC.MQOO_FAIL_IF_QUIESCING);
                  getQueue =
                        queueManager.accessQueue(
                              parent.arguments.getProperty("o"),
                        MQC.MQOO_INPUT_SHARED | MQC.MQOO_FAIL_IF_QUIESCING);
    if ( parent.verbose ) System.out.println(getName() + ": connected");
                  startTime = new Date().getTime();
                  while (!shutdown) {
                        try {
                              putQueue.put(outMessage, pmo);
                              if (transacted) queueManager.commit();
       if (copyCorrelId) inMessage.correlationId = outMessage.messageId;
                              getQueue.get(inMessage, gmo);
                              if (transacted) queueManager.commit();
                              numberOfMessages++;
                        } catch (MQException e) {
                  if (e.reasonCode == MQException.MQRC_NO_MSG_AVAILABLE)
                                    System.err.println(
                                          getName() + " : Cannot see a
response to message "+outMessage.messageId);
                              throw e;
                        } // End try catch
                  } // End while ! shutdown
            } catch (Throwable e) {
                  // Handle a fatal error
                  System.err.println(
              getName() + " : Fatal Error.  Exception follows: \n" + e);
                  parent.signalShutdown();
            } finally {
                  // Clear up code carefully in fair weather or foul.
                  runlength = new Date().getTime() - startTime;
                  if (getQueue != null) {
                        if ( parent.verbose ) System.out.println(
                       getName() + " : Closing queue " + getQueue.name);
                        try {
                              getQueue.close();
                        } catch (MQException e) {} finally {
                              getQueue = null;
```

```
                        }
                  }
                  if (putQueue != null) {
                        if ( parent.verbose ) System.out.println(
                       getName() + " : Closing queue " + putQueue.name);
                        try {
                              putQueue.close();
                        } catch (MQException e) {} finally {
                              putQueue = null;
                        }
                  }
                  if (queueManager != null) {
                        if ( parent.verbose ) System.out.println(
                              getName()
                                    + " : Closing queue manager "
                                    + queueManager.name);
                        try {
                              queueManager.disconnect();
                        } catch (MQException e) {} finally {
                              queueManager = null;
                        }
                  }
        if ( parent.verbose ) System.out.println(getName() + " : STOP");
            } // End try/catch/finally
      } // End run method
}
```

Clientthread.java

```
import java.util.Date;
import com.ibm.mq.*;
//import  com.ibm.mq.jms.services.ConfigEnvironment;
public class ClientThread extends Thread {
      private byte[] fixedCorrelId = null;
      private final static MQJavaRoundTrip parent =
MQJavaRoundTrip.getInstance();
      int numberOfMessages = 0;
      boolean shutdown = false;
      long runlength = 0;
      public ClientThread(String name) {
            super(name);
      }
      public ClientThread(String name, byte[] correlId) {
            super(name);
            this.fixedCorrelId = correlId;
      }
      public void run() {
            MQQueueManager queueManager = null;
            MQQueue putQueue = null;
```

```
        MQQueue getQueue = null;
    if ( parent.verbose ) System.out.println(getName() + " : START");
      boolean transacted = parent.arguments.getProperty("x") != null;
        boolean useCorrelId =
              (parent.arguments.getProperty("id") != null)
                    || (fixedCorrelId != null);
        boolean copyCorrelId =
              (parent.arguments.getProperty("id") != null)
                   && (fixedCorrelId == null);
        // SETUP MESSAGES AND OPTIONS
        MQPutMessageOptions pmo = new MQPutMessageOptions();
        MQGetMessageOptions gmo = new MQGetMessageOptions();
        gmo.waitInterval =
         Integer.parseInt(parent.arguments.getProperty("w")) * 1000;
        gmo.matchOptions =
             useCorrelId ? MQC.MQMO_MATCH_CORREL_ID : MQC.MQMO_NONE;
        gmo.options = MQC.MQGMO_WAIT | MQC.MQGMO_FAIL_IF_QUIESCING;
        gmo.options |= transacted
              ? MQC.MQGMO_SYNCPOINT
              : MQC.MQGMO_NO_SYNCPOINT;
    pmo.options = MQC.MQPMO_FAIL_IF_QUIESCING | MQC.MQPMO_NEW_MSG_ID;
        pmo.options |= transacted
              ? MQC.MQPMO_SYNCPOINT
              : MQC.MQPMO_NO_SYNCPOINT;
        long startTime = 0;
        try {
              MQMessage outMessage = parent.createMessage();
              MQMessage inMessage = new MQMessage();
              if (fixedCorrelId != null) {
                    outMessage.correlationId = fixedCorrelId;
                    inMessage.correlationId = fixedCorrelId;
              }
              // Make connections to WMQ resources
              queueManager = parent.getQueueManagerConnection();
              putQueue =
                    queueManager.accessQueue(
                          parent.arguments.getProperty("i"),
                      MQC.MQOO_OUTPUT | MQC.MQOO_FAIL_IF_QUIESCING);
              getQueue =
                    queueManager.accessQueue(
                          parent.arguments.getProperty("o"),
                  MQC.MQOO_INPUT_SHARED | MQC.MQOO_FAIL_IF_QUIESCING);
if ( parent.verbose ) System.out.println(getName() + ": connected");
              startTime = new Date().getTime();
              while (!shutdown) {
                    try {
                          putQueue.put(outMessage, pmo);
                          if (transacted) queueManager.commit();
    if (copyCorrelId) inMessage.correlationId = outMessage.messageId;
                          getQueue.get(inMessage, gmo);
```

```
                              if (transacted) queueManager.commit();
                              numberOfMessages++;
                        } catch (MQException e) {
                  if (e.reasonCode == MQException.MQRC_NO_MSG_AVAILABLE)
                                    System.err.println(
                                          getName() + " : Cannot see a
response to message "+outMessage.messageId);
                              throw e;
                        } // End try catch
                  } // End while ! shutdown
            } catch (Throwable e) {
                  // Handle a fatal error
                  System.err.println(
              getName() + " : Fatal Error.  Exception follows: \n" + e);
                  parent.signalShutdown();
            } finally {
                  // Clear up code carefully in fair weather or foul.
                  runlength = new Date().getTime() - startTime;
                  if (getQueue != null) {
                        if ( parent.verbose ) System.out.println(
                       getName() + " : Closing queue " + getQueue.name);
                        try {
                              getQueue.close();
                        } catch (MQException e) {} finally {
                              getQueue = null;
                        }
                  }
                  if (putQueue != null) {
                        if ( parent.verbose ) System.out.println(
                       getName() + " : Closing queue " + putQueue.name);
                        try {
                              putQueue.close();
                        } catch (MQException e) {} finally {
                              putQueue = null;
                        }
                  }
                  if (queueManager != null) {
                        if ( parent.verbose ) System.out.println(
                              getName()
                                    + " : Closing queue manager "
                                    + queueManager.name);
                        try {
                              queueManager.disconnect();
                        } catch (MQException e) {} finally {
                              queueManager = null;
                        }
                  }
                  if ( parent.verbose ) System.out.println(getName() + "
: STOP");
```

```
            } // End try/catch/finally
        } // End run method
}
```

Kevin Braithwaite (braithwa@uk.ibm.com)
WebSphere Business Integration Message Broker Performance Specialist
IBM (UK)
Marc Carter (mcarter@uk.ibm.com)
WebSphere MQ/JMS Performance Specialist
IBM (UK)

A command server for MQMONNTP

The WebSphere MQ SupportPac MO71 (WebSphere MQ for Windows – GUI Administrator) is a simple yet powerful tool for monitoring and administering WebSphere MQ. It is referred to as MQMONNTP in the *User Guide* for the SupportPac. MQMONNTP may be downloaded from http://www-3.ibm.com/software/ts/mqseries/txppacs/mo71.html.

Although MQMONNTP runs on Windows platforms, it can connect to queue managers on any platform that will accept client connections. Platforms that do not accept client connections (typically OS/390) can be administered with another queue manager. When connection is via another queue manager WebSphere MQ objects can be displayed and modified, but messages cannot normally be browsed, copied, moved, or deleted. The *User Guide* states, 'It is possible to browse messages on other queue managers but you need a program on the remote system that understands the PCF messages generated by MQMONNTP'. Programs CmdSvr and CmdSvr2 have been written to fill this gap.

HOW IT WORKS

By way of example, let us assume that the queue manager to which MQMONNTP connects is called LOCAL_QMGR and the

remote queue manager whose messages we want to browse is called REMOTE_QMGR. In MQMONNTP's location settings for REMOTE_QMGR the *Via QM* field will specify LOCAL_QMGR. In the location settings there is also a field called *Server Queue*. The *User Guide* defines this as, 'The name of the queue at the remote machine where MQMONNTP command messages should be sent'. Let's call this queue MQMON.SERVER. This queue must be defined on REMOTE_QMGR with triggering on and a process that will initiate program CmdSvr.

The MQMD *ReplyToQ* field set by MQMONNTP when messages are sent to MQMON.SERVER contains the queue name specified in the location settings field *Reply Queue* for LOCAL_QMGR. One would expect CmdSvr to send reply messages to this queue. However, a problem arises when the code page or integer encoding of the two queue managers differs and WebSphere MQ attempts to convert the messages (either on the channel if conversion is done there or when MQMONNTP does an MQGET with the MQGMO_CONVERT option). Conversion fails because the reply messages expected by MQMONNTP are not standard PCF messages. To overcome this problem CmdSvr sends standard PCF messages to another queue on LOCAL_QMGR – we'll call it MQMON.SERVER2. This queue triggers program CmdSvr2, which gets the messages with the MQGMO_CONVERT option to ensure that they are converted, changes them to the format expected by MQMONNTP, and puts them into MQMONNTP's reply queue.

Note that in order for CmdSvr2 to deliver the reply messages to MQMONNTP in the correct format, it should run on a Windows platform – normally the same machine that runs MQMONNTP. MQMONNTP connecting to a local queue manager triggering CmdSvr2 is not a problem. However, if MQMONNTP is running as a client, it is recommended you use SupportPac MA7K (WebSphere MQ for Windows 2000 – Trigger Monitor Service) to trigger CmdSvr2. The Client Trigger Monitor Service may be downloaded from http://www-306.ibm.com/software/integration/support/supportpacs/individual/ma7k.html.

If you do not wish to run CmdSvr2 on a Windows platform most functions will still work correctly in spite of the incorrect format of some fields. For example, if REMOTE_QMGR runs on OS/390, LOCAL_QMGR on AIX, and MQMONNTP connects as a client to the AIX queue manager, you will still be able to browse, copy, move, and delete messages. However, selecting a message by MessageId will fail with 'MessageId mismatch error'. To overcome this restriction, messages may be selected by position. For more information on message selection in MQMONNTP see the *User Guide*.

OTHER PARAMETERS

The file CmdSvr.TST contains sample definitions for queues MQMON.SERVER and MQMON.SERVER2 and their associated processes. Please note the following:

- A trigger monitor needs to be running against the initiation queues for the processes to be triggered.
- The TRIGDATA field in the definition of the queue MQMON.SERVER contains the name of the reply queue for CmdSvr.
- The ENVRDATA field in the definition of both processes contains the name of a directory to which the programs will write a log file with error messages.
- If CmdSvr or CmdSvr2 fails to open the log file, an error message will be written with a printf statement. This may be redirected to a file with >> in the APPLICID field of the process definitions.
- On OS/390, all error messages are written with printf. Because CmdSvr is written to run under CICS, these messages will be written to transient data queue CESO.
- On OS/390 the ENVRDATA field of the process definition may contain the name of a CICS terminal id. If so, CmdSvr will be restarted on that terminal. This is to facilitate tracing program execution with CEDF or another debugging tool.

- Parameters to control the execution of CmdSvr and CmdSvr2 are specified in the process USERDATA field and separated by commas:
  - WaitInterval=nnnnnn milliseconds – if this parameter is specified the program will issue its MQGET calls with MQGMO_WAIT and the specified WaitInterval. This can result in improved performance by eliminating reconnecting to WebSphere MQ and re-opening queues between successive commands from MQMONNTP.
  - CodedCharSetId=nnn – this parameter specifies the codepage of the ultimate destination of the reply messages. If MQMONNTP connects to LOCAL_QMGR as a client, this is the codepage of the client machine. If MQMONNTP runs on a Windows server and LOCAL_QMGR is also running on that server, this is the codepage of LOCAL_QMGR. This parameter must be specified for CmdSvr if the codepages of LOCAL_QMGR and REMOTE_QMGR differ and conversion is performed on the channels between the queue managers. If not specified, the default for this parameter is MQCCSI_DEFAULT, which causes the codepage of the current queue manager to be used.
  - Encoding=Normal – this parameter applies only to CmdSvr2. It should be specified if the integer encoding of the platform on which CmdSvr2 runs is not reversed. For example, if CmdSvr2 runs on a Unix platform and MQMONNTP connects to the Unix queue manager from a Windows platform, then CmdSvr2 should convert the reply messages to reversed integer encoding before they are retrieved by MQMONNTP.

Compiling CmdSvr and CmdSvr2

The header file CmdSvr.h contains various constants taken from MQMONNTP as well as a few extras used only by CmdSvr and CmdSvr2.

Platforms on which these programs have been compiled and tested are:

- Windows NT and Windows 2000:
  - Microsoft Visual C++ 5.0 was used.
  - project created as a Win32 console application.
  - CmdSvr linked with library module mqm.lib.
  - CmdSvr2 linked with mqm.lib or mqic32.lib for server or client versions respectively.
- OS/390 and CICS 4.1.0:
  - compiler version OS/390 C V2 R9 M0.
  - compiler option DEF(_MVS_) was specified.
- HP-UX V10.20:
  - compiled with command:

```
cc +DAportable +u1 -Ae +z CmdSvr.c -D_HPUX_ -lmqm -o CmdSvr
```

- AIX 4.3.3 and 5.1:
  - compiled with command:

```
make -f CmdSvr2.aix.mak
```

The files included in the ZIP file available from www.xephon.com/extras/cmdsrv.zip are:

- CmdSvr.aix.mak – makefile for compiling CmdSvr on AIX.
- CmdSvr.c – the C source code for CmdSvr.
- CmdSvr.h – header file for CmdSvr and CmdSvr2.
- CmdSvr.hpux.mak – command for compiling CmdSvr on HP-UX.
- CmdSvr.TST – sample queue and process definitions for REMOTE_QMGR.
- CmdSvr2.aix.mak – makefile for compiling CmdSvr2 on AIX.

- CmdSvr2.c – the C source code for CmdSvr2.
- CmdSvr2.TST – sample queue and process definitions for LOCAL_QMGR.

CmdSvr and CmdSvr2 have been tested with MQMONNTP versions 5.2, 5.2.2, 5.3, and 5.3.2 and may not be compatible with older releases of MQMONNTP.

Eric Judd
Systems Engineer
T-Systems (South Africa)

July 2000 – June 2004 index

Below is an index of all topics covered in *MQ Update* since Issue 13, July 2000. References show the issue number followed by the page number(s). Subscribers can download copies of all issues in Acrobat PDF format from Xephon's Web site.

If you have ever experienced any difficulties with MQ, or made an interesting discovery, you could receive a cash payment or a free subscription to any of our *Updates*, simply by telling us all about it. Articles can be of any length and can be e-mailed to Trevor Eddolls at trevore@xephon.com. A free copy of our *Notes for Contributors* is available from our Web site at www.xephon.com/nfc.

Sonic Software has released Version 6.0 of SonicMQ, its enterprise messaging system using Sonic Continuous Availability Architecture (CAA).

SonicMQ 6.0 reduces the time required for an enterprise messaging system to resume operations after hardware, software, or network failures, eliminating transaction rollbacks.

SonicMQ provides fault-tolerant messaging architecture, reducing failover time. Most fault tolerant architectures rely entirely on custom coded solutions and hardware-based mechanisms to ensure failover. SonicMQ 6.0 provides a software-based solution that uses stateful replication between a pair of servers, eliminating the need for dedicated hardware, specialized fault sensing software, and mirrored/redundant disks.

SonicMQ 6.0 also eases administrative load, resulting in a simpler and more cost-effective approach to enterprise messaging availability, the company claims.

For further information contact:
Sonic Software, 14 Oak Park, Bedford, MA 01730, USA.
Tel: (781) 999 7000.
URL: http://www.sonicsoftware.com/products/sonicmq.

* * *

IBM has announced Version 5.1.1 of WebSphere Studio Enterprise Developer. This new version is focused on supporting multi-platform mixed workload development, integration, debugging, and collaboration. It helps reduce the skills needed to develop component-based Web applications, lets developers integrate WebSphere with traditional transactional environments, promotes the reuse and transformation of existing applications to reduce costs and shorten the development cycle, and provides the engine for integrating inbound and outbound Web Services in z/OS environments.

For further information contact your local IBM representative.
URL: http://www-306.ibm.com/software/awdtools/studioenterprisedev.

* * *

Versata has announced Version 5.6.2 of its Logic Suite. The solution adds support for the IBM WebSphere Application Server Version 5.1 and provides a range of enhanced features to increase developer productivity, Versata claims.

The new Versata Logic Suite, which consists of the Versata Logic Server and the Versata Logic Studio, will also be released by IBM and distributed through the company's Passport Advantage Program.

In addition to support for the IBM WebSphere Application Server Version 5.1, the solution provides improved XML integration and J2EE packaging.

For further information contact:
Versata, 300 Lakeside Drive, Suite 1500, Kaiser Building, Oakland, CA 94612, USA.
Tel: (510) 238 4100.
URL: http://www.versata.com/products/inSuite/products.html.