40

MQ update

October 2002

In this issue

MQ Update

Published by

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

North American office

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

Disclaimer

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

Editor

Madeleine Hudson
E-mail: MadeleineH@xephon.com

Subscriptions and back-issues

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

Contributions

When Xephon is given copyright, articles published in *MQ Update* are paid for at the rate of £170 ($260) per 1000 words and £100 ($160) per 100 lines of code for the first 200 lines of original material. The remaining code is paid for at the rate of £50 ($80) per 100 lines. In addition, there is a flat fee of £30 ($50) per article. For more information about contributing an article you can 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.

MQSeries Publish/Subscribe Service Pack MA0C

In typical publish/subscribe implementations there is often a need for persistent queues to store publications if the subscribing applications are not alive. Organizations where applications are subscribers may typically have not more than 30 subscribers and 100 events. In these cases, although software jurisprudence may direct you towards an established product such as Tibco or MQSI, the MQSeries Publish/Subscribe service pack provides a no-licence-cost, sturdy publish/subscribe mechanism. It's also a good springboard to MQSI.

The MQSeries Publish/Subscribe service pack supports both events and states. Events are independent of each other while state publications contain information, which is regularly updated. If missed, event publications cannot be retrieved, while state publications – even if missed – will be updated in the next publication. State publications are implemented typically through retained publications stored by the broker. The service pack also supports broker-to-broker networks, where publications have to be sent across queue managers on the same or multiple hosts.

For a publish/subscribe mechanism where the subscribers are users this service pack may not provide the performance required. When users are subscribers the number of subscriptions is typically in the thousands. But if the number of user subscribers is less than 100 it might make good business sense to use service pack MA0C: Figure 1 shows a typical implementation.

THE BROKER

The broker is the engine for any publish/subscribe architecture and it is installed with the service pack. It has to be started separately when the queue manager is started. Each queue manager will have a corresponding broker. Publishing and subscribing applications send messages to the broker in MQRFH format, which contains a NameValueString. Using the NameValueString applications can specify the command they want the broker to perform.

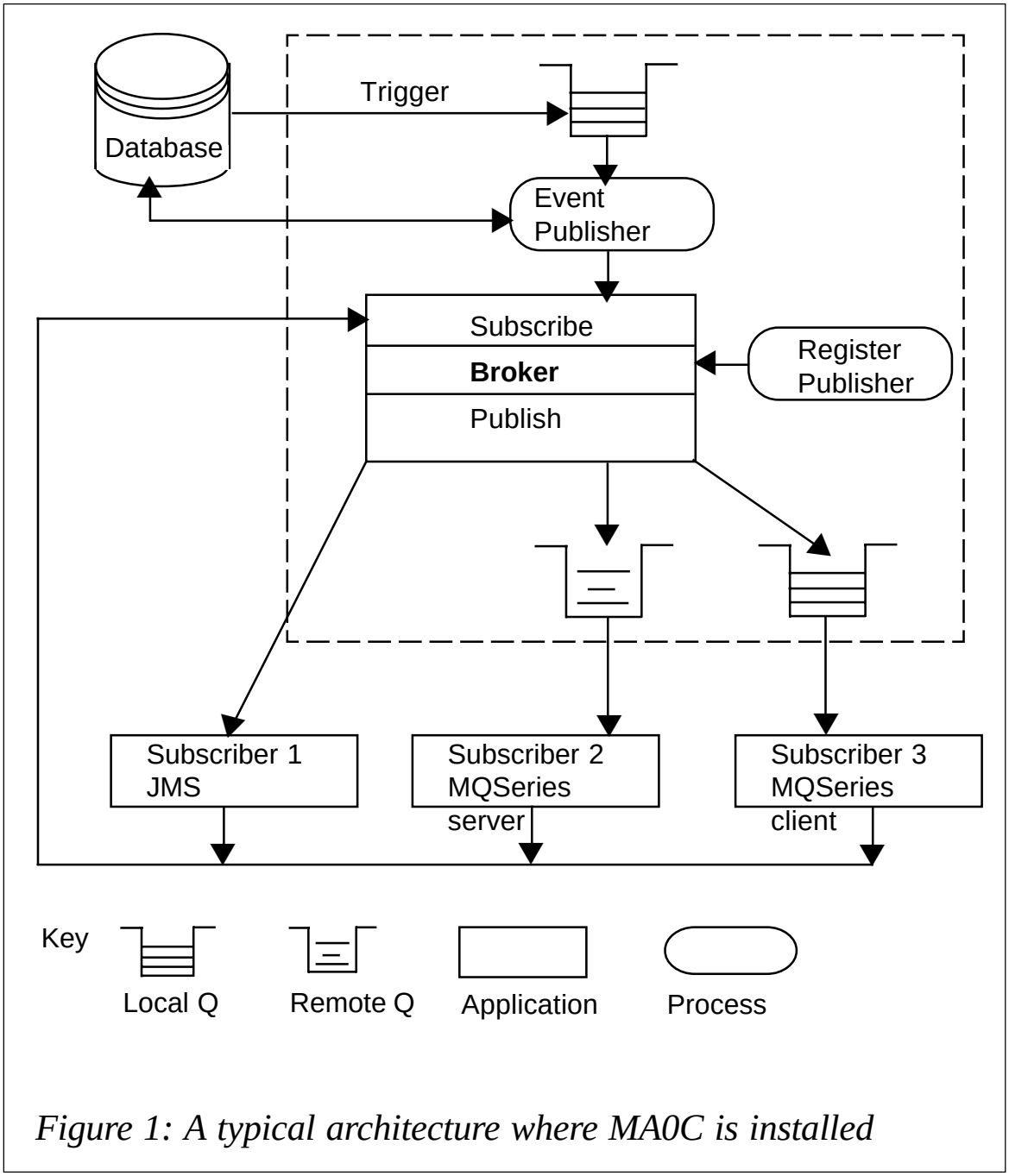


Figure 1: A typical architecture where MA0C is installed

TOPICS

Topics are events that the publisher will publish and subscribers subscribe to. They are typically of the form XXX/YYYY/ZZZZ. For example, a stock ticker subscription to Microsoft stock might look like "STOCK/MSFT". You can use wild card characters to receive one or more events. So, using the previous example, "STOCK/*" will give all stock events while "STOCK/M*" will give all events where the stock name starts with ‘M’.

PUBLISHER

The EventPublisher will feed the stream queues that are part of the broker infrastructure in pseudo real-time. The EventPublisher will poll the queue to which triggered messages are sent. It will parse the data and identify the topic and put the message on the appropriate stream.

The RegisterPublisher will allow administrators to register a publisher before it starts publishing events. This allows the broker to recognize a stream and pick events from it to be published. You may choose to send all published events to one stream queue.

RegisterPublisher

A stream queue to which messages are published has to be registered with the broker. The RegisterPublisher process will send a Register Publisher command message to the broker's control queue to indicate that a process has been defined, which is capable of publishing data on one or more specified topics. The RegisterPublisher will use the following NameValueStrings for the commands:

- MQPSCommand: RegPub.
- MQPSRegOpts: Local.
- MQPSStreamName: <Passed as parameter to the process>.
- MQPSTopic: <Passed as parameter to the process>.

This process will be used by an administrator to register publishers as required.

EventPublisher

The EventPublisher will put messages into the stream defined above for a particular topic. For each event identified by the trigger the EventPublisher will collect the required data and identify the topic name.

To get replies back from the broker all stream queues and queues should be defined by an administrator before the EventPublisher starts publishing messages.

This process will build the MQRFH header, the NameValueString

defined below, and the related data as part of the message. It will put these messages in the appropriate stream, depending on the topic name. The stream to which the message is being put should have been registered with the broker, using the RegisterPublisher process.

- MQPSCommand: Publish.
- MQPSTopic: <Set by EventPublisher>.
- MQPSSeqNum: <Set by EventPublisher>.
- MQPSTimeStamp: <Set by EventPublisher>.

The queue name to get replies from the broker or to get update requests from subscribers directly will have to be defined. This should be set as the ReplyToQ in the MQMD. The ReplyToQMgr in the MQMD should be set to the same as the one on which the broker resides.

SUBSCRIBER

There are three ways of subscribing to published information:

- MQSeries subscribers can subscribe dynamically, by sending a subscription message to the control queue on the ODS host queue manager.
- Subscribers who have MQSeries server software installed can send a subscribe message through a remote definition of the control queue.
- Subscribers who have MQSeries client software installed can send a subscribe message through a channel to the host queue manager control queue.

Subscribers build a NameValueString as below, and send it to the control queue of the broker to subscribe to topic(s).

- MQPSCommand: RegSub.
- MQPSTopic: <Provided by Subscriber>.
- MQPSStreamName: <Provided by Subscriber>.
- MQPSRegOpts: InclStreamName.

MQSeries Server Subscriber

The broker will publish all events to which the subscriber has subscribed into a stream queue (specified during subscription). This queue will be defined as a remote queue to a local queue on the MQSeries Server at the remote host. The subscriber should send subscription messages to a remote queue defined for the Host Broker Control queue. Subscription messages should contain one of the topic names mentioned above and the stream name.

MQSeries Client Subscriber

MQSeries client does not support queue managers. The client software only allows applications to access queues defined as part of an MQSeries server on a different host. The ODS MQSeries server will provide a local queue as a resource for applications which will use the MQSeries client software. It can pick publications from this local queue if the channel is running.

The subscriber application can access the Host Broker Control queue if the channel is running, to send subscription messages. Subscription messages should contain one of the topic names mentioned above and the stream name.

J2EE applications

J2EE applications residing on application servers such as WebSphere or WebLogic can use JMS to subscribe directly to the broker. These applications can have local stream queues defined on the same host as the broker; the Java application can pick up messages from these local queues or the J2EE application can subscribe on-the-fly and receive events that are of interest.

INTEGRATION WITH OTHER MIDDLEWARE

MQSeries can be easily integrated with other middleware, such as Tuxedo and Tibco. In these situations you will have application-specific code, including MQI and API calls. During compilation/build of the code you may have to set some specific options to compile your code.

PERFORMANCE

The performance of the broker is proportionate to the number of subscribers in a single-broker network. The publish/subscribe architecture should be able to handle up to 128 subscribers for multiple topics and still provide reasonable response times.

These performance rates are projected from the *MQSeries – Publish/ Subscribe Performance report* (see below).

REFERENCES

- Service pack MA0D: a quick-start guide to the MQSeries Publish/ Subscribe service pack. *(http://www-4.ibm.com/software/ts/ mqseries/txppacs/ma0d.html.)*
- Service pack MA0C: software and user guide. *(http://www-4.ibm.com/software/ts/mqseries/txppacs/ma0c.html*.)
- Performance metrics of service pack MA0C. *(http://www-4.ibm.com/software/ts/mqseries/txppacs/supportpacs/mp03.pdf*.)

Ramnath Cidambi, Engineer
United Airlines (USA)

A server connection channel security exit

Channel security exits normally come in pairs – one at each end of the channel. However, in our organization we have a security exit that runs only on the server connection channel (not on the client end). Two problems in our setup resulted in the development of this exit: an authentication problem and a concurrent connection problem.

THE AUTHENTICATION PROBLEM

We have several hundred Windows MQSeries clients connecting to one of two Windows NT queue managers, which in turn communicate with queue managers on other platforms. User authentication is carried out by an NT domain controller with a trust relationship with the domain containing the MQSeries servers. The authentication service security

policy on MQSeries is set to *NTSIDsRequired* to ensure that the user-ID being sent to the MQSeries server is the same user-ID that has been validated by the domain controller.

However, when the MQSeries client code was at V5.0 authentication did not happen as we hoped. We found that no user-ID was being sent in the MQCD structure with the connect request. As a result the connected client assumed the user-ID and authority of the MQSeries listener on the queue manager. Being part of the mqm group this user-ID effectively gave them universal access.

We found that upgrading the clients to V5.1 resulted in the logged-on user-ID being sent to the queue manager and the OAM did the necessary verification as expected. However, if no user was logged on to the client machine a blank user-ID was still sent to the server. Also, how were we to be sure that all clients were using MQSeries 5.1 (or later) code?

These issues were solved with the first version of this security exit, which simply checks the user-ID length field in the MQCD structure. If the length is zero the connect request is rejected by returning MQXCC_SUPPRESS_FUNCTION in the ExitResponse field. An error message is also written to a log file stating why the connection was rejected.

THE CONCURRENT CONNECTION PROBLEM

Some client applications were failing to issue an MQDISC call before terminating. When this happens the IP connection is not terminated and as far as the queue manager is concerned the channel is still active. If this happens repeatedly, the number of active channels may soon reach the MaxActiveChannels setting. Increasing this parameter is not the answer because resources on the queue manager are being held unnecessarily. We tried to find a way of killing a particular instance of a server connection channel but only managed to stop all channel occurrences with the same name.

In order to track down the offending applications we included some code in the channel security exit to run on a connect request and check the number of concurrent connections already active for the incoming IP address. If this exceeds a specified limit the MQCONN request is rejected and a message is written to the log file.

REJECTED CONNECTIONS

When the channel security exit rejects a connect request for one of the above reasons, MQSeries writes an error message to *AMQERR01.LOG* "AMQ9536 Channel ended by an exit". The log messages generated by the channel security exit have a similar format to the explanation of the AMQ9536 message as well as additional information explaining the reason for rejection.

CONCLUSION

It is unlikely that your own requirements will dictate an identical channel security exit to ours, but perhaps some of the ideas can be used. In particular, the function ConnectionCount shows how to use PCF messages to invoke the command server – in this case to inquire on the status of channels. It is interesting to note that most MQSeries calls are supported in channel exits. Comments at the beginning of the Channel Security Exit program, which can be found at www.xephon.com/extras/channelsecurity.txt, explain what it does and how to specify parameters for the exit program. For instructions on how to compile channel exit programs see the *MQSeries Intercommunication Manual*.

Eric Judd, Technical Consultant
T-Systems (South Africa)

Very large queues with WebSphere MQSeries V5.3

It is often said that the best queue is an empty queue, but for those times when a queue isn't empty the following information might prove useful.

The total amount of data that can be stored on a queue has notionally always been controlled by two attributes – the maximum size of a message (MAXMSGL) and the maximum depth of a queue (MAXDEPTH). In practice, however, another limit was normally reached before either of these formal values. This article shows how the latest version of MQ for the distributed platforms (Windows NT, Unix, Linux, OS/400) has removed that limit and dramatically extended the storage available for messages.

The key point to understand is that files on disk in the local filesystem are used to represent each queue defined to MQ. All queues, including alias and remote definitions, are stored this way. You can see them in subdirectories under the */var/mqm/qmgrs/<qmgr>/queues* tree on Unix and in similar places on other operating systems. We call these the qfiles and there is always a single qfile for each defined queue.

All qfiles contain information about an object's attributes, which is why alias and remote queues require files, and those qfiles that represent local queues (of which transmission queues are a subset) also hold message data.

One hidden attribute of a queue is a 32-bit number that says how large the qfile is allowed to get. This is the MaxQFileSize attribute. It cannot be viewed as an MQSC attribute but it is automatically set and read by the queue manager, exactly like external attributes such as the creation time of the object. Whenever a message is put to a queue the queue manager will check that the data does not cause this attribute to be exceeded; if it does, it will reject the MQPUT.

The earliest version of MQSeries (V2.*x*) for the distributed platforms had a limit of 320MB for message data. The reasons for this number are lost in the mists of time but, essentially, it was derived from the size of various internal data structures and how many of them could fit into a particular type of OS/400 file. With a maximum file size of 320MB to hold all persistent and non-persistent messages, including their MQMD, a limit of 640,000 messages per queue was chosen. This was a suitable limit, matching the available space and based on a minimum message size of approximately 512 bytes, but should be compared to the definable maximum depth of 999,999,999 messages on a queue held on MQSeries for OS/390.

The OS/390 depth can never actually be reached as queues there also have a physical architectural limit – in this case 4GB – but that was chosen as the MQSC attribute as it is the same as many of the maximum values for other integer attributes.

Very quickly, a number of customers found the 320MB size restrictive and so SupportPac MP01 was released, which documented how to tune the queue managers so that newly created queues could reach 1GB. This was the real limit that the queue manager could cope with at the time;

to make the qfiles larger required internal code changes.

When MQSeries V5.0 was released the default maximum size of a queue was changed to 2GB. This was also the maximum size of a file on most operating systems of the time and was a reasonably sensible choice. However, when existing queue managers were upgraded by running the newer version of the product, the MaxQFileSize for those existing queues was not changed. Only new queues got the larger space for data. There were some internal algorithm changes to manage the qfiles better, but nothing too radical. The advice in MP01 about tuning the qfile sizes now became irrelevant, although other parts of that SupportPac are still useful.

Over the last few years changes to other parts of MQ have made both the 640,000 and 2GB limits more of a concern to customers. The introduction of MQ clustering, with its use of a single transmission queue and the enormous performance boosts started in V5.2 and extended in V5.3, have combined to mean that it is now possible to fill a 2GB queue in well under a minute.

A network failure can mean that applications start to get bad return codes as a transmission queue fills up before any monitoring tool has had much of a chance to react to the failure.

With V5.3 these concerns have now been addressed. The only directly visible change is that queues can now be set to have the same MAXDEPTH value as MQSeries for OS/390. Internally there have been major changes to the storage of queues and messages. There is still a one-to-one mapping of a logical queue to the physical qfile. However, the availability of filesystems that support larger files is now near-universal and the 2GB limit can be discarded while maintaining the single qfile design. The queue manager has been changed to handle efficiently qfiles that get to nearly 2TB in size – that's about 1,000 times larger than previously.

Many internal algorithms have been enhanced to make sure that messages can still be found quickly and that the disk space is reclaimed in a timely way when it is no longer needed, but none of these changes can be seen directly.

No migration is needed for queue manager data as far as MQ is concerned. Unlike the change from V2 to V5 you do not need to

redefine queues to reset the MaxQFileSize. As this was never a public attribute of a queue, V5.3 simply re-interprets the existing 32-bit value stored in the qfile by multiplying it by 1024! This immediately gets the larger qfiles available for all queues. However, a migration step might still be needed for your filesystems before the larger size can be used.

On some operating systems, such as Windows 2000, large files (ie files bigger than 2GB) are always available in the filesystem. On most of the Unix platforms, however, large files are only available as a non-default option on the filesystems and filesystems cannot be modified after their creation to permit large files.

If you have used the normal design for Unix filesystems you will have a */var/mqm* and a */var/mqm/log* filesystem and it is likely that you have not created these filesystems with the large file option. To enable large files you will have to take action along the following lines:

- Stop all queue managers and listeners.
- Unmount the */var/mqm/log* filesystem.
- Create a new filesystem the same size as */var/mqm* but using the options that allow large files.
- On AIX use the **crfs** command with the option "-a bf=true". (Other operating systems have similar commands and options.)
- Mount this new filesystem in a temporary place, such as */tmp/mqm*.
- Copy all data from */var/mqm* to */tmp/mqm*, being careful to preserve permissions.
- Unmount */tmp/mqm* and */var/mqm*.
- Remount the new filesystem in its correct, final position of */var/mqm*.
- Remount */var/mqm/log*.
- Use the **ipcrm** command to remove all IPC elements owned by the mqm user as they may be anchored from the original */var/mqm* filesystem.
- Start the queue managers.

- Delete the original */var/mqm* filesystem (delay this step until you're happy that the queue managers are running successfully).

Note that this procedure does not modify the log directories. Although the qfiles have changed, the log files still do not exceed 64MB each and so do not need any change to the filesystem configuration.

If a filesystem does not support large files the queue manager will get an appropriate return code when trying to extend the qfile. This will be exactly the same as running out of space on the filesystem when a message is put.

It is still recommended that you do not allow queues to get too deep and do not expect magnificent performance if you do lots of msg-ID or correl-ID retrieval on very deep queues. The 999,999,999 depth limit is still an unreachable number. However, for those times when temporary storage of lots of messages becomes unavoidable, this new capability should find a few friends.

Mark E Taylor
MQSeries Technical Strategy
IBM Hursley (UK)

A tale of two queues

There is a philosophical difference between IBM's Message Queueing (MQ) and Oracle's Advanced Queueing System (AQS) that can cause some interesting and challenging headaches when you are handling messages that are being relayed between these two queue types.

MQ assumes that all the message content will be transmitted as a payload and that a single payload or message will have a set of fixed attributes that are provided by the queueing system. These attributes include Enqueue Time, Dequeue Time, Queue Name, Reply To Queue, Message-ID, Correlation-ID, and others.

The AQS system provides similar attributes for a message but views a message as a collection of columns in an Oracle table. In fact at the simplest high level, an AQS message is a row in a table that the user has

defined. Enqueueing consists of inserting a row and dequeueing involves retrieving a row (and deleting it to remove it from the queue).

In truth, IBM does the same thing and a message in MQ is actually a row in a DB2 database. But this fact is well hidden under the MQ application and the API layer. The difference is that MQ predefines the columns in the database and the programmer/user never sees them as columns. Oracle allows the user to define the columns of the 'message' and provides a collection of Oracle database procedures – called a package – for enqueueing and dequeueing.

From a structural perspective an MQ message is a bunch of predefined attribute columns, one of which happens to be a large column/attribute for holding the message payload. An AQS message is a bunch of predefined attributes columns plus a bunch of user-defined attributes. If a user decided to define a single payload column as the meat of a message an AQS message would be almost identical to an MQ message.

Of course it never works out that way. Usually you have to collect a message from an Oracle AQS that has numerous user-defined attributes and drop it off in an MQ queue that has only one truly usable attribute, the payload.

In this real world example for an insurance company I had to collect such a message from an AQS queue. The message had one large payload column but certain key pieces of information were made to stand out by creating separate columns for them. These columns were ClaimNo, PolicyNo, and Application-ID; respectively, the insurance claim number, the policy number, and a unique ID generated by the initiating application. This ID had to travel up the mainframe for logging and be returned by the mainframe so that the response could be tied to the initiating request back at the application. This was similar in concept to the MQ correlation-ID but had been generated by the application.

If it weren't for these three additional attributes the message could have been picked up at AQS, dropped off at MQ, and then moved merrily on its way.

There were various solutions available: keep a temporary table of these three attributes, generate a correlation-ID, send the message off to the MQ with the correlation ID, and then, on the way back, use the returning

correlation-ID to look up the three attributes, put them back into the outbound Oracle message columns and . . . . . oh, yuk.

The best solution was to come up with a way of embedding the extra attributes in the message itself so that when it got placed into the MQ payload the message carried its extra attributes with it, but these attributes could still be identified as separate from the payload itself.

XML is a data format that allows multiple attributes to appear within a set of data.

Some thought is needed to convert a message and any associated attributes into XML but it is surprisingly easier than you would imagine thanks to Open Source XML packages such as Xerces (*http://xml.apache.org/*).

Assume for a moment that you have a message in an Oracle AQS table as described in Listing 1. When retrieving it you need to extract each of the four attributes.

LISTING 1: THE ORIGINAL MESSAGE

```
Message = AA042720020522BAKER              CHARLES etc.
ClaimNo = 00-4004
PolicyNo = XX345678
AppId = APP-1212
```

To pass it on to MQ you need to wrap the message up so that the attributes travel within the body of the message. The scheme I used is shown in Listing 2.

LISTING 2: THE MESSAGE WRAPPED IN XML

```
<The_Wrapped_Message>
  <Message>AA04279920020522BAKER              CHARLES etc</Message>
  <ClaimNo>00-4004</ClaimNo>
  <PolicyNo>XX345678</PolicyNo>
  <AppId>APP-1212</AppId>
</The_Wrapped_Message>
```

All characters from <The_Wrapped_Message> through to </The_Wrapped_Message> are included as the payload in the MQ message.

At the mainframe an XML utility or even a COBOL program using the

UNSTRING verb can be used to break out the pieces, as shown in Listing 3.

LISTING 3: USING UNSTRING TO EXTRACT XML DATA

```
UNSTRING INPUT-XML
    DELIMITED BY "<ClaimNo>" OR "</ClaimNo>"
    INTO JUNK-1,
         CLAIM-NO,
         JUNK-2.
UNSTRING INPUT-XML
    DELIMITED BY "<Message>" OR "</Message>"
    INTO JUNK-1,
         THE-MESSAGE,
         JUNK-2.
```

The turnaround from the mainframe creates an XML return value by reassembling all the tags (Listing 4).

LISTING 4: A COBOL RESPONSE IN XML

```
MOVE "|AA042799APPROVED|" TO THE-MESSAGE
STRING
  "<The_Wrapped_Message>" DELIMITED BY SIZE
  "<Message>" DELIMITED BY SIZE
  THE-MESSAGE DELIMITED BY "|"
  "</The_Message>" DELIMITED BY SIZE
  "<ClaimNo>" DELIMITED BY SIZE
  CLAIM-NO DELIMITED BY SPACE
  "</ClaimNo>" DELIMITED BY SIZE
* remaining fields
  "<.The_Wrapped_Message>" DELIMITED BY SIZE
  INTO OUTPUT-XML.
(* and drop it in the appropriate queue.)
```

The message arriving from MQ when picked up by the middleware looks like that shown in Listing 5.

LISTING 5: THE RETURN MESSAGE WRAPPED IN XML

```
<The_Wrapped_Message>
  <Message>AA042799APPROVED</Message>
  <ClaimNo>00-4004</ClaimNo>
  <PolicyNo>XX345678</PolicyNo>
  <AppId>APP-1212</AppId>
</The_Wrapped_Message>
```

It must be converted to the one shown in Listing 6 to be dropped off at the AQS.

LISTING 6: THE ORIGINAL MESSAGE UNWRAPPED

```
Message = AA042799APPROVED
ClaimNo = 00-4004
PolicyNo = XX345678
AppId = APP-1212
```

The key to this is the middleware piece I had to write that wraps and unwraps the message.

I implemented mine in C++ because of very high volume requirements that caused Java to break down. However, if you do not have this high volume the similarity between C++ and Java would make the conversion simple enough.

The base class for the system is a qMsg class that is derived from a Standard Template Library map class. In the examples below I have also created simple aqs and mq objects for reading from and writing to these two queue types.

A simplified example of how the objects are used in practice is shown in Listing 7. Listing 8 shows the output of AQS2MQ on an 80-character screen.

LISTING 7: SAMPLE CODE USING WRAPPED MESSAGES

```
// AQS2MQ.cpp
// illustration of using wrapped and unwrapped messages
// to accommodate the extra attributes in an AQS message
#include <iostream>
#include <sstream>
using    std::cout;
using    std::endl;
using    std::stringstream;
#include "aqs.h"
#include "mq.h"
#include "qMsg.h"
// format into rows
string qMsg_rows(qMsg& q)
{
  stringstream strWork;
  qMsg::iterator pos;
```

```
  for(pos = q.begin(); pos != q.end(); ++pos)
  {
    strWork << pos->first <<  "=" << pos->second << endl;
  }
  return strWork.str();
}
int main(int argc, char* argv[])
{
  mq mq_q;         // an MQ processing object
  aqs aqs_q;       // an AQS processing object
  qMsg m1,m2;      // the actual messages
  // connecting to aqs and mq goes here
  // (not shown)
  // pick up a message from AQS
  aqs_q.getMsg(m1);
  cout << "msg from aqs >>>\n"<< qMsg_rows(m1)  << endl;
  // wrap the message so that all elements appear in the payload as XML
  m1.addWrapper();
  cout << "after wrap   >>>\n"<< qMsg_rows(m1)  << endl;
  // output to MQ
  mq_q.putMsg(m1);
  // wait for the reply
  mq_q.getMsg(m2);
  cout << "msg from mq  >>>\n"<< qMsg_rows(m2)  << endl;
  // remove the XML wrapper
  m2.removeWrapper();
  cout << "after unwrap >>>\n"<< qMsg_rows(m2)  << endl;
  // put result on the AQS queue
  aqs_q.putMsg(m2);
  // disconnecting from aqs and mq here
  // (not shown)
  return 0;
}
```

LISTING 8: AQS2MQ OUTPUT ON AN 80-CHARACTER SCREEN

```
msg from aqs >>>
AppId=APP-1212
ClaimNo=00-4004
Message=AA04279920020521BARKER      CHARLES
PolicyNo=XX345678
after wrap   >>>
Message=<The_Wrapped_Message><AppId>APP-1212</AppId><ClaimNo>00-4004</
ClaimNo><M
essage>AA04279920020521BARKER       CHARLES</
Message><PolicyNo>XX345678</PolicyN
o></The_Wrapped_Message>
msg from mq  >>>
Message=<The_Wrapped_Message><Message>AA042799APPROVED</
```

```
Message><ClaimNo>00-4004
</ClaimNo><PolicyNo>XX345678</PolicyNo><AppId>APP-1212</AppId></
The_Wrapped_Message>
after unwrap >>>
AppId=APP-1212
ClaimNo=00-4004
Message=AA042799APPROVED
PolicyNo=XX345678
```

In practice the qMsg class would probably be itself a base class from which specific messages derive, but I am trying to illustrate how to make AQS and MQ interoperable so I will keep the derivation chain short.

The qMsg class shown in Listing 9 has two public methods for wrapping and unwrapping, a couple of private assistance methods for placing XML tags around a string, and the private 'isWrapper()' method for determining whether or not a message has arrived with a wrapper.

LISTING 9: QMSG.H

```
#ifndef  _QMSG_H
#define  _QMSG_H
#include    <map>
using   std::map;
#include    <string>
using   std::string;
// These define keys/tags used for the four parts of the message
extern    const string    PAYLOAD_KEY;
extern    const string    CLAIM_NO_KEY;
extern    const string    POLICY_NO_KEY;
extern    const string    APP_ID_KEY;
// The Xerces global XMLPlatform Utilities Class
#include <util/PlatformUtils.hpp>
class qMsg   :   public    map<string,string>
{
public:
    long addWrapper();
    long removeWrapper();
private:
    bool isWrapped();
    string& tagOpen(string& dest,const string& tag);
    string& tagClose(string& dest,const string& tag);
    string& tagWrap(string& dest, const string& tag, const string& val);
};
// This class will be used a sentry to ensure that
// XMLPlatformUtils::Terminate() is called when
```

```
// The method using XMLPlatformUtils() completes
class XMLPlatformSetup
{
public:
    XMLPlatformSetup() {}
    void initialize()    {XMLPlatformUtils::Initialize();}
    ~XMLPlatformSetup() {XMLPlatformUtils::Terminate();}
};
#endif   //   _QMSG_H
```

Starting at the first step of Listing Seven, the call to retrieve the message from AQS uses aqs_q.getMsg(m1). This method is shown in Listing 10.

The elements or keys of a map can be accessed directly, using the [] operator so the qMsg object can be loaded directly. In Listing ten the getMsg() method of the aqs class calls a lower level getAqsMsg() method and passes in receivers for the four pieces of data that are to be retrieved from an AQS message.

Assume that upon return from getAqsMsg(), the sMsg, sClaimNo, sPolicyNo, and sAppId variables have been filled in. Upon return, the four pieces are used to populate the qMsg object.

I am not providing the details of getAqsMsg(). The API to the AQS looks nothing like the MQ API that you might be familiar with and is fairly complex and beyond the scope of this article.

LISTING 10: THE GETMSG() MEMBER OF AQSMSG

```
// collect the four part message and place them in the qMsg map
long aqsMsg::getMsg(qMsg& m)
{
    string sMsg,  sClaimNo, sPolicyNo,  sAppId;
    // call into the AQS to fill in the 4 strings
    getAqsMsg(sMsg,sClaimNo,sPolicyNo, sAppId);
    m.clear();              // clear the map before loading
    m[PAYLOAD_KEY] = sMsg;
    m[CLAIM_NO_KEY] = sClaimNo;
    m[POLICY_NO_KEY] = sPolicyNo;
    m[APP_ID_KEY] = sAppId;
    return 0;
}
```

The next step in the process calls the m1.addWrapper() method shown in Listing 11. This method functions by taking each element of the map,

adding a beginning and ending tag, and then appending that to a string that is being built.

LISTING 11: THE ADDWRAPPER() METHOD

```
// Adding a wrapper can be done in text mode
// Take each attribute of the message and wrap it in a start and end tag
// using the attribute name as the tag name
// then wrap the whole lot in a start and end wrapped message tag.
long qMsg::addWrapper()
{
    string strWrk, strOutput;
    qMsg::iterator pos;
    strWrk = "";
    // for each element in the map
    for(pos = begin(); pos != end();++pos)
    {
        // wrap the value with the element key as a tag
        strWrk += tagWrap(strOutput,(pos->first).c_str(),pos->second);
    }

    strWrk = tagWrap(strOutput,WRAPPED_MESSAGE_KEY,strWrk);
    // clear out the current map
     (*this).clear();
    // and insert the newly built string as the only payload
     (*this)[PAYLOAD_KEY] = strWrk;
    return 0;
}
```

Since map elements are in alphabetical order the first element to be located will be AppId. Using the example previously illustrated this element will have a value of APP 1212. This will be wrapped up as <AppId>APP 1212</AppId>. The next element is ClaimNo which becomes <ClaimNo>00 4004</ClaimNo> and is added to the output string to become <AppId>APP 1212</AppId><ClaimNo>00 4004</ClaimNo> and so on until all elements are wrapped up as shown here in Example 1:

```
<AppId>APP-1212</AppId><ClaimNo>00-4004</ClaimNo><Message>
<AA0427992002052 1BARKER CHARLES</Message><PolicyNo>XX345678</PolicyNo>
```

The final step puts a beginning and ending tag around the whole assembly and then makes this the payload of the qMsg object, as shown here in Example 2:

```
<The_Wrapped_Message><AppId>APP-1212</AppId><ClaimNo>00-4004</ClaimNo>
<Message>AA0427992002052 1BARKER  CHARLES</Message>
```

```
<PolicyNo>XX345678</PolicyNo></The_Wrapped_Message>
```

Listing 12 shows the private tagging methods called by addWrapper().

LISTING 12: METHODS FOR ADDING TAGS TO TEXT

```
// these methods pass string references in
// and out to avoid building temp strings on the return
// create an open tag
string& qMsg::tagOpen(string& strDest,const string& strTag)
{
    strDest = string("<") + string(strTag) + string(">");
    return strDest;
}
// create a close tag
string& qMsg::tagClose(string& strDest,const string& strTag)
{
    strDest = string("</") + string(strTag) + string(">");
    return strDest;
}
// wrap a value with a starting and ending tag
string& qMsg::tagWrap(string& strDest, const string& strTag, const
string& strVal)
{
    string strOutput1,strOutput2;
    strDest = (tagOpen(strOutput1,strTag) + strVal +
tagClose(strOutput2,strTag));
    return strDest;
}
```

We then return to Listing 7, where the wrapped message is sent off to an MQ queue using mq_q.putMsg(m1). That completes one side of the transaction.

Now we await the return from the mainframe. In this instance the mainframe returns XML. There is no restriction on XML requiring children elements to be in alphabetical order so the return message children can be in any order as long as they are within a well-formed XML message, as shown here in Example 3:

```
<The_Wrapped_Message><Message>AA042799APPROVED</Message> <ClaimNo>00-
4004</ClaimNo><PolicyNo>XX345678</PolicyNo> <AppId>APP-1212</AppId></
The_Wrapped_Mesage>
```

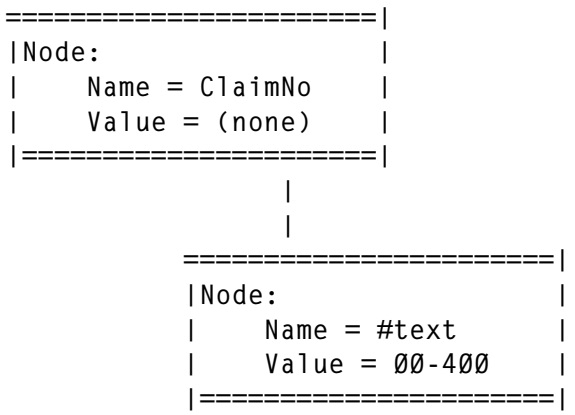
The returned message calls its public removeWrapper() method to unwrap the message into map elements and values.

Before you explore Listing 13 it is important to understand that the

information in a Document Object Model (DOM) tree is arranged slightly differently from the information in an XML document.

In an XML document a tag, a value, and an end tag make up a complete unit, eg <ClaimNo>00-4004</ClaimNo>.

In a document object an additional layer is placed between the tag and the value called #text. The natural arrangement would be a node named ClaimNo with a value of 00-4004 but the DOM arranges these as a node named ClaimNo with no value. This node has a child named '#text' with a value of 00 4004, as shown here in Example 4. (Remember this when reviewing the code.)



Example five, which is shown in Figure 1, is the complete document object created by loading the XML message from Example 3 above.

The removeWrapper() function starts by checking whether the payload portion of the message is wrapped. A wrapped message is one in which the first part of the string matches the wrapper tag <The_Wrapped_Message>. If there is no match the message is not wrapped and is returned without further processing. Listing 13 includes the isWrapped() method.

LISTING 13: REMOVEWRAPPER() AND THE ISWRAPPED() HELPER METHOD

```
// removing a wrapper can be done in text mode,
// but it is more instructive and more efficient to
// do it using an XML parser and a DOM tree
long qMsg::removeWrapper()
{
    // if the message is unwrapped then we can bail out
    if(isWrapped() == false)
        return 0;
```

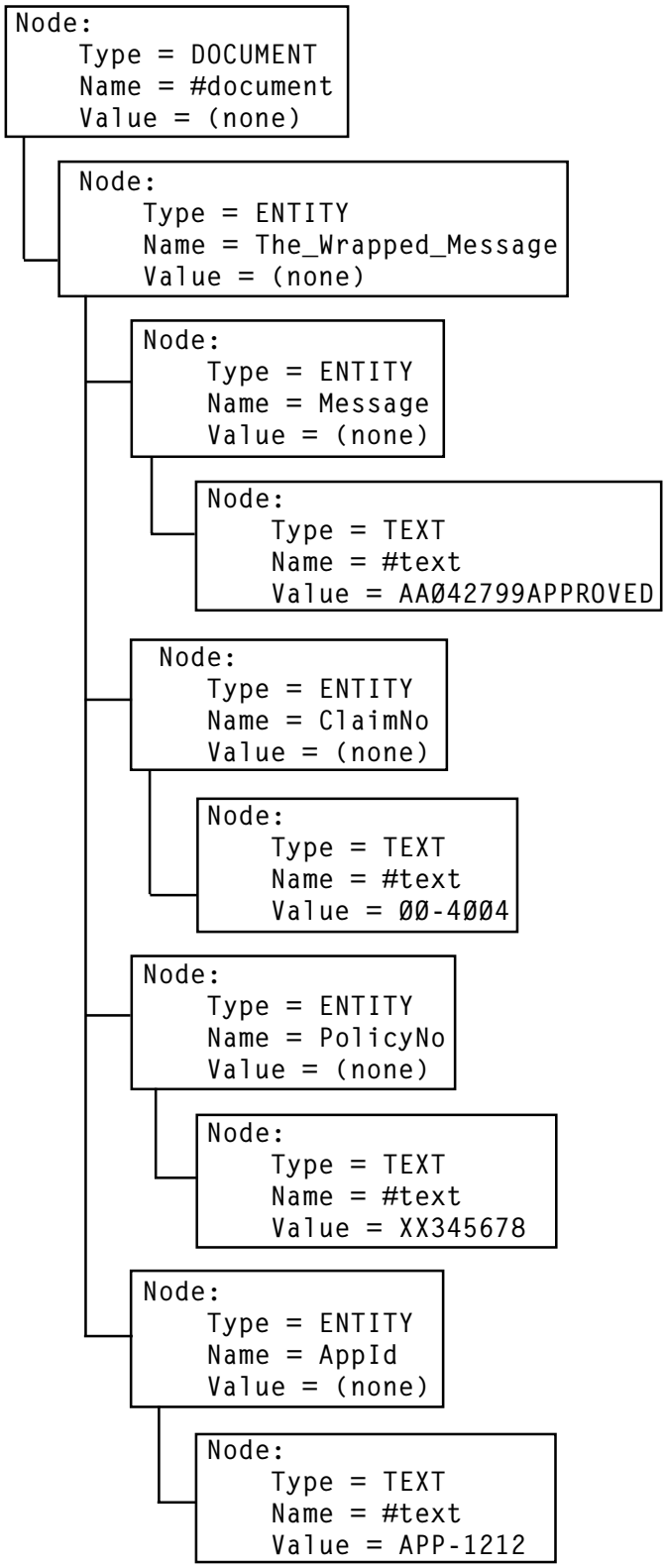


Figure 1: The document object (Example 5) created from Example 3.

```
    // Set up a Sentry so that we
    // can exit from anywhere and the sentry will call
    // XMLPlatformUtils::Terminate() for us.
    XMLPlatformSetup XMLSetup;
    // Initialize the Xerces XML system
    // through the sentry
    // and bailout if that fails
    try
    {
        XMLSetup.initialize();
    }
    catch (const XMLException& toCatch)
    {
         cerr << "Error during initialization!" << endl;
         return -1;
    }
    // the parser needs values during set up
    DOMParser::ValSchemes     valScheme = DOMParser::Val_Never;  // non-
validating
    bool doNamespaces = false;                                   //
without name spaces
    // And an error handler
    DOMError dError;
    DOM_Node wrapperNode;
    DOM_Node pNode;
    DOM_Node childNode;
    string strNodeValue, strNodeName, strChildNodeName;
    // this temporary message is populated while parsing the input
    qMsgwrkMsg;
    string& wrappedMsg = (*this)[PAYLOAD_KEY];
    // Instantiate the DOM parser.
    DOMParser parser;
    // and set it up
    parser.setValidationScheme(valScheme);
    parser.setDoNamespaces(doNamespaces);
    // Install the error handler
    parser.setErrorHandler(&dError);
    // In this named memory buffer, the name is just a convenience
    // that the parser can use when printing error messages.
    // The parse uses a named memory buffer when it is parsing and
    // and XML string in memory.
    char* gMemBufId="NamedBuff";
    // Set up an input buffer source that uses the characters in the
    // wrapped message string
    MemBufInputSource xmlBuf((XMLByte*) wrappedMsg.c_str(),
                                        wrappedMsg.size(),
                                        gMemBufId,false);
    //  Kick off the parse of a wrapped message string
    //  file. Catch any exceptions that might propagate out of it.
    try
    {
        parser.parse(xmlBuf);
```



```
    }
    catch (const XMLException& toCatch)
    {
        DOMString s(toCatch.getMessage());
        cerr << "\nError during parsing: '" << gMemBufId << "'\n"
             << "Exception message is:  \n"
             << s.transcode() << "\n" << endl;
        return -1;
    }
    //  Extract the DOM tree
    DOM_Document doc = parser.getDocument();
    // down one layer to the highest level wrapper which should be
    // the wrapped message tag
    wrapperNode = doc.getFirstChild();
    if(wrapperNode == NULL)
    {
        cerr << "Could not find wrapper key" << endl;
        return 1;
    }
 // extract the node name and translate to ASCII
 // getNodeName() returns a DOMString (Unicode 16 bit wide characters)
 // the DOMString provides a transcode() method to retrieve the value as
    // 8 bit characters which are inserted into a string object
    strNodeName = (wrapperNode.getNodeName()).transcode();
    //error if not the wrapper message
    if (strNodeName != WRAPPED_MESSAGE_KEY)
    {
        cerr << "Could not find wrapper key" << endl;
        return 1;
    }
    // We extract all tagged pieces inside the wrapped message
    // Walk the DOM tree using
    // getFirstChild() returns the first child node of a parent node
    // getSibling() returns the node next to the a child node
    for(pNode = wrapperNode.getFirstChild();
        pNode != NULL;
        pNode = pNode.getNextSibling())
    {
        // get the node name
        strNodeName = (pNode.getNodeName()).transcode();
        // the child of each node here should be a #text node
        // get child node and the child nodes name
        childNode = pNode.getFirstChild();
        strChildNodeName = (childNode.getNodeName()).transcode();
           // for the pay load tag
        if (strChildNodeName != "#text")
        {
            cerr << "Contents of " << strNodeName << " tag are not
text." << endl;
            return 1;
        }
        else
```

```
        {
            // We use the name of the node as the map key and
            // we use the value of the child #text node as the value
            strNodeValue = (childNode.getNodeValue()).transcode();
            wrkMsg[strNodeName] = strNodeValue;
        }    }

    // clear this message and
    (*this).clear();
    // store the resulting work message in me
    (*this) = wrkMsg;
    return 0;
}
// Return true if this is a wrapped message
bool qMsg::isWrapped()
{
    string strPayload;
    string strWrapped;
    // a message is wrapped if the value of the first piece of
    // information in (first part of) the payload string
    // matches the WRAPPED_MESSAGE_KEY
    strPayload = (*this)[PAYLOAD_KEY];
    if(strPayload.size() < WRAPPED_MESSAGE_TAG.size())
        return false;
    strWrapped = strPayload.substr(0,WRAPPED_MESSAGE_TAG.size());
    if(strWrapped == WRAPPED_MESSAGE_TAG)
        return true;
    return false;
}
```

A call to XMLPlatformUtils::Initialize() initializes the Xerces XML processing environment. This is done inside the XMLPlatformSetup sentry that is used to ensure that XMLPlatformUtils::Terminate() is correctly called when removeWrapper exits.

The next steps set up the XML parser. Most of Xerces' tools need to be instantiated with options and the parse is no exception. Here it is set up without validation or name spaces.

The Xerces DOM parser parses from any source that is derived from the XML base class InputSource. The Xerces package provides a number of classes derived from this base. These sources include LocalFileInputSource, MemBufInputSource, StdInInputSource, and URLInputSource. They are each set up differently, but ultimately handed to the parser. It would have been possible to derive a StringInputSource and use it directly, but it was just as easy to specify MemBufInputSource that pointed to the buffer inside a string object, because the parser does not change the buffer.

The input is parsed and errors are trapped and displayed in detail by wrapping the Unicode error message into a DOMString and using the transcode() method to get into a displayable ASCII. The parser error messages in the Exception object are provided in Unicode and the DOMString provides this convenient method for translating.

Once parsing succeeds, DOM_Document doc = parser.getDocument() provides the document object, which is the top of the DOM tree.

A DOM tree is made up of nodes (DOM_Node class). The document object is a specialized example of the node class and has all of the methods of that node class, which include, most importantly:

```
  getNodeName()    Retrieve the node name as a DOMString
  getNodeValue()   Retrieve the node value as a DOMString
  getFirstChild()  Retrieve the first child as a DOM_Node
  getNextSibling() Retrieve the next sibling as a DOM_Node
```

Listing 14 and Example 6, which follows, show the uses of these key functions.

LISTING 14: USING THE FOUR KEY METHODS OF A DOM NODE

```
// Call this function passing the DOM_Document to
// process the entire DOM tree
void treeWalk(DOM_Node n)
{
    static int indents = 0;
    string strIndents(indents,' ');
    DOM_Node sib;
    cout << strIndents << "node=" << (n.getNodeName()).transcode()
         << " value=" << (n.getNodeValue()).transcode() << endl;
    for(sib = n.getFirstChild(); sib != NULL; sib =
sib.getNextSibling())
    {
        indents += 2;
        treeWalk(sib);
        indents -= 2;
    }
}
```

Example Six is the output of processing the document we have been using in this article using the treeWalk function.

```
node=#document value=
  node=The_Wrapped_Message value=
    node=Message value=
      node=#text value=AA042799APPROVED
    node=ClaimNo value=
```

```
    node=#text value=00-4004
  node=PolicyNo value=
    node=#text value=XX345678
  node=AppId value=
    node=#text value=APP-1212
```

Review Figure 1 – the Document Object – and you will see that the DOM_Document is the top node. The first child of this special node must be The_Wrapped_Message node if this process is to work correctly, so the method extracts the child of the document node and verifies that a node exists and that it is named The_Wrapped_Message. If either test fails we abort with an error.

The four elements that we want are all children of The_Wrapped_Message node and, of course, siblings of each other.

The process of walking through these is handled with a for loop of for(pNode = wrapperNode.getFirstChild(); pNode != NULL;pNode = pNode.getNextSibling()) using the getFirstChild() and getNextSibling() methods to walk the tree.

Inside the loop a test is made to ensure that the payload is #text.

POTENTIAL PROBLEMS

In practice there are several potential problem areas and they are not all discussed in this article. I have handled many of them but the key ones to be aware of are detailed below.

If the initial message is itself XML then the value of the <Message> tag will not show up as text. An XML message that has been wrapped using this method is just a larger XML document. The tags and text values in the initial message become DOM nodes within the document and the <Message> node will not have a #text child; consequently there will be no value in that #text node. To extract such values it is necessary to create an XML writer or formatter that takes all nodes below the <Message> node and formats them back into XML. The formatAsXML() method would have to be written using the tools provided in the Xerces package. Listing 15 shows what that might look like.

LISTING 15: HANDLING A <MESSAGE> THAT IS XML RATHER THAN #TEXT

```
for(pNode = wrapperNode.getFirstChild();
```

```
  pNode != NULL;
  pNode = pNode.getNextSibling())
  {
      // get the node name
      strNodeName = (pNode.getNodeName()).transcode();
      // get child node and child node name
      childNode = pNode.getFirstChild();
      strChildNodeName = (childNode.getNodeName()).transcode();
      // for the pay load tag
      if ((strNodeName == PAYLOAD_KEY) && (strChildNodeName != "#text"))
      {
              // we create XML from the child
              // and use it as the payload
              string ss = formatAsXML(childNode);
              wrkMsg[PAYLOAD_KEY] = ss.str();
      }
      else
      {
              // we use the value of the #text node as the value
              strNodeValue = (childNode.getNodeValue()).transcode();
              wrkMsg[strNodeName] = strNodeValue;
      }
}
```

The next big concern is that the mainframe does not usually return XML. The output from the mainframe is more likely to be a flat string of bytes containing all the information that is to be returned to the application. Example 7 shows a flat record returned from a COBOL application.

```
A042799APPROVED00-4004XX345678APP-1212
```

In this example bytes 1 to 15 form the message, 16 to 22 the ClaimNo, 23 to 30 the PolicyNo, and 31 to 38 the AppId.

We have just stepped off the edge here into the subject of data transformation. You can write a message-specific handling programme for this message or apply a more general approach. I ended up writing a generic data translation package that was script-configurable and very fast and that did most of the functions of the MQ Series Integrator. But that is perhaps a subject for another article.

ONE FINAL NOTE ON XML PACKAGES

I have used both the Xerces package for XML/DOM processing and Microsoft's MSXML package. If you are so unlucky as to be forced to use the MSXML package you can use the same approach, but getting

there is much more tedious and error prone. The MSXML package is designed to be used as a COM object. Within the framework of COM it does a pretty good job. When used directly from C/C++ it does not automatically free allocated resources unless you use various mechanisms that slow the package down. So if you want speed you spend a lot of time hunting for memory leaks.

I recommend the Xerces package. It works well in C/C++, Java can be compiled on a Windows environment, and it has a COM wrapper for the Visual Basic and script users. IBM's XML parser XML4C is based on the Xerces package so it should be as easy to use.

Mo Budlong (Mobudlong@aol.com), Middleware Integration Specialist
King Computer Services (USA)

Multiple CKTI trigger monitor transactions in CICS

The IBM-supplied MQSeries CICS adapter and CKTI trigger monitor enables the triggering of CICS transactions when messages arrive on queues. In most cases one CKTI trigger monitor and the proper initqueue are sufficient for a single CICS system for triggering.

However, there are instances – when using shared queues for example – where additional CKTI transactions within the same CICS system are required. Figure 1 illustrates a possible scenario.

MQA and MQB are members of a queue-sharing group running on different MVS images. CICSA and CICSB are connected to MQA, CICSC is connected to MQB. Shared queue SQ1 uses INITQ1 for triggering and is processed by CICSA and CICSC; shared queue SQ2 uses INITQ2 for triggering and is processed by CICSB and CICSC.

The CKTI transactions of CICSA and CICSC have to listen to INITQ1 and CKTI of CICSB and CICSC have to listen to INITQ2, so CICSC needs two CKTI because it has to listen to two different initqueues.

The initqueue has to be of type QLOCAL; therefore, it is not possible to use QALIAS definitions to point to a single initqueue.

The initqueues can also be defined as shared queues but this is irrelevant

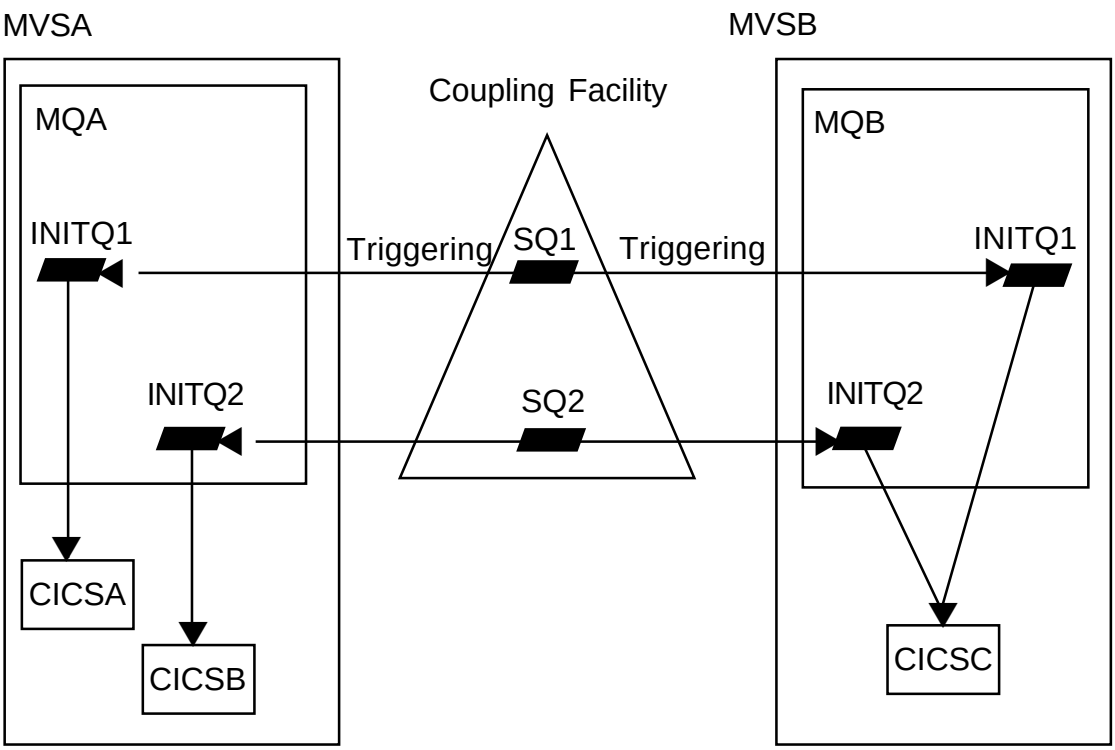


Figure 1: Implementation requiring additional CKTI transactions

and does not remove the requirement for two CKTI transactions in CICSC (check with *Concepts and Planing Guide, Chapter 2, Shared queues and queue-sharing groups*).

How to deal with multiple CKTI transactions in a single CICS system is described in the *System Administration Guide* in the chapter entitled *Operating the CICS Adapter*.

Your reason for multiple CKTI may differ from the shared-queue example. No matter why, if multiple CKTI transactions are required there are two problems that have to be solved:

- The transaction used to start the CKTI trigger monitor transactions has to be a terminal-oriented task.
- There is no automatic CKTI restart.

STARTING THE CKTI TRIGGER MONITOR TRANSACTIONS

The CKTI-starting transaction must run on a terminal (start transaction() userid() is not an option). If done manually, CKTI will be assigned the starter's user-ID, and, therefore, all started application transactions will run with this user-ID. This may result in security problems.

To solve this issue two transactions/programs and a special terminal definition are needed. Transaction one, which is very small, is required to start transaction two on a terminal (console) that has the proper user-ID defined. Transaction two issues the **CKQC STARTCKTI** commands. The started CKTI transactions will then get the terminal user-ID assigned.

Transaction one can be started manually, or the related program can be used in CICS PLTPI to launch the CKTI-starter two at CICS start-up time.

In this example the terminal name is CNCK (typeterm, using device console user-ID CICSDFLT), STC1 transaction (program STCKTI1) starts transaction STC2 (program STCKTI2) on terminal CNCK.

STCKTI2 reads a namelist named sysid.INITQ.NAMELIST and issues the CKQC STARTCKTI command for every initqueue to start the proper CKTI transaction.

NO AUTOMATIC CKTI RESTART

If MQ is stopped and the adapter is in a pending state all CKTI transactions are stopped. If MQ is restarted and the adapter reconnects then only the 'default' CKTI is restarted, which is the CKTI that was started automatically when the adapter became active the first time. The CICS adapter does not restart all other previous running CKTI transactions.

Instead of using automation tools to create a restart procedure it is possible to make MQSeries restart these CKTI transactions by using triggering.

When the CICS Adapter is started and the MQSeries connection is established the 'default' CKTI is started, which opens the initqueue. MQSeries now checks all QLOCAL to see if this queue is used as initqueue and, if it is, whether the trigger conditions match; if they do

a trigger message is created for that specific local queue.

This mechanism can be used to trigger transaction STC1 within CICS. STC1 will start STC2 on terminal CNCK and STC2 will start (or restart) the CKTI transactions.

This solution provides a QLOCAL named sysid.INITQ.CONTROL with the proper trigger attributes and a persistent message to make MQSeries create the trigger at initqueue open time.

CONCLUSION

The advantage of this solution is that, at any time, all CKTI transactions will be restarted automatically when the CICS adapter starts or reconnects to MQSeries.

Besides that, if a manual restart is required it can be achieved by triggering the control queue (eg alter no trigger/alter trigger) that starts transaction STC1, or by starting the transaction STC1 manually.

All CKTI transactions will run with the user-ID defined in terminal CNCK.

To add an initqueue to a running CICS system just add the name in the sysid.INITQ.NAMELIST and trigger the sysid.INITQ.CONTROL queue. This will also try to restart already running CKTI transactions; this will only result in a CSQC383D (another CKTI already running for that INITQ) console message, but no error.

To remove an initqueue just remove the name from the sysid.INITQ.NAMELIST. Wait for CICS restart or stop the proper CKTI manually by using the CKQC transaction.

Some organizations do not like the idea of having these kinds of ‘control-queues’ and ‘control messages’ within MQSeries because MQSeries should be used for transportation only (and not be misused as a database or for operational purposes).

In that case STC1 and STC2 transactions may still be used to start the CKTI transactions but the start of STC1 has to be done in a different way (eg by an automation tool that checks for proper adapter console messages).

SETTING UP

Check object names and adapt to your naming conventions if required. Of course attributes like storageclasses or typeterms may differ in your system. Do not forget to check security definitions.

If your CICS systems are designed to run on different MVS images and if the appropriate MQSeries queue managers are members of a queue sharing group then the MQ objects sysid.INITQ, sysid.INITQ.NAMELIST, sysid.INITQ.CONTROL, and all other initqueues used to trigger applications should be defined with DISP(GROUP) so that they are the same on all queue managers.

MQSERIES DEFINITIONS

- Define QLOCAL(sysid.INITQ), which will be used as the default initqueue when the CICS adapter is started.
- Define QLOCAL(sysid.INITQ.CONTROL), which will be used to trigger transaction STC1 when the CICS adapter is started and CKTI has become active. Use TRIGGER, TRIGTYPE(FIRST), INITQ(sysid.INITQ), PROCESS(STC1), DEFPSIST(YES).
- Put a persistent message to sysid.INITQ.CONTROL. This is required to fulfil all trigger conditions. If the message is not present STC1 will not be triggered. If you use DISP(GROUP) a persistent message has to be put to every instance of the queue.
- Define PROCESS(STC1), APPLTYPE(CICS), and APPLICID(STC1).
- Define NAMELIST(sysid.INITQS.NAMELIST), holding the names of the initqueues used to trigger applications, eg NAMES('APP01.INITQ''APP02.INITQ'). Of course these have to be defined too, perhaps as group entries.
- Check that the triggered application queues do not refer to sysid.INITQ but to one of the application initqueues specified in the namelist. Otherwise you may run into security problems when restarting the default CKTI manually.

CICS DEFINITIONS

- Compile and link STCKTI1 and STCKTI2 programs and place the load modules into CICS DFHRPL.
- Define transaction STC1 (program STCKTI1) and STC2 (program STCKTI2) to CICS.
- Define terminal CNCK to CICS using a TYPETERM that has device CONSOLE defined, NETNAME(CNCK), USERID(CICSDFLT), or use whatever exists or fits into your system.
- Specify sysid.INITQ in the CICS INITPARM for the CICS Adapter and default CKTI trigger monitor.
- Install all definitions into the CICS system (or perform a cold restart).

Your CICS log after restart should show something similar to this:

```
+DFHSI8434I A01CW781 Control returned from PLT programs during the ...
 +DFHSI1517 A01CW781 Control is being given to CICS.
 +START-CKTI PROGRAM STARTED
 + ISSUE CKQC STARTCKTI FOR INITQ APP01.INITQ
 +CSQC386I A01CW781 CSQCSSQ STARTCKTI initiated from TERMID=CNCK
    TRANID=STC2
USERID=CICSDFLT and is accepted
 + ISSUE CKQC STARTCKTI FOR INITQ APP02.INITQ
 +CSQC386I A01CW781 CSQCSSQ STARTCKTI initiated from TERMID=CNCK
    TRANID=STC2
USERID=CICSDFLT and is accepted
 +00000002 CKTI START COMMANDS ISSUED
 +START-CKTI NORMAL COMPLETION
```

Stefan Raabe (stefan.raabe@t-online.de)
Independent Consultant (Germany)

A better MQSeries batch trigger monitor

At my company we have discovered a need to trigger batch jobs from MQSeries queues. Through support pack MA12 IBM supplies what it calls a sample batch trigger monitor. This program runs as a started task and waits for a trigger message on an initiation queue and

then triggers a job pointed to by a DD card in its own JCL. The program as implemented can only start one job for each batch trigger monitor that is running. I found this to be a bit limiting so I set out to design a more versatile replacement. What I came up with works as follows.

My monitor runs as a started task and awaits a trigger message on a single initiation queue, as does IBM's, but I do not rely on a DD card to get the JOB card JCL to submit to the internal reader. Instead, I decided to use the trigger message itself to supply the JCL.

To do this you first define a QLOCAL as trigger with its initiation queue as the queue that the trigger monitor is monitoring. You then define a PROCESS that has the JOB JCL coded in the APPLICID, ENVRDATA, and the USERDATA fields. The APPLICID field allows 256 bytes and the ENVRDATA and USERDATA fields allow for 128 bytes each. I break these fields up into 64-byte records so you can code eight different JCL cards that this program will deliver to the internal reader. That should be enough to start most jobs.

Using this method you can define as many triggered QLOCALs as you have jobs you want to start and point them all to a single initiation queue that this program is watching. With all the JCL provided in the PROCESS definition no DD cards are needed in the trigger monitor started task and no modifications are needed in the program when a new job needs to be started.

PROGRAM LOGIC

When the trigger monitor is started it first reads the Parm input to determine the name of the initiation queue to listen on and the name of the queue manager to connect to. It then connects to the queue manager and opens the initiation queue. While it is doing this it is putting out little informational messages to the SYSPRINT DD.

The program then goes into a GET wait on the initiation queue until it is awakened by a trigger message. When a trigger message arrives the monitor wakes up and parses the message for the JCL it is to submit to the internal reader. After doing this it then goes back into a GET wait on the initiation queue.

Stopping the trigger monitor

To stop this trigger monitor you must put a message on the initiation queue which is of type REPORT and feedback of MQFB-QUIT. The sample program, *CKTIEND*, which is part of the MA12 support pac, can be used to generate this message. I have included the source for this program but you can also download it from the IBM Web site if you find that easier. JCL to run CKTIEND follows the program source.

The batch trigger monitor program source can be found on the Web at www.xephon.com/extras/batchtrigger.txt.

Bruce Borchardt, OS/390 Systems Coordinator
Kohls Department Stores (USA)

WebSphere Financial Network Integrator: technical preview

INTRODUCTION

Early this year IBM announced a preview of WebSphere Financial Network Integrator (WebSphere FNI). This is a product for the financial market and consists of a messaging hub or base, an extension for accessing the SWIFT financial network, and an extension for trusted e-payments. Both extensions are implemented on top of WebSphere MQSeries Integrator (MQI) as message processing middleware. They both share a common customization, configuration, and security model, and therefore use a common set of subflows and plug-ins. These common functions are available separately in the messaging hub – the WebSphere FNI base product. Even though the extensions are positioned in the financial market the common functions are also useful for other exploiters of MQI.

This article concentrates on the WebSphere FNI base product and provides an overview of its concepts and functionality.

TERMINOLOGY

To understand WebSphere FNI you need to know some new terms that

were introduced with the product and the ways in which they relate to MQI and other terminology.

All message processing functions in WebSphere FNI are called services. A service can be implemented as an MQI message flow or subflow or both. A service implemented as a message flow can be accessed by non-MQI programs using WebSphere Message Queueing (MQ) messages. A service implemented as a subflow can be embedded by any WebSphere FNI-enabled message flow just like any MQI delivered primitive.

In addition to message flows WebSphere FNI supports services implemented as normal WebSphere MQ message processing programs, but this will not be described in this article.

Figure 1illustrates the services and server in a WebSphere FNI instance.

Services can be provided either by WebSphere FNI itself or developed with WebSphere FNI functionality. A service does not only require a message processing implementation; usually it also requires a set of resources or resource definitions, eg WebSphere MQ queues or database tables or their equivalent. The naming of such resources must be consistent with their reference in the message flow. Typically, you must transfer resources from a development environment to some test environments and to the production environment. These environments usually have different naming standards or use different resource managers, eg databases.

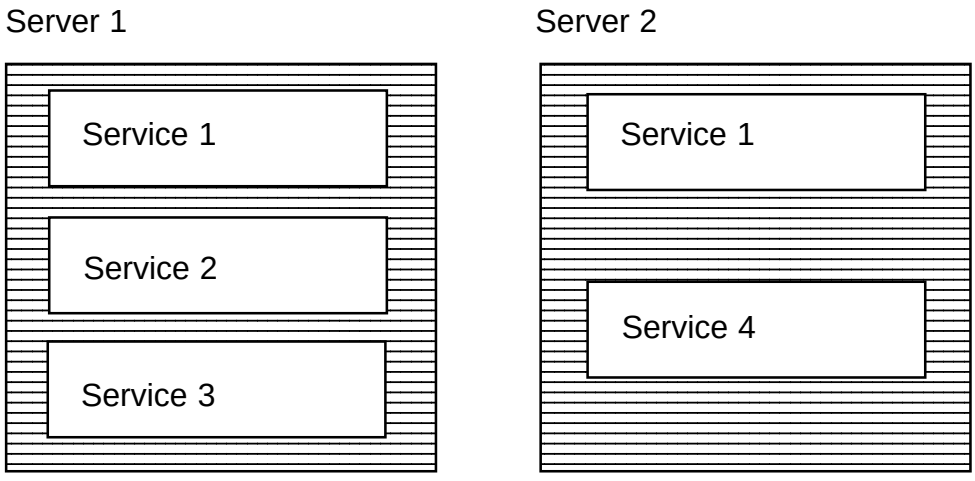


Figure 1: WebSphere FNI instance

Each service runs in a WebSphere FNI server, which is implemented on an MQI broker. As shown with Service 1 in Figure 1, a service can run on multiple servers for availability or throughput reasons. All servers that belong together form a WebSphere FNI instance. This instance is maintained by one MQI broker domain (one MQI Configuration Manager). You can have as many instances as you like. MQI itself provides a graphical tool, the Control Centre, to maintain this set of brokers with its message flows. They can also be administered as a set for the publish/subscribe functionality. In addition, WebSphere FNI provides a component called WebSphere FNI Customization to maintain the message flows and their resources in a consistent manner.

WEBSPHERE FNI CUSTOMIZATION

The WebSphere FNI Customization program allows you to create your instance, to introduce WebSphere FNI server, and to define organizational units (OUs). In addition, it lets you import service bundles. A service bundle is a set of services and their related resources. The customization program does not handle single services. Many services, at least in the WebSphere FNI extensions, share resources with other services. Examples of shared resources include a status table or an error processing queue.

A message flow usually references external resources. Bringing this message flow to a new environment, say from a development environment to a test environment, usually requires adaptations to the resource definitions and the message flow. Resource definitions in development, test, and production environments at most customer sites have different naming conventions, eg different high-level qualifiers. This has to be reflected in the message flow also. Adapting a message flow and the resource definitions to a new environment is very time-consuming and error-prone. This is especially true if you provide your message flow as part of a service offering, solution, or product. In this case you must rely on your customer to perform the adaptations correctly. Maintaining and servicing such adapted message flows and resource definitions can be a nightmare.

To solve the problem WebSphere FNI collects all the necessary information about the instance and its servers during customization. When loading a service bundle to a new server in a new instance the customization program automatically adapts the resource definitions

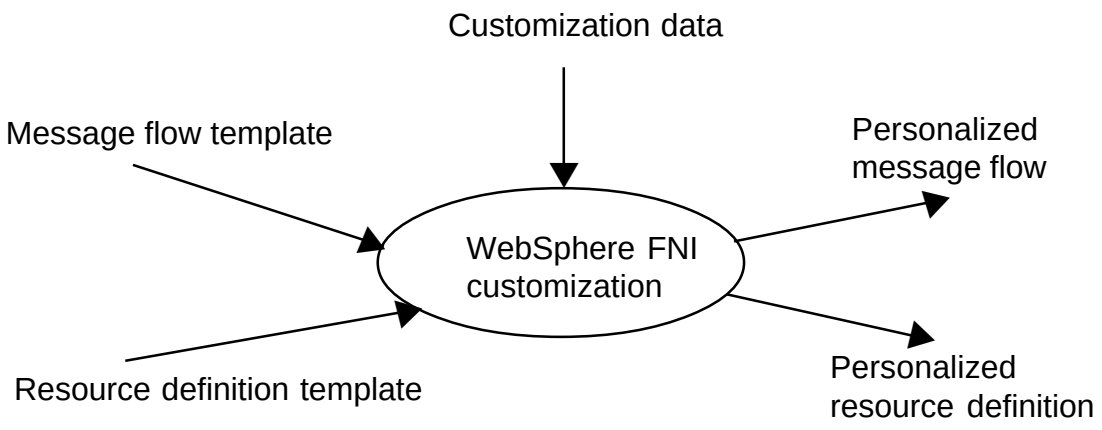


Figure 2: WebSphere FNI customization process

and the message flow according to the collected information. This process is shown in Figure 2, where the imported message flow and resource definition are referenced as templates because they will be enriched with information fed into the customization program as customization data. As a result of the customization process you get personalized message flows and personalized resource definitions, which you can deploy to your resource managers.

The personalized message flow has to be imported into the Control Centre that manages the MQI domain in which the WebSphere FNI instance resides. This message flow can then be assigned and deployed to the broker running the WebSphere FNI server. It is a similar process for personalized resource definitions, eg a WebSphere MQ queue definition has to be processed with the WebSphere MQ program *runmqsc* on distributed platforms for the queue manager that is used for the broker. With this process you get message flows that are consistent with the resource definitions required to run the message flow.

Service bundles are used by WebSphere FNI to adapt WebSphere FNI services to the customer environment. But they can also be developed by anyone else via a simple process.

An intermediate object in the customization is an organization unit (OU). An OU is a logical construct used to restrict access to resources. It can be used to represent a department in an enterprise, an entire company, or any unit that, from a resource access point of view, is to be

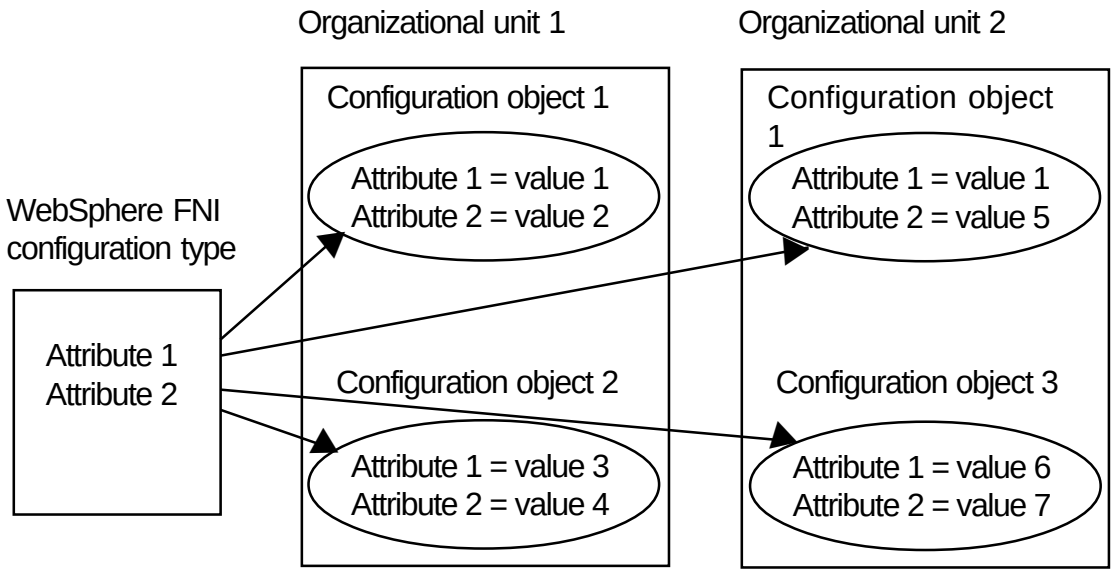


Figure 3: WebSphere FNI configuration types and configuration objects

considered distinct from other units. One WebSphere FNI instance can handle requests for any number of OUs. Several OUs can share the same implementation of a service, ie they can use the same message flow or each can have its own implementations of a service. WebSphere FNI uses a special OU called SYSOU for its administration and configuration services.

WEBSPHERE FNI CONFIGURATION

If a message flow should be shared by different OUs that need different resources, eg OU-specific destination queues or OU-specific databases, static attributes as supported with WebSphere FNI customization would lead to a set of very similar message flows. To avoid this WebSphere FNI offers you dynamic attributes with a WebSphere FNI configuration service.

With the configuration service you can define configuration types and configuration objects, as shown in Figure 3. A configuration type is a definition with a name and a collection of attribute names. The configuration type is defined for the whole instance. For a configuration type you can define configuration objects and assign values to each attribute of the object. This can be done for each OU. These configuration

objects can then be fed into message flow processing and be accessed with the usual MQI compute or filter nodes, for example.

The configuration service is implemented as a message flow that can process a set of commands. To access this message flow WebSphere FNI provides the WebSphere FNI Command Line Interface (CLI). This program formats user input, sends it as a WebSphere FNI request message to the configuration message flow, and displays the results.

With the configuration service you have the option to set up dual authorization. Dual authorization means that a change to configuration definitions done by one administrator becomes active only after it is approved by another administrator. Such approval processes are often required by financial institutions. Dual authorization is enabled by default but can be switched on or off using the configuration service.

WEBSPHERE FNI SECURITY

All services process messages. Each message has a user-ID associated with it. For an application this is the user-ID of the person using the program. In most cases the use of the service is either restricted to some people or a message is only allowed to be processed by a user if certain criteria are met, eg the addressee is allowed for this person. You can protect the whole service by protecting its input queue using external security managers, eg RACF on z/OS. For further security WebSphere FNI provides the WebSphere FNI Security service.

WebSphere FNI provides a service that lets you create and maintain security definitions. Like the configuration service, this service is accessed via the CLI. The WebSphere FNI security model comprises the following:

- *Access rights*, which define the operations that can be performed on configuration types. Customers can define which operations on a specific configuration type can be protected. The access right definitions are carried out for a WebSphere FNI instance.
- *Roles*. A role is a set of access rights that are required to perform a specific task. Roles are independent of OUs and they can be defined by customers. WebSphere FNI already has some predefined roles for configuration and security administration.

- *Users*. Users of WebSphere FNI are those user-IDs that are transported by WebSphere MQ in the message descriptor (MQMD). Users can be assigned to roles in an OU. A user can exercise different roles in an OU or the same role in different OUs.

In a service message flow you can check whether a message is allowed to be processed by the message flow. This check is done against WebSphere FNI security definitions.

As with configuration, dual authorization can also be implemented for security definitions.

WEBSPHERE FNI FUNCTIONS

In addition to the configuration and security services WebSphere FNI provides a set of services that can be used in message flows. These services are implemented as MQI subflows. Examples of these subflows are detailed below.

- *Configuration Provider*. The Configuration Provider subflow retrieves configuration information provided by the WebSphere FNI configuration service. This subflow enriches the message currently being processed with this information. This way, the information is available to subsequent nodes. These nodes then can use the information to make decisions or to use the values as resource names. A decision in a message flow could be whether a specific operation, eg auditing, should be performed for the OU. When using the information as a resource name the values could represent a queue name for an MQOutput node, a database or database table name, or any other resource you need.
- *Access Control*. The Access Control subflow checks whether the user-ID referenced in the MQMD is allowed to perform an operation. This check is done against the definitions made using the WebSphere FNI Security service.
- *Message auditing*. This subflow writes a part of a message to a WebSphere FNI audit database. The audit data is separate for each OU so that an administrator for the OU can view only those audit records written by its own services.
- *Timer*. WebSphere FNI provides a timer service for processing

timer events. This service is implemented in a set of subflows and a message flow. A timer event can be defined, updated, or cancelled, using the appropriate subflow.

A WebSphere FNI timer message flow regularly checks timer events and generates WebSphere FNI service request messages for expired events.

This kind of timer is useful for timer events that are typically longer than several minutes and where the probability that the time-out occurs is low.

- *Warehouse*. The WebSphere FNI warehouse subflow is more sophisticated than the warehouse function provided with MQI. The warehouse entries written using the WebSphere FNI warehouse subflow can be used for customer-defined queries based on fields in the message body.

  Some of the functions provided as subflows are also available as services that can be accessed using WebSphere MQ messages.

  Internally, WebSphere FNI uses additional services that are not disclosed in the first release. One of these services is an event-emitting service. The WebSphere FNI events emitted by this service can be monitored centrally using a monitoring program provided by WebSphere FNI.

WEBSPHERE FNI PROGRAMMING MODEL

Accessing WebSphere FNI services is simple. Services implemented as message flows and that are accessible as WebSphere MQ messages have to use standard WebSphere MQ messages with a WebSphere MQ request and format header version 2 (RFH2). In this header WebSphere FNI requires a specific folder; the use of such folders is a standard technique when creating MQI messages.

Some of the WebSphere FNI subflows also require the information contained in the WebSphere FNI-specific folder of the RFH2. Others do not. Those that do not can be controlled using usual MQI properties if required.

SUMMARY

WebSphere FNI is a general infrastructure based on MQI. It extends this product so that you can base your service offering, solutions, or products on it. WebSphere FNI lets you adapt message flows and resource definitions required by these message flows to a customer environment and provides a configuration and security model that extends MQI functionality.

Michael Groetzner
IBM (Germany)

Natural – MQSeries interface

Information resources are generally managed by key enterprise applications that have been designed and developed using software products from many different vendors, eg IBM and Software AG. In order to integrate these information resources it is necessary to ensure that these heterogeneous applications can interoperate seamlessly across the enterprise.

The code provided (which can be found at www.xephon.com/extras/natural.zip) illustrates how an interface can be built and used to support such seamless interoperation. Specifically, the code illustrates how any application built using Software AG products can interoperate with the rest of the world by using MQSeries as the messaging mechanism.

Developers experienced with either Software AG's Natural or IBM's Websphere MQ – or both – can use this code as a starting point for building a more functionally rich interface between applications built using Software AG's product set and those from other vendors.

The code supplied illustrates an interface that will work on the Microsoft Windows platform. However, IBM provides additional support packs on its Web site (*http://www-3.ibm.com/software/ts/mqseries/txppacs*), detailing interfaces that will work on the mainframe and on Unix platforms as well. The support pack that provides details of the interface for IBM's mainframe platform and Unix platforms is *md07*.

Mohammed Ajab, Martin Howson, Michael Fabianski
IBM (UK)

CommerceQuest has announced Version 7 of its EnableNet Data Integrator, a bulk data/ file movement and integration application that exploits WebSphereMQ. It provides integration with CommerceQuest's integration software, a Web-based interface, and the addition of a scripting integration language.

Product improvements include a process-centric, component-based, service-oriented architecture, common scripting language across operating systems, role-based command and control centre access control, and XML-based callable interfaces. It is WebSphere cluster-enabled and exploits the latest WebSphereMQ capabilities.

Version 7.1, planned for Q4, will feature expanded platform support for the IBM 4690 point-of-sale system and additional Unix platforms.

The software will also allow customizable file-to-message and message-to-file components to facilitate rapid integration with WebSphereMQ-enabled applications, including WebSphereMQ Integrator.

For more information contact:
CommerceQuest, 2202 N Westshore Blvd, Tampa, FL, 33607, USA.
Tel: +1 813 639 6300.
Fax: +1 813 639 6900.
Web: http://www.commercequest.com

CommerceQuest (UK), Doncastle House, Doncastle Road, Bracknell, Berkshire, RG12 8PE, UK.
Tel: +44 1344 861010.
Fax: +44 1344 861011.

* * *

MQSoftware has updated its WebSphere MQ educational offerings to support IBM's newest release of WebSphere MQ.

MQSoftware's WebSphere MQ courses will cover IBM's new features for version 5.3, including IBM's JMS applications, API exits, new product functionality, and added security features.

Announced separately, MQSoftware has signed a distribution agreement to resell Europe-based Primeur's DataSecure product line in the United States.

Primeur's DataSecure is a security solution for WebSphere MQ and the IBM OS/390 environment. It includes an application developer's toolkit for customization, an MQ-based solution for transparent end-to-end security, and a link solution that provides link-oriented security for MQ channels.

DataSecure supports ICSF, HSM and MQ clients and MQ clusters, and offers full PKI compliance, PEA, encryption and authentication, and integrated compression.

For more information contact:
MQSoftware,1660 South Highway 100, Suite 400, Minneapolis, Minnesota 55416, USA.
Tel: +1 952 345 8720.
Fax: +1 952 345 8721.

MQSoftware, Surrey Technology Centre, 40 Occam Road, Surrey Research Park, Guildford, Surrey, GU2 7YG, UK.
Tel: +44 1483 295400.
Fax: +44 1483 573704.

* * *