64

MQ update

October 2004

In this issue

3 SSL in WebSphere MQ 5.3

7 WebSphere Translation Server

10 MQ V5.3 for z/OS page set removal procedure

19 Setting up a WebSphere MQ Integrator Broker in a parallel Sysplex

31 Queue back-up and restore tool for Unix

43 Java Message Service, WebSphere Application Server, and Message Driven Beans

47 MQ news

A tc PUBLICATION

MQ Update

Published by

Xephon Inc
PO Box 550547
Dallas, Texas 75355
USA

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

Editor

Trevor Eddolls
E-mail: trevore@xephon.com

Publisher

Nicole Thomas
E-mail: nicole@xephon.com

Subscriptions and back-issues

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

Contributions

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

MQ Update on-line

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

SSL in WebSphere MQ 5.3

WebSphere MQ security is now enhanced with the introduction of support for Secure Sockets Layer (SSL), the Internet standard for secure communication. This article describes what SSL in WebSphere MQ 5.3 for z/OS offers and how to set it up.

WHAT IS SSL?

Secure Sockets Layer (SSL) technology is the standard Internet security protocol, designed to secure the data transmission over an insecure network.

SSL makes use of digital certificates to establish the identity of the two parties that want to establish an SSL connection. With this process, typically referred to as SSL handshake, a secure confidential communications 'pipe' is created between these two entities.

SSL basically addresses the following security issues:

- Impersonation – SSL handshake allows the two parties involved to be sure of each other's identity (identification and authentication service).
- Eavesdropping – data transmitted is encrypted to ensure that someone in between doesn't get access to the information sent (confidentiality service).
- Tampering with information – hash functions are used to detect whether someone has intercepted and changed the information (data integrity service).

Of course, the extent of the protection offered depends on whether the symmetric (secret) key or asymmetric (public and private key) approach is used. Also the length of the key influences this because it determines how quickly the key can be broken using a brute force approach. The standards key sizes (512, 768, and 1024 bit keys) provide low, medium, and high protection respectively (the larger the key size, the longer it takes to break!).

WEBSPHERE MQ AND SSL

SSL can be used to provide link level security with both MCA channels (for queue manager to queue manager communication) and MQI channels (for client applications connecting to a queue manager).

A digital certificate has to be obtained for each queue manager and each client user ID that wishes to communicate over an SSL secured channel. The digital certificates are maintained in a key repository.

The CipherSpec, which includes the encryption/decryption algorithm to be used by the SSL protocol, is specified in the channel definition.

Whenever the channel is started, the certificate given to the queue manager is used to prove its identity. After this handshake (during channel start), the message exchanges are encrypted using the algorithm specified in the CipherSpec defined for the channel.

SETTING UP WEBSPHERE MQ SSL IN Z/OS

Setting up SSL tasks

On z/OS, the number of server subtasks used for processing SSL calls is set up using the SSLTASKS parameter of the ALTER QMGR command. At least two server subtasks are required to use SSL channels. Though the range of values could be zero to 9999, IBM recommends that the SSLTASKS value doesn't exceed 50 – otherwise it is expected to result in storage allocation problems.

Associating key repository with queue manager

Having a key repository at each end of the connection is a prerequisite for SSL. The key repository mainly contains:

1 CA certificates from various certification authorities, which allow the queue manager to verify the certificates from its partners (from the other end of the connection) to establish their identity.

2 Personal certificate received from a certification authority. Each queue manager and WebSphere MQ client gets associated with a single certificate (using which they establish their identity to the other partner).

On z/OS, digital certificates are stored in a key ring that is managed by RACF (or other external security managers). Each queue manager must have access to a key repository. The steps to be followed in establishing this access are as follows:

1 Create a new key ring for the queue manager using the following command (userid is the user ID of the channel initiator address space):

```
RACDCERT ID(userid) ADDRING(ring-name)
```

2 Connect the relevant CA certificates to it using the command:

```
CONNECT(CERTAUTH LABEL('CA 1') RING(ring-name) USAGE(CERTAUTH))
```

3 Use the SSLKEYR parameter on the ALTER QMGR command to associate a key repository with a queue manager.

```
ALTER QMGR SSLKEYR(ring-name)
```

You need to add the personal certificate obtained from CA to the key ring. The steps involved are:

1 Add the certificate to the RACF database, specifying a label that would be used to associate the digital certificates with the queue manager. On z/OS, WebSphere MQ uses the ibmWebSphereMQ prefix followed by the name of the queue manager for the label name:

```
RACDCERT ID(userid) ADD(input-data-set-name) WITHLABEL('label-name')
```

2 Connect the personal certificate to the key ring created for the queue manager using:

```
CONNECT(ID(userid) LABEL('label-name') RING(ring-name) USAGE(PERSONAL))
```

To know more about managing the certificates – adding new certificates, deleting certificates, or transporting from another key ring – refer to the RACDCERT command.

In this context the only important point to note is that the changes to the certificates in the key ring and to the key repository attribute become effective only when the channel initiator is started or restarted.

Defining channels to use SSL

To use SSL, your channel must be defined accordingly using the three SSL parameters – SSLCIPH, SSLPEER, and SSLCAUTH – in the DEFINE CHANNEL command. Only the SSLCIPH parameter is mandatory if you want your channel to use SSL.

The SSLCIPH parameter is used to specify the CipherSpec used on the channel. The CipherSpec actually determines the hash algorithm (MD5/SHA), encryption algorithm (AES/DES/3DES/RC2/RC4/none), and number of encryption bits. Obviously, the CipherSpec used should be the same on both ends of the channel – allowing the decryption of data encrypted by the other partner. For the list of all possible values of CipherSpec, refer to the *WebSphere MQ Script (MQSC) Command* reference manual.

According to SSL protocol, it is mandatory for the SSL client (the initiating end of the channel) to obtain and validate the certificate of the SSL server during the SSL handshake, whereas the SSL client authentication is optional.

In WebSphere MQ, SSL client authentication becomes mandatory under the following conditions:

- Specifying SSLCAUTH as ‘REQUIRED’.
- Specifying the SSLPEER parameter, which defines the filter used to compare the identified name of the certificate sent by the SSL client (peer). If the identified name received from the peer doesn’t match the one specified, the authentication fails and the channel does not start.

Note that the SSL server always validates the client certificate if one is sent (even if SSLCAUTH is set as OPTIONAL).

CONCLUSION

The SSL support provided by WebSphere MQ provides authentication, message integrity checking, and data encryption for messages when they travel across the Internet. It is important to understand that link level security offered by WebSphere MQ protects messages while they are being transferred from one queue manager to another (especially useful when messages are transmitted over an insecure network). But this doesn’t include protection of the messages when they are stored in queues, and should be sufficient when the queue managers are running in a controlled and trusted environment.

Sasirekha Cota
Tata Consultancy Services (India)

WebSphere Translation Server

With more and more businesses being international in nature, there will come a time when somebody in one of those companies is presented with a document in a language they do not understand. You can’t sell to someone if they don’t know what you’re offering, and you daren’t buy from someone if you don’t understand the contract!

Step forward IBM's WebSphere Translation Server for Multiplatforms, which is now at Version 5. This product provides machine translation, and is specifically geared towards companies that want to provide pages on their Web sites in the reader's native language. And, of course, it is designed to do its job at a cost to the company that uses it that is much less than a room full of human translators. The product is also designed to perform its translation in real-time.

The IBM WebSphere Translation Server for Multiplatforms 5.0 is a machine translation (MT) offering that can help companies remove language as a barrier to global communication and e-commerce. WebSphere Translation Server (WTS) enables enterprises to provide content in multiple languages in real time. Specifically designed for enterprise use, the WebSphere Translation Server allows companies to leverage their existing Web infrastructure to provide content to users in their native language, at a fraction of the cost of professional translation.

WebSphere Translation Server (WTS) is based on IBM machine translation technology. It can run on a dedicated server, using Java Remote Method Invocation (RMI) and Java protocol to communicate with the WebSphere Application Server. In addition, Web page HTML content can be translation-enabled to support HTTP servers from Apache, IBM, Microsoft, or Netscape – hence the 'Multiplatform' part of its name.

WTS consists of:

- Machine translation engines for translating text from one language to another (eg French to English).
- User Dictionary Manager tools, which allow specific words to be added to a domain. What this means is that slang or technical terms can be added as a way of tuning for a specific application.
- Support for WebSphere Application Server (WAS) and HTTP servers from Apache, IBM, Microsoft, and Netscape.

So, before you rush out and buy it, you probably want to know what languages the product can translate. The current list is:

- English-to-French/French-to-English
- English-to-German/German-to-English
- English-to-Italian/Italian-to-English
- English-to-Spanish/Spanish-to-English
- English-to-Chinese (traditional)/Chinese (traditional)-to-English
- English-to-Chinese (simplified)/Chinese (simplified)-to-English
- English-to-Japanese/Japanese-to-English
- English-to-Korean
- English-to-Brazilian Portuguese.

Could well be worth a look for companies that are trading (or trying to trade) in areas where those languages are spoken.

Nick Nourse
Independent Consultant (UK)

E-mail alerts

Our e-mail alert service will notify you when new issues of *MQ Update* have been placed on our Web site. If you'd like to sign up, go to http://www.xephon.com/mq and click the 'Receive an e-mail alert' link.

MQ V5.3 for z/OS page set removal procedure

BACKGROUND

The administration tasks to manage page sets are described in the IBM supplied *System Administration Guide*, specifically Chapter 10, 'Managing page sets'.

At one particular customer's site, some page sets were hardly used and had to be removed.

The addition of page sets is a well-known procedure, but their removal was not described in any of the manuals. This article aims to describe this procedure with the hope that IBM will add it to their manuals.

PROCEDURE

This is not for the faint-hearted because it involves removing recovery information from a system that is working fine, and effectively 'cold' starting it!

The reasons for wanting and needing to remove page sets were:

- Valuable disk space was being used. This is because even an empty page set uses up space because it is pre-formatted.
- Part of the back-up and recovery procedure demanded that all page sets be backed up, which in the case of nearly 'empty' page sets means more wasted space and CPU cycles.

The important part is to ensure that all queue definitions are moved from the page set(s) to be removed to another page set.

As a reminder, this is the mapping used by MQ for z/OS:

- Queue definition to storage class
- Storage class to page set
- Page set to buffer pool.

It is assumed that the reader is already familiar with the way that queue definitions can be moved, but here is a quick summary:

- Find an appropriate storage class or define a new one.
- If messages reside on queues, back them up to a dataset or move them to ‘back-up’ queues. Ensure the back-up queues are not on the page sets to be emptied!
- Alter the queue definition to use the new storage class.
- Move any messages back onto the queue.
- Finally, alter the original storage class to point to a different page set.

Note that in some cases this may report an error message like ‘STGCLASS(XXXX) IS CURRENTLY IN USE’.

Eventually, a stage will be reached where the page sets selected to be removed will be devoid of queues. Double-check this by issuing these commands:

- **DIS QL(*) PSID(n)** – where *n* is the pageset id, eg 18.
- **DIS USAGE(*)**

An important part to understand is that MQ maintains recovery information within the page sets and even an empty page set needs a recovery point.

This is true even if the page sets are ‘commented’ out of the start-up.

The author prefers not to comment them out as they produce ‘OFFLINE’ warning messages and MQ is still ‘aware’ of them.

One way to see how MQ maintains recovery information is to look at the joblog of the MQ master address space. At start-up, shutdown, and at periods of update activity, system checkpoints are taken. These checkpoints record the state of the system and include the so-called ‘recovery RBA’ of the pagesets. The recovery RBA is stored in several places:

- Checkpoints on the log.
- On every physical (4K) page that has been changed. One ‘special’ place is the first page (page 0) of each page set, which holds the lowest recovery RBA of all its pages and this is used at system recovery time in order to see whether recovery is required on that page set.

To remove the recovery information for the page set being removed from the log, it is necessary to restart the queue manager with new logs. This can be safely done only if the queue manager has been shut down cleanly so that its entire state is recorded consistently on the page sets.

In order to ensure that all recoverable resources were safely on the archived logs and no activity was missed, the following two-stage process was used.

SHUTDOWN PHASE 1

Step 01 – stop the queue managers. Use the **STOP QMGR MODE(FORCE)** command and check that it worked by ensuring there were no ‘in-doubt’ threads.

What we’re trying to achieve is a 100% clean shutdown of MQ with all recoverable resources on the archived logs. At the client’s site, however, this was not possible because a number of applications (internal and external) had not been coded with the ‘FAIL-IF-QUIESCING’ option, which would tell the application that the system is shutting down.

So in order to force applications to detach from the queue manager, we had no option but to use MODE(FORCE).

Step 02 – stop all external MQ activity. This includes batch, CICS, IMS, and any automation products.

SHUTDOWN PHASE 2

Step 03 – restart the queue managers. As soon as they're up, stop the channel initiators. What we're really doing here is preventing any clients getting on to the queue manager (if the CHIN won't come down, CANCEL it). An alternative method to ensure no-one uses the system is to alter the connection rules in the external security manager.

Step 04 – copy active logs onto the archived logs. Use the command **ARCHIVE LOG**.

Step 05 – stop the queue manager(s) with MODE(QUIESCE) (if possible).

Step 06 – double-check that archived logs were created as well as a back-up of the BSDS (see the archived 'B' datasets).

Step 07 – back up the current set of page sets – all of them! During normal running, the normal back-up procedures ran while the queue managers were active (so-called 'fuzzy' back-ups) and was a multi-step job causing the page sets to be backed up in single-stream mode.

At this point, however, in order to save time, each page set was backed up by its own job, which ran in parallel.

Note: steps 8–11 can be done while step 7 is running.

Step 08 – back up all the active logs and BSDS, plus their DUAL copies (eg via DFDSS).

Step 09 – back up the contents of any shared queues using the command ***BACKUP CFSTRUCT(x)*** *for each cfstruct(x) in the queue sharing group.* Performing the back-up on another queue manager in the queue sharing group after the subject queue manager has been stopped will ensure that recovery from the back-up would not require any log data from the subject queue manager.

Step 10 – delete and redefine the active log datasets and dual copies. This is required because MQ will have written certain ‘page set set control’ records, telling the system what RBA ranges are required for media recovery. This information is not wanted because we want to remove the page sets.

Note: the archived log datasets (on cartridge) are left ‘as is’.

Step 11 – delete and redefine the BSDS and its dual copy. This is required because it is an inventory of checkpoints and logs needed for recovery, and recovery is not what is wanted – provided, of course, the queue manager came down cleanly!

Step 12 – run the BSDS log change utility, CSQJU003. This is required to store the names of the (new) active logs in the BSDS. It is, in fact, exactly the same job that was run when the queue manager was first set up.

Step 13 – remove the recovery information from each of the page sets (including the ones being removed). Use the CSQUTIL command **RESETPAGE FORCE**.

This can take a relatively long time as the utility has to alter each 4K page because each page has an associated recovery RBA.

Note: steps 14–15 can be done while step 13 is running.

Step 14 – alter the MSTR start-up JCL by removing references to the page sets to be removed.

Step 15 – alter the CSQINP1 members by removing references to the page-set-to-bufferpool mapping for the relevant page sets.

Step 16 – restart the queue manager. So what will happen at restart? Here are some excerpts from the MQ MSTR joblog that should be present:

```
08.49.50 STC04353  CSQJ127I ?QMP1 SYSTEM TIME STAMP FOR BSDS=*********
********.**
08.49.52 STC04353  CSQJ001I ?QMP1 CURRENT COPY 1 ACTIVE LOG DATA SET IS
874
    874            DSNAME=QMP1.LOGCOPY1.DS01, STARTRBA=000000000000
ENDRBA=00002BF1FFFF
08.49.52 STC04353  CSQJ001I ?QMP1 CURRENT COPY 2 ACTIVE LOG DATA SET IS
```

```
875
    875             DSNAME=QMP1.LOGCOPY2.DSØ1, STARTRBA=ØØØØØØØØØØØØ
ENDRBA=ØØØØ2BF1FFFF
Ø8.49.52 STCØ4353  CSQJØ99I ?QMP1 LOG RECORDING TO COMMENCE WITH  876
876 STARTRBA=ØØØØØØØØØØØØ

Ø8.49.53 STCØ4353  CSQRØØ1I ?QMP1 RESTART INITIATED
Ø8.49.53 STCØ4353  CSQRØØ3I ?QMP1 RESTART - PRIOR CHECKPOINT
RBA=ØØØØØØØØØØØØ
Ø8.49.53 STCØ4353  CSQRØØ4I ?QMP1 RESTART - UR COUNTS -  91Ø
    91Ø             IN COMMIT=Ø, INDOUBT=Ø, INFLIGHT=Ø, IN BACKOUT=Ø
Ø8.49.53 STCØ4353  CSQIØ49I ?QMP1 Page set Ø has media recovery  911
    911             RBA=ØØØØØØØØØØØØ, checkpoint RBA=FFFFFFFFFFFF
Ø8.49.53 STCØ4353  CSQIØ49I ?QMP1 Page set 1 has media recovery  912
912  RBA=ØØØØØØØØØØØØ, checkpoint RBA=FFFFFFFFFFFF
```

etc – it is the same for the rest of the page sets.

```
Ø8.49.54 STCØ4353  CSQRØ3ØI ?QMP1 Forward recovery log range  928
    928             from RBA=ØØØØØØØØØØØØ to RBA=ØØØØØØØØØØØØ
Ø8.49.54 STCØ4353  CSQRØØ5I ?QMP1 RESTART - FORWARD RECOVERY COMPLETE -
929
    929             IN COMMIT=Ø, INDOUBT=Ø
Ø8.49.54 STCØ4353  CSQRØ32I ?QMP1 Backward recovery log range  93Ø
    93Ø             from RBA=ØØØØØØØØØØØØ to RBA=ØØØØØØØØØØØØ
Ø8.49.54 STCØ4353  CSQRØØ6I ?QMP1 RESTART - BACKWARD RECOVERY COMPLETE -
931
    931             INFLIGHT=Ø, IN BACKOUT=Ø
Ø8.49.58 STCØ4353  CSQRØØ2I ?QMP1 RESTART COMPLETED

Ø8.49.58 STCØ4353  CSQPØ18I ?QMP1 CSQPBCKW CHECKPOINT STARTED FOR ALL
BUFFER POOLS
Ø8.49.58 STCØ4353  ?QMP1 DISPLAY THREAD(*) TYPE(INDOUBT)
Ø8.49.58 STCØ4353  CSQPØ21I ?QMP1 Page set Ø new media recovery  935
    935             RBA=ØØØØØØØØØ8A4, checkpoint RBA=ØØØØØØØØØ8A4
Ø8.49.58 STCØ4353  S QMP1CHIN
Ø8.49.58 STCØ4353  CSQPØ19I ?QMP1 CSQP1DWP CHECKPOINT COMPLETED FOR  937
    937             BUFFER POOL 2, 28 PAGES WRITTEN
Ø8.49.58 STCØ4353  CSQPØ21I ?QMP1 Page set 1 new media recovery  938
    938             RBA=ØØØØØØØØØØØØ, checkpoint RBA=ØØØØØØØØØØØØ
Ø8.49.58 STCØ4353  CSQPØ19I ?QMP1 CSQP1DWP CHECKPOINT COMPLETED FOR  939
    939             BUFFER POOL 3, 47 PAGES WRITTEN
Ø8.49.58 STCØ4353  CSQPØ21I ?QMP1 Page set 2 new media recovery  94Ø
94Ø  RBA=ØØØØØØØØØF66, checkpoint RBA=ØØØØØØØØØF66
```

etc, followed by similar output for the rest of the page sets.

Step 17 – issue another ***BACKUP CFSTRUCT*** *to establish a new point of recovery for the messages in the CF structures.*

```
Ø8.5Ø.38 STCØ4353  CSQE1Ø5I ?QMP1 BACKUP task initiated for structure
CC1
Ø8.5Ø.38 STCØ4353  CSQE12ØI ?QMP1 Backup of structure CC1 started at
RBA=ØØØØØØØØDED8
Ø8.5Ø.38 STCØ4353  CSQ9Ø22I ?QMP1 CSQELRBK 'CFSTRUCT CC1' NORMAL
COMPLETION
Ø8.5Ø.4Ø STCØ4353  CSQE121I ?QMP1 CSQELBK1 Backup of structure CC1  Ø85
    Ø85             completed at RBA=ØØØØØØ4E2BA1, size 5 MB
```

The output is similar for the other application structures.

```
Ø8.51.Ø6 STCØ4353  CSQJ31ØI ?QMP1 ASYNCHRONOUS ARCHIVE LOG COMMAND  114
    114             QUIESCE PROCESSING STARTING FOR MAXIMUM OF   5
SECONDS
Ø8.51.Ø6 STCØ4353  CSQJØØ2I ?QMP1 END OF ACTIVE LOG DATA SET  115
    115             DSNAME=QMP1.LOGCOPY1.DSØ1, STARTRBA=ØØØØØØØØØØØØ
ENDRBA=ØØØØØØ686FFF
Ø8.51.Ø6 STCØ4353  CSQJØØ1I ?QMP1 CURRENT COPY 1 ACTIVE LOG DATA SET IS
116
    116             DSNAME=QMP1.LOGCOPY1.DSØ2, STARTRBA=ØØØØØØ687ØØØ
ENDRBA=ØØØØ2C5A6FFF
Ø8.51.Ø7 STCØ4353  CSQJØØ2I ?QMP1 END OF ACTIVE LOG DATA SET  117
    117             DSNAME=QMP1.LOGCOPY2.DSØ1, STARTRBA=ØØØØØØØØØØØØ
ENDRBA=ØØØØØØ686FFF
Ø8.51.Ø7 STCØ4353  CSQJØØ1I ?QMP1 CURRENT COPY 2 ACTIVE LOG DATA SET IS
118
    118             DSNAME=QMP1.LOGCOPY2.DSØ2, STARTRBA=ØØØØØØ687ØØØ
ENDRBA=ØØØØ2C5A6FFF
Ø8.51.Ø7 STCØ4353  CSQJ311I ?QMP1 ASYNCHRONOUS LOG ARCHIVE (OFFLOAD)
TASK INITIATED
Ø8.51.Ø7 STCØ4353  CSQJ312I ?QMP1 ARCHIVE LOG QUIESCE ENDED, UPDATE  12Ø
    12Ø             ACTIVITY IS NOW RESUMED
```

Step 18 – testing.

Some suggestions follow. Check that:

- Existing messages can still be accessed (browse).
- New messages can be added and deleted (both via batch and on-line).

Step 19 – update the regular page set back-up jobs and delete the removed page sets and their back ups.

Do this only when the system has been up and running for a day or so.

BACKOUT SCENARIO

If for some reason the change had to be backed out, use the following procedure to keep the page sets that remained allocated and add the ‘old’ page sets back in (even though they are empty!).

Step B1 – shut down the queue manager using a standard **STOP FORCE**.

Step B2 – reintroduce the page sets’ DD statements to MSTR start-up JCL.

Step B3 – reintroduce the **DEF PSID** commands to CSQINP1.

Step B4 – start the queue manager.

There is no need to go through all the previous steps because we want to retain the status of the page sets that remained allocated and allow MQ to resolve any recoveries.

RECOMMENDATIONS

I recommend that you:

- Plan this operation carefully, including its back-out.
- Work out the longest part of the operation, namely the back-up and ‘resetpage’ of the largest page set(s).
- Test it out in a pre-development environment.
- In a queue-sharing environment, perform the change on one queue manager at a time. In this way, persistent shared messages remain recoverable via the logs of other queue managers in the queue sharing group.
- Keep one page set available for emergencies.
- Do the change in three stages: stage 1 to move all queues and their contents off the page sets; stage 2 to remove the page sets from MQ; and stage 3 to physically remove the page sets after one or two days of successful runs.

- Update the disaster recovery procedures and re-test.

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

Setting up a WebSphere MQ Integrator Broker in a parallel Sysplex

INTRODUCTION

WebSphere MQ messages can be processed using IBM's WebSphere MQ Integrator Broker product. The broker can be used for a wide range of message volumes because the broker is highly scalable by having multiple message flow instances using threads and by multiple processes when deploying the message flow to multiple execution groups. But there are cases where the volume of messages that can be processed on one computer is not high enough for the business need. In this case a second broker is needed on another computer to increase the throughput. But, in addition to throughput, there are also other reasons for running a

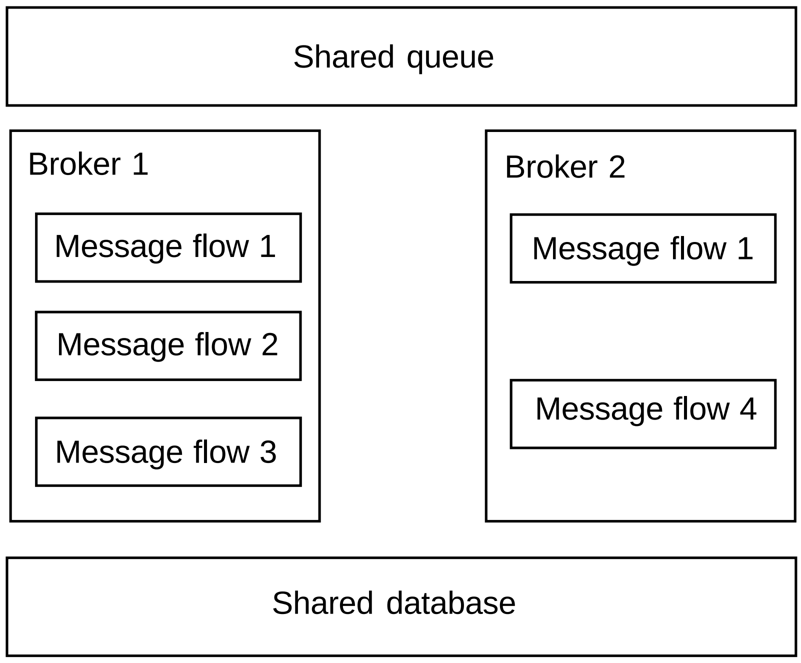

Figure 1: WebSphere BI for FN Instance with two WebSphere MQ Integrator Brokers

message flow on another computer: for example the availability of the service provided by a message has to be increased, or the costs to access the message flow from another application running on another computer are too high.

Such requirements arise for WebSphere Business Integration for Financial Networks (hereafter called WebSphere BI for FN). This product is designed for high message throughput and high availability. WebSphere BI for FN allows a configuration with message flows on multiple brokers as shown in Figure 1. Message Flow 1 in this figure could be the message flow that needs higher throughput or better availability. The other message flow could be used for administration purposes or for other low-volume message processing. The concepts described here for two brokers apply also to configurations with any higher number of brokers.

The WebSphere BI for FN product is divided into a base part and network-specific extensions. WebSphere BI for FN Base provides functionality to deliver products on top of WebSphere MQ Integrator Broker and a set of common functions that are required by different product extensions, like auditing, security, or configuration. Configuration describes the run-time behaviour of a message flow. This requires shared queues to share the workload and a shared database to access the same data as shown in Figure 1.

This article describes possible ways to set-up multiple WebSphere MQ Integrator Brokers for z/OS in a parallel Sysplex, and the decisions that have to be taken while defining the system.

WEBSPHERE MQ

Each WebSphere MQ Integrator Broker requires its own WebSphere MQ queue manager. Using WebSphere MQ there are three possible ways to set up queue managers in a parallel Sysplex. The first possibility is that the queue managers are unrelated, the queue managers can be part of a cluster of queue managers, and, thirdly, the queue managers can be part of a queue-sharing group.

Unrelated queue managers

If the queue managers are unrelated, each broker is processing just the workload that applications are addressing directly to it. With this system set-up, the volume of messages being processed can be increased, but there is no workload balancing and no failover in case one of the brokers, the queue managers, or an entire system, fails.

Queue manager clusters

A WebSphere MQ cluster, as shown in Figure 2, is a connection between two independent computers. This is possible on most WebSphere MQ-supported platforms including z/OS.

In a cluster, each WebSphere MQ Integrator broker is connected to a separate WebSphere MQ queue manager. This allows each broker to process the workload that is addressed to its queue manager. Workload balancing between both brokers can be achieved if applications sending the

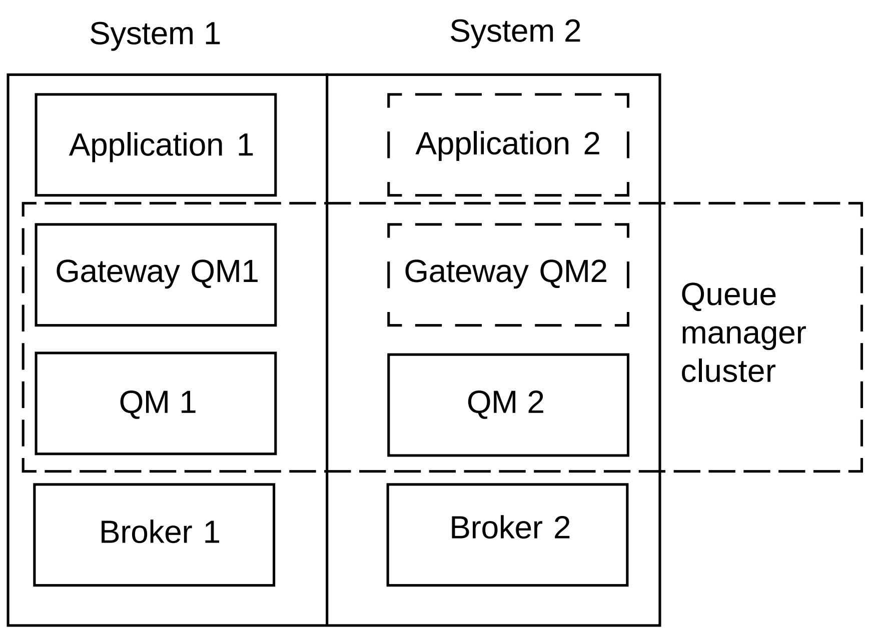

Figure 2: A WebSphere MQ cluster

messages are connected to a separate queue manager, named Gateway QM in Figure 2, which is also part of the same WebSphere MQ cluster. The benefits of workload balancing come from the costs for remote messaging. These costs are processing costs and latency.

The system works as follows. The application, Application 1, connects to the queue manager, Gateway QM1. It is sending a message to the input queue of the message flow. The address is just using the name of the queue with the queue manager name left blank. The queue manager Gateway QM1 automatically finds out the name of the queue manager where the queue is located. To be able to do this, the input queue of the message flow must be defined as a cluster queue. This way, the information about the availability of this queue is distributed to all queue managers in the cluster. For additional throughput and availability, a queue with the same name is defined on both queue managers, QM 1 and QM 2. Since this information is available to Gateway QM, this queue manager can decide which one to use. The default behaviour in such a case is that the queue manager selects the queue on the other queue manager, in a round-robin sequence. This workload distribution algorithm is only performed for all queues with the same name on queue managers that are up and running and connected to the cluster.

As long as everything is up and running, the system works well. There are some drawbacks in the case of system failures or the failure of a broker. If a complete system fails, eg System 2 falls down, the queue manager cluster detects this and will not send any more messages to the system until it is started again. But messages that are already on System 2 at the time of the failure will no longer be processed. Also messages that have already been routed to the queue manager on System 2 will stay in the transmission queue to this system. They will not automatically be re-routed to any system that is active. The situation is much worse if just the broker on the system fails. In this case the Gateway QM1 continues to direct messages to the queue manager on System 2 even if they are not processed. In such a case, either an operator needs to shut down the queue manager or, if the broker can be recovered, the broker has to process a large backlog of messages. To overcome the problem of messages that are still in the queue and are not processed, you may configure the broker and the queue manager in a way that they can be restarted on another system, for example System 1. But this still has the problem of a potentially large backlog of messages that need to be processed at start-up time.

Figure 2 also shows an optional gateway queue manager, Gateway QM2. This is not required for the availability of the message flows, but having a second gateway queue manager is a good choice to increase the availability and message throughput of sending and receiving applications.

Queue sharing groups

A system with better availability and workload balancing characteristics is a queue sharing group. Such a system is

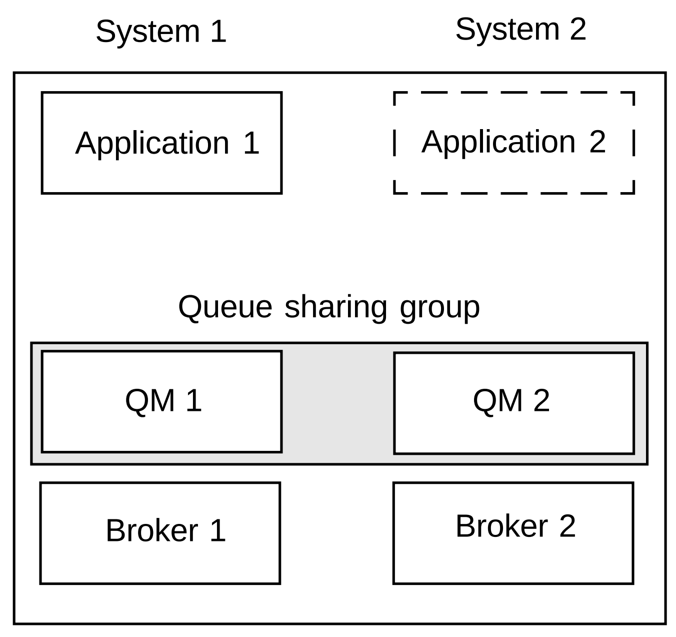

Figure 3: A WebSphere MQ sharing group

shown in Figure 3. What's obvious when comparing this figure to Figure 2 is that no gateway queue manager is required. A queue-sharing group is available only on z/OS. All queue managers in the sharing group share some common resources, eg all queue managers must have access to common files where messages in shared queues can be stored.

The applications are sending their messages directly into the input queue of the message flow. This queue must be defined as a sharing group queue in the queue managers. This way, the queue appears to be a single local queue that spans System 1 and System 2. The handling and coordination of messages across the systems is achieved by WebSphere MQ exploiting the coupling facility. The coupling facility can be thought of as shared memory between both systems.

For the message flows, it appears as if each of their queue managers has a local queue from which they get their messages. Each message flow will take as many messages as it can process. This means that there is an automated workload distribution according to the processing capabilities of each system.

This set-up has the advantage that if one broker fails, the message flow in the second broker can process all the messages, albeit with a lower throughput rate. If it's just the broker that is failing, then System 2 is able to recover messages that were in-flight at the time of the failure, and these messages can then be processed on the remaining broker. The broker and the message flow in the available broker will not be aware of the fact that the second broker is no longer available and they do not have to take any action.

The behaviour is similar if a complete system fails. This is detected by the remaining queue manager, which makes all the messages available to the remaining system and those messages can be processed. The only problem here might be for those messages that were processing when the system failed. If these messages cannot be recovered until System 2 is back, they get into an in-doubt status and cannot be processed.

The implementation of the queue-sharing group has some limitations, for example the total amount of data that can be held in such queues and the maximum length of a message in a sharing group queue. The total amount of data depends on the available resources for the coupling facility that can be used by WebSphere MQ. These coupling facility resources may have to be shared with other resource managers, for example a database. Messages in a sharing group queue are currently limited in length to a maximum of around about 63KB.

If a broker application consists of multiple message flows, like most WebSphere BI for FN extensions, you will need to check which queues really need to be sharing group queues. Only those that require high availability and fail-over capabilities should be selected to reduce to a minimum the required resources in the coupling facility.

Based on the advantages and disadvantages of all methods,

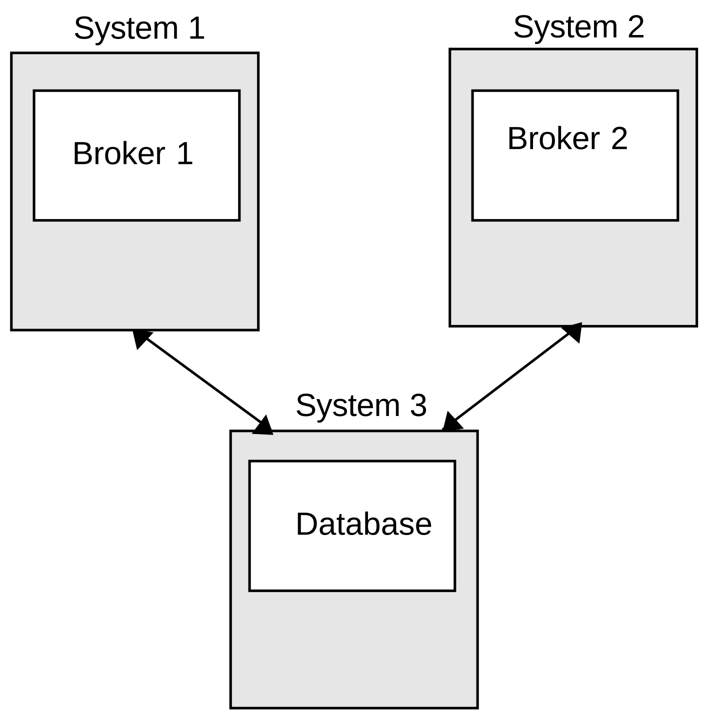

Figure 4: Sharing data in a distributed environment

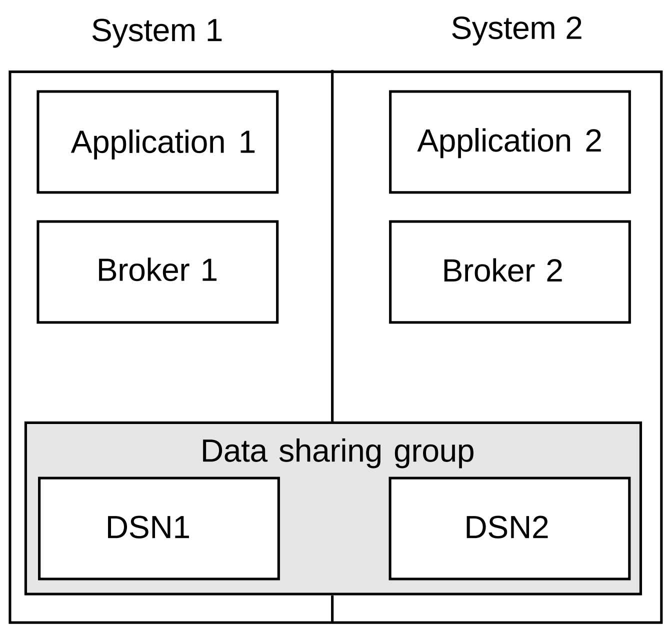

Figure 5: Broker with data sharing group

WebSphere BI for FN has decided to start with support of a queue manager cluster. The main reasons for this decision are that those messages, which are processed by the existing extensions, can exceed 63KB and the cluster approach also works on all other WebSphere BI for FN supported platforms.

DATABASE

If you are processing data with message flows in multiple brokers, then all processing should work with the same data. In distributed environments, this problem is usually solved by having one database, which is referenced by all brokers. The database for such a configuration could be on the system where one of the brokers reside or a separate system, as shown in Figure 4.

Such a system has the drawback that System 3 is a single point of failure. To decrease the impact on availability if this system fails, a standard mechanism (such as a cold standby

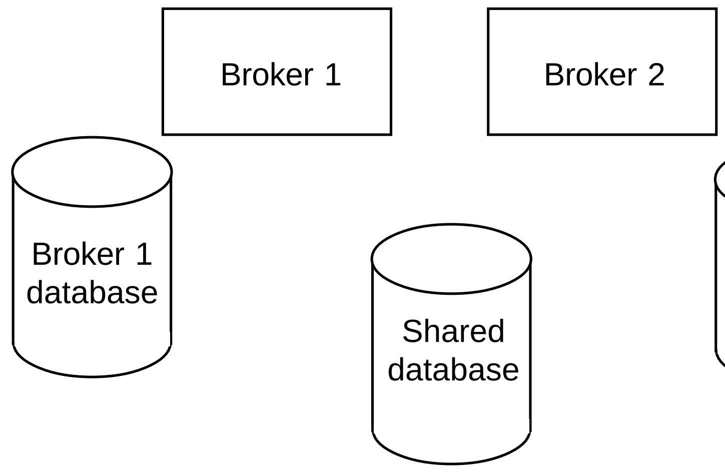

Figure 6: Data distribution for a data sharing group

system) can be applied. Another drawback with this set-up is the communication costs involved for communicating from the broker systems to the database system. For a low volume of messages this may not be relevant, but, when processing high volumes of messages with many database interactions, the processing costs and latency introduced by a network could be significant.

On z/OS there's another possibility with DB2 (the only supported database on this platform). Similar to a queue sharing group, database subsystems can be organized into a data sharing group as shown in Figure 5. In such a configuration there's a database running on each server but the databases can communicate with each other using a coupling facility and therefore process the same data in the same tables.

Each database in the data-sharing group has its own subsystem id and its own ODBC data source name (DSN). Any application could use this name to connect to this member of the data sharing group as usual. The data-sharing group also has its own identification and hence its own DSN. Any request addressed to this DSN is then routed by the sharing group to a sharing group member that can process it.

When setting up the broker in such an environment, the broker tables could be located non-shared in the corresponding database subsystem that is located on the same system as the broker. For example Broker 1 would be customized to have its broker database tables in DSN1. Database tables for application message flows that are available to both brokers need to be shared so they are accessible by both brokers. They can be defined in any of the subsystems. The message flows accessing such shared tables would reference them using the DSN of the data-sharing group. Such a set-up is shown in Figure 6.

WEBSPHERE MQ INTEGRATOR BROKER

In addition to the underlying subsystems, some precautions also have to be taken for the broker. First of all, it must be assured that all processing components for the message flows are at the same level. In addition to the broker executables this must also be assured for all plug-ins and libraries added by the broker application – in this case WebSphere BI for FN. In general this could be assured by always installing the same level on both systems. This could be error prone so common data could not be assured. It should be possible to install the data on one system and also make this data available to all other systems, for example by using Network File System services (NFS). Such an approach has the disadvantage that this introduces a new single point of failure. If the system where the installation data is located fails, all the other systems are unable to work.

z/OS supports the Hierarchical File System (HFS). This is a single dataset on a disk that can be accessed by multiple systems. Since all plug-in files and shared libraries used by WebSphere MQ Integrator Broker and WebSphere BI for FN are read-only, there are no sharing or locking problems. When designing WebSphere BI for FN, attention was paid to the fact that no files need to be accessed in write or update mode.

Having all executables at the same level is also required for any other kind of program that is involved in the message

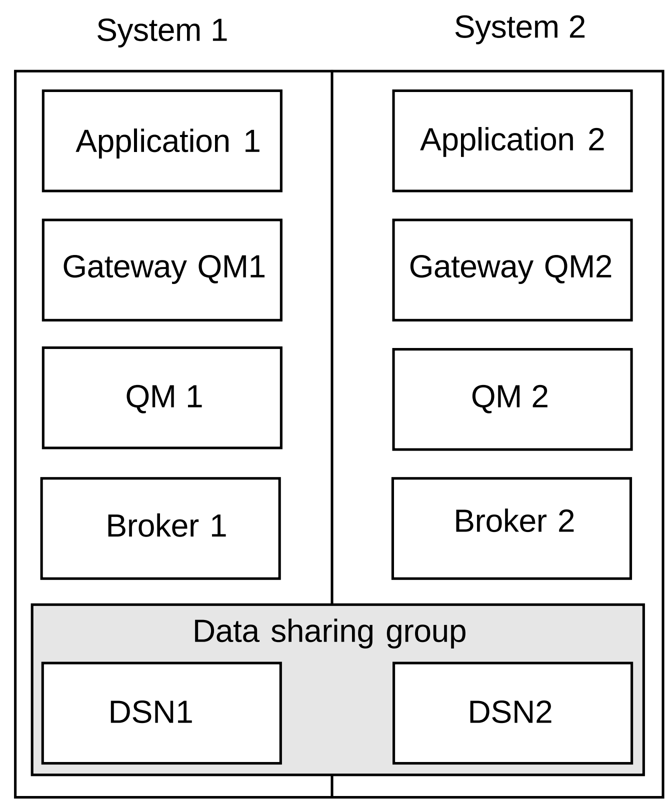

Figure 7: Final multiple broker configuration

processing. One example is that WebSphere BI for FN uses DB2 stored procedures to do some processing. Other examples could be programs invoked by a flow or application used to initiate messages or to process the result of the activities in the message flows.

Not only the executables of the products have to be on the same level, but so do the message flows processing the messages. To ensure this, the flow should be administered in such a way that, after every change to a main flow or a sub-flow, all instances of the flow are brought to the new level. With WebSphere MQ Integrator Broker this can be achieved by doing a delta deploy on the topology level after any change to a message flow. Deploying identical message flows to different brokers is possible if the resources they access, mainly WebSphere MQ queue and DB2 tables, have the same names. This is achieved by the use of shared queues and the shared database.

SUMMARY

WebSphere Business Integration for Financial Networks has shown that it is possible to set up WebSphere MQ Integrator Broker in a parallel Sysplex. It was achieved with a WebSphere MQ cluster and data in a data-sharing group as shown in Figure 7.

With this configuration higher throughput and higher availability are achieved. Nevertheless, use performance statistics to validate whether this is achieved with a reasonable processing overhead, compared with processing on a single server, are still outstanding.

Michael Groetzner
IBM (Germany)

Queue back-up and restore tool for Unix

INTRODUCTION

Sometimes it is useful to copy or move WebSphere MQ messages from one queue to another queue or to a file for later use. The stored messages may be used for development testing (to create the same input data several times) or for analysing off-line after a problem has occurred. It may also be used to extract test data from a production system, or to rescue messages when a queue manager has to be recreated – eg to increase the size of the log files.

THE QUEUE BACK-UP AND RESTORE TOOL

Intention for backupQ

I have worked for several years within a heterogeneous WebSphere MQ environment. On z/OS there is a utility, CSQUTIL, to back up and restore queue contents. The Unix guys often asked for such a tool on their systems. So I created the program backupQ, to enable the Unix administrators to back-up and restore queue contents.

How backupQ works

The program backupQ will move or copy messages from one queue to another queue or to a file, and from a file back to a queue. The program has the following parameters:

- -f ...: – a function with the following possible values:
  - q2q: – copy or move from an input queue to a target queue.
  - q2f: – copy or move from an input queue to a file.
  - f2q: – copy from a file to a queue.

- -i ...: – the name of the input queue or file.
- -o ...: – the name of the output queue or file.
- -m ...: – optional; the name of the queue manager (if not configured as the default queue manager).
- -d: – optional; move messages instead of copying them (destructive get, not for input from a file and not in combination with option -r).
- -r first,last: – optional; range of the messages to copy or move (not in combination with option -d). If the last value is not set, all messages up to the end of the queue or file will be copied.

Files created by backupQ always contain the message descriptor. This will be partially restored by backupQ when the messages are copied back to a queue. Partially means that attributes like persistency are preserved, whereas attributes like the put time and date will be set by WebSphere MQ. BackupQ creates a start and an end tag, to identify files created by itself. If these tags are missing, backupQ returns an error message. It is also possible to use manually-created plain text files, eg to create some test data. Such a file needs a manually-added start and an end tag (BACKUPQ_START_OF_FILE and BACKUPQ_END_OF_FILE respectively) in the first and the last line, to be used by backupQ. BackupQ then reads the contents, line by line, and puts each line as a new message on the queue.

Building the program backupQ

The following lines create the binary – on non-DCE platforms – from the file backupQ.c for AIX and Sun Solaris systems. I assume that the GNU compiler gcc is installed in */usr/local/bin*.

AIX:

```
/usr/local/bin/gcc -o backupQ -lmqm backupQ.c
```

SunOS:

```
/usr/local/bin/gcc -o backupQ -lmqm -lmqmcs -lmqmzse -lsocket -lnsl -ldl
backupQ.c
```

How to build the software on further platforms is described in the IBM document *WebSphere MQ Application Programming Guide*.

Installation of the queue back-up and restore tool

There is nothing to install, just copy the binary to your program search path.

EXAMPLES

Example 1

Copy the whole contents of a queue to another queue. Leave the messages in the original queue. The program connects to the default queue manager:

```
backupQ -f q2q -i InputQueue -o OutputQueue
```

Example 2

Move the whole contents of a queue to another queue. The messages in the original queue are deleted. The program connects to the default queue manager:

```
backupQ -f q2q -i InputQueue -o OutputQueue -d
```

Example 3

Move the whole contents of a queue to a file. The messages in the queue are deleted. The program connects to the queue manager TESTQM:

```
backupQ -f q2f -i InputQueue -o OutputFile -d -m TESTQM
```

Example 4

Copy the whole contents of a file to a queue. The messages in the file are preserved. The program connects to the queue manager TESTQM (the option -d is not available in combination with a file as an input device):

```
backupQ -f f2q -i InputFile -o OutputQueue -m TESTQM
```

Example 5

Copy the 10th to 20th message of a file to a queue (this means 11 messages in total). The messages in the file are preserved. The program connects to the default queue manager (the option -d is not available in combination with a file as an input device or the option -r):

```
backupQ -f f2q -i InputFile -o OutputQueue -r 10,20
```

Example 6

Copy the 30th to the last message of an input queue to an output queue. The messages in the input queue are preserved. The program connects to the queue manager TESTQM (the option -d is not available in combination with the option -r):

```
backupQ -f q2q -i InputQueue -o OutputQueue -r 30 -m TESTQM
```

Example 7

Copy the 10th message of an input to an output queue. The message in the input queue is preserved. The program connects to the default queue manager (the option -d is not available in combination with the option -r):

```
backupQ -f q2q -i InputQueue -o OutputQueue -r 10,10
```

DESCRIPTION OF THE CODE

The code consists of several parts.

Global parameters

Two groups of global parameters are used in the program. The parameters beginning with FUNC define the types of input and output device and whether read messages will be removed (this is called the function of the program):

```
 62: /* Define some flags */
 63: #define FUNC_NONE    0    /* nothing to do */
 64: #define FUNC_Q2Q     1    /* copy or move from a queue to another
queue */
 65: #define FUNC_Q2F     2    /* copy or move from a queue to a file */
 66: #define FUNC_F2Q     4    /* copy from a file to a queue */
 67: #define FUNC_MOVE    8  /* move messages instead of copying them */
```

The parameters beginning with FILE define some strings that are used as marks in the input or output file:

```
 69: /* Define some file parameters */
 70: #define FILE_MD_HEADER    "MQMD"    /* message line contains a
descriptor */
 71: #define FILE_MQ_START_TAG "BACKUPQ_START_OF_FILE"  /* start tag of
files */
 72: #define FILE_MQ_END_TAG   "BACKUPQ_END_OF_FILE"    /* end tag of
files */
```

Function main

The function main first calls the function check_args, which checks the command line parameters (line 877). If this check is successful, the program connects to the queue manager (line 881) and opens the input and output queue(s) or file (lines 894 and 903). Then the program calls the function copy_messages (line 913), which copies or moves the message(s) from the input to the output device. When the copying or moving has finished, the program closes the input and output devices (line 917) and disconnects from the queue manager (line 923).

```
858: int main(int argc, char **argv)
859: {

...

877:    function = check_args(argc, argv, input, output, QMName,
878:       &first_msg, &last_msg);
879:
880:    /* Connect to queue manager. */
881:    MQCONN(QMName,                    /* queue manager */
          &Hcon,                         /* connection handle */
          &CompCode,                     /* completion code */
          &CReason);                     /* reason code */

...

   /* Open the input device. */
   open_input_device(function, input, Hcon, &OpenSrcCode,
      &HobjSrc, &fp);

...

   /* Open the output device. */
   open_output_device(function, output, Hcon, &OpenDestCode,
      &HobjDest, &fp);

...

   /* Copy or move messages from the input device to the */
   /* output device. */
   copy_messages (function, first_msg, last_msg, Hcon, input,
      &HobjSrc, output, &HobjDest, fp);

   /* Close the open devices. */
close_devices (function, Hcon, OpenSrcCode, &HobjSrc, OpenDestCode,
      &HobjDest, fp);

   /* Disconnect from MQM if connected. */
   if (CReason != MQRC_ALREADY_CONNECTED )
   {
      MQDISC(&Hcon,                           /* connection handle */
             &CompCode,                       /* completion code */
             &CReason);                       /* reason code */

...

   }

   exit(0);
}
```

Function for checking the command line arguments

The function check_args checks the command line parameters and displays an error message if invalid or duplicate parameters are used. Check_args returns a number, which describes the function of the program. For example the value:

```
 FUNC_Q2Q | FUNC_MOVE
```

(bit wise OR) means, move messages from a queue to another queue. The messages in the source queue will be deleted.

Functions for opening and closing input and output devices

The functions open_input_device and open_output_device open the queue(s) or the file using the calls of MQOPEN (for a queue) or fopen (for a file), depending on the function of the program. If the input device is a file, it will be opened in read-only mode. If the output device is a file, it will be checked for whether it already exists. When it exists, the user will be asked whether the file should be overwritten or the messages should be appended to the existing file.

The function close_devices closes the queue(s) or the file by calling the functions MQCLOSE and fclose.

Function to copy or move messages

The central function, which does most of the work, is copy_messages. This function contains a loop in which it reads messages from a queue or file (lines 639 and 652) and writes it back (lines 678 and 690). If no message range has been specified (parameter first_msg is equal to 0), any message is read from the input device (queue or file) and written to the output device. When the input device is a queue, the messages may be removed from the queue – if option -d has been passed to backupQ on the command line. Otherwise the messages are just browsed from the queue.

When a message range is specified (parameter first_msg is greater than 0), only messages numbered between the parameters first_msg and last_msg are copied. The program stops reading after the last message, and exits (line 625).

```
589: static int copy_messages(int function, int first_msg, int last_msg,
590:    MQHCONN Hcon, char *input, MQHOBJ *HobjSrc, char *output,
591:    MQHOBJ *HobjDest, FILE *fp)
592: {
```

```
...

/* Loop until an error occurs. */
while (CompCode != MQCC_FAILED)
{
   /* Read when all messages have to be read (first_msg is 0) */
   /* or the last message to copy or move has been reached. */
   if ((first_msg == 0) || (pos_count <= last_msg))
   {

...

      /* Input device is a queue. */
      if ((function & FUNC_Q2Q) || (function & FUNC_Q2F))
      {
         MQGET(Hcon,                /* connection handle */
                *HobjSrc,           /* object handle */
               &md,                 /* message descriptor */
               &gmo,                /* get message options */
               buflen,              /* buffer length */
               buffer,              /* message buffer */
               &messlen,            /* message length */
               &CompCode,           /* completion code */
               &Reason);            /* reason code */
      }
      /* Input device is a file. */
      else
      {
         read_from_file(fp,         /* file handle */
               &md,                 /* message descriptor */
               buflen,              /* buffer length */
               buffer,              /* message buffer */
               &messlen,            /* message length */
               &CompCode,           /* completion code */
               &Reason);            /* reason code */
      }

    /* Write the read message again, when position counter is */
      /* greater than number of the first message. */
      if ((CompCode != MQCC_FAILED) && (pos_count >= first_msg))
      {

...

         /* Put each buffer to the message queues. */
         if (buflen > 0)
         {
            /* Output device is a queue. */
            if ((function & FUNC_Q2Q) || (function & FUNC_F2Q))
            {
```

```
678:                   MQPUT(Hcon,              /* connection handle */
679:                         *HobjDest,           /* object handle */
680:                         &md,               /* message descriptor */
681:                         &pmo,      /* default options (datagram) */
682:                         messlen,             /* buffer length */
683:                         buffer,              /* message buffer */
684:                         &CompCode,           /* completion code */
685:                         &Reason);            /* reason code */
686:                }
687:                /* Output device is a file. */
688:                else
689:                {
690:                   put_to_file(fp,            /* file handle */
691:                         &md,               /* message descriptor */
692:                         messlen,             /* buffer length */
693:                         buffer,              /* message buffer */
694:                         &CompCode,           /* completion code */
695:                         &Reason);            /* reason code */
696:                }

...

703:             }
704:             /* Satisfy end condition when empty line is read. */
705:             else
706:             {
707:                CompCode = MQCC_FAILED;
708:             }
709:          }
710:
711:          pos_count++;
712:       }

...

772:       }
773:    }
774:
775:    return(0);
776: }
```

Functions for file input and output

I created two functions, read_from_file and put_to_file, which work similarly to the functions MQGET and MQPUT, but read or write to or from a file. First the function read_from_file tries to read a line header (line 483). It compares the read data with the global constant FILE_MD_HEADER (line 487). If the strings are not equal, the file is interpreted as a manually-created test file. Then the function rewinds the file pointer and reads a complete line out of the file (lines 494 and 497). In lines 499 and 500 a trailing newline character is replaced by a NULL character.

If the strings that have been compared in line 487 are equal, the function branches to the else part (line 511). In this case the file is interpreted as a data file that was created previously by backupQ. The function now reads the characters up to the next colon. This string is interpreted as the length of the stored message descriptor and is written into the array len (lines 525 to 533). The following bytes up to the next colon are interpreted as the length of the message, and this length is also written into the array len in the same way.

Now the message descriptor and the message itself are read (lines 537 and 542). The length of the message is returned by copying it from the array len to the parameter messlen (line 539). In line 545, the trailing newline character is read – just to set the file pointer. When the read string contains the end mark of the file (line 549), the completion and reason codes are set to MQCC_FAILED and MQRC_NO_MSG_AVAILABLE, to satisfy the check in the calling function copy_messages (line 551 and 552).

The write function put_to_file is much easier than the read function. It first creates a string with a line mark and the sizes of the message descriptor and the message, separated by colons (line 568). Then it writes the message descriptor (line 571) and the message itself (line 574).

```
467: static void read_from_file(FILE *fp, MQMD *md, MQLONG buflen,
468:  MQBYTE *buffer, MQLONG *messlen, MQLONG *CompCode, MQLONG *Reason)
469: {

...

482:    /* Try to read a header string. */
483:    fread (header, 1, sizeof(FILE_MD_HEADER), fp);
484:    header[sizeof(FILE_MD_HEADER) - 1] = 0;
485:
486:    /* Look for message descriptor mark. */
487:    if (strncmp(header, FILE_MD_HEADER, sizeof(FILE_MD_HEADER) - 1)
!= 0)
```

```
      md_flag = FALSE;

 /* Message descriptor mark not found, file is manually created. */
   if (md_flag == FALSE)
   {
      /* Rewind the file pointer. */
      fseek (fp, -sizeof(FILE_MD_HEADER), SEEK_CUR);

      /* Read one line from the file. */
      if (fgets(buffer, buflen, fp) != NULL)
      {
         if (buffer[strlen(buffer) - 1] = ‘\n’)
            buffer[strlen(buffer) - 1] = ‘\0’;

         *messlen = strlen(buffer);
      }

...

   }
   /* File is created by this program (contains a message descriptor). */
   else
   {

...

  /* Read the beginning of the line (contains the lengths of the */
      /* message descriptor and the message itself, separated by colons). */
      for (idx = 0; idx < 2; idx++)
      {
         c = 0;
         buf[0] = 0;

         while (c != ‘:’)
         {
            c = getc(fp);

            if (c != ‘:’)
               sprintf (buf, "%s%c", buf, c);
         }

         len[idx] = atol(buf);
      }

      /* Read the message descriptor. */
      fread (md, 1, len[0], fp);

      *messlen = len[1];

541:        /* Read now the message. */
542:        fread (buffer, 1, *messlen, fp);
543:
544:        /* Get the newline character (to set the file pointer). */
545:        c = getc(fp);
546:     }
547:
548:     /* Check for an end tag. */
549: if (strncmp(buffer, FILE_MQ_END_TAG, strlen(FILE_MQ_END_TAG)) == 0)
550:     {
551:        *CompCode = MQCC_FAILED;
552:        *Reason = MQRC_NO_MSG_AVAILABLE;
553:     }
554: }

...

560: static void put_to_file(FILE *fp, MQMD *md, MQLONG messlen, MQBYTE
*buffer,
561:     MQLONG *CompCode, MQLONG *Reason)
562: {

...

567:     /* Write a mark and the lengths of message descriptor and
message. */
568:     num = fprintf(fp, "\n%s:%ld:%ld:", FILE_MD_HEADER, sizeof(MQMD),
messlen);
569:
570:     /* Write the message descriptor to the file. */
571:     num += fwrite(md, 1, sizeof(MQMD), fp);
572:
573:     /* Write the message itself to the file. */
574:     num += fwrite(buffer, 1, messlen, fp);

...

587: }
```

LISTING OF BACKUPQ.C

The full listing of the program backupQ is too long to be written here. The program code may be downloaded from Xephon's Web site at www.xephon.com/extras/backupQ.c. Parts of the listing are shown in the text above.

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

Java Message Service, WebSphere Application Server, and Message Driven Beans

This article looks at Java Message Service, WebSphere Application Server, and Message Driven Beans.

The good news is that Java Message Service (JMS) is now easier to install, administer, and use. It is designed to provide a robust asynchronous messaging model, which will deliver services for constructing high-performance, encapsulated, portable, and transactional applications.

WebSphere Application Server (WAS) 5.0 is Java 2 platform Enterprise Edition (J2EE) 1.3 compliant – which means that it comes with an integrated JMS provider. WAS 5.0 provides support for both the JMS point-to-point and publish/subscribe messaging models. WAS 5.0 also offers full support for Message Driven Beans (MDBs).

MDBs are part of the Enterprise Java Bean (EJB) 2.0 specification, and their role is to provide asynchronous messaging using base JMS functionality.

It was difficult to build messaging applications using J2EE before MDBs were introduced as part of J2EE 1.3. MDBs are good for application developers because they delegate the responsibility of providing infrastructure for transactions, security, and concurrently processing messages to the EJB container.

So let's take a look at each of these in detail.

If you want to access asynchronous messaging systems from Java applications you need an API to do it. JMS is a Sun Microsystems Java specification that defines just such an API. As a consequence, JMS provides an asynchronous messaging model with services for constructing encapsulated, portable, and transactional applications that are high performing.

Importantly, it is an integral part of the J2EE 1.3 specification. Remember that WebSphere Application Server (WAS) is now J2EE 1.3 compliant.

WAS 5.0 now includes an integrated JMS Provider with the product, called the WebSphere JMS Provider. This delivers the enhanced JMS integration with the application server.

Customers no longer need to purchase MQSeries 5.2 or WebSphere MQ 5.3 for this functionality as they did with WAS 4.0; and even then, it only delivered the back-end messaging support of the provider – the JMS implementation was obtained through a separate download/install of the corresponding MQ classes for Java and MQ classes for Java Message Service LPP. There was then a long period of setting everything up.

WAS 5.0 now includes:

- WebSphere MQ 5.3 – providing the back-end messaging server used to receive, store, and send asynchronous messages.
- WebSphere MQ Classes for Java and JMS 5.3 – providing Java classes and interfaces used to access the back-end messaging server through JMS.

These products are installed by default during the WAS 5.0 installation process and provide the underlying JMS Provider by WAS. All access is performed through the WAS user interfaces (eg the administrative console), so users don’t need to interact directly with these products.

Licensing restrictions mean that a full WebSphere MQ licence has to be purchased if applications don’t integrate with WAS 5.0 JMS applications. Otherwise, for example, as long as a WebSphere 5.0 JMS application receives and processes these messages, an RPG application could use these products to send messages to a WebSphere MQ queue using the native APIs of the product.

In the same way, an RPG application could use the native WebSphere MQ APIs to receive messages sent by a WebSphere 5.0 JMS application.

The JMS server provides an integrated JMS security service. This (not surprisingly) allows JMS resources to be secured. It is similar to the security service used with other types of J2EE component (such as servlets, JSPs, and EJBs) in WAS.

The JMS server can be found in the WAS 5.0 run-time. It interacts with the WebSphere JMS Provider and the application server run-time. It runs in the application server process/job, although if the WAS 5.0 instance is part of a Network Deployment cell, the JMS server runs as a separate process/ job and interacts with the application server process.

The JMS server is used by the application server to interact with the WebSphere JMS Provider to send/receive and publish/ subscribe JMS messages. For backward compatibility, JMS through WAS 5.0 continues to support the point-to-point messaging models.

JMS resources, such as connection factories, queues, topics, and message listeners, are administered by the WAS administration tools using the JMS server. There are now standard WAS administrative interfaces for this – the WAS administrative console and the WAS administrative scripting engine (wsadmin). This means that JMS resources can now be administered in the same way as other resources (JDBC drivers, data sources, etc) using these common interfaces.

Moving on to Message Driven Beans (MDBs), WAS 5.0 fully supports MDBs, which are part of the EJB 2.0 specification. They are a specialized form of session beans that wrap a JMS resource such as a queue or topic.

In essence, an MDB gets activated when a message arrives and listens to a message destination or a message endpoint. MDBs are anonymous in nature and cannot be directly invoked by a client. An MDB is invoked by sending a message to the destination or endpoint to which it is listening.

An MDB does not have interfaces like other types of EJB; it only has a bean-implementation class. MDBs implement two interfaces – one is an EJB interface and the other is a JMS interface. The MDB bean class has to implement the javax.ejb.MessageDrivenBean interface. It must also implement the message listener interface required by the messaging type that it supports. An MDB that supports JMS must implement the javax.jms.MessageListener interface.

MDBs provide asynchronous messaging using base JMS functionality. They are server-side components and are basically stateless session beans. They are used to process JMS messages, although they are capable of participating in global transactions.

Part of the MDB implementation is the message listener service. One or more listeners associated with a given MDB are controlled and monitored by a listener manager in WAS 5.0. As soon as an incoming JMS message arrives, it is passed on for processing. The listener then goes back to listening – there is no waiting.

MDBs are thread-safe and capable of receiving many messages from various applications and processing them simultaneously. MDBs are only accessible via asynchronous messages and can't be accessed via standard EJB methods such as the Java RMI/IIOP API.

Without WAS 5.0 there's no JMS. Without JMS there's no MDB. Without MDB there's no efficient processing of asynchronous messages.

Nick Nourse
Independent Consultant (UK)

DataPower Technology has announced the XI50 Integration Appliance, a networking device that makes XML and non-XML data usable for mainframes, Enterprise Service Buses (ESBs), and application integration.

The product supports a range of transport protocols, including MQ Series, and can perform translations between formats other than XML. XI50 can parse and transform arbitrary binary, flat text, and XML messages, including COBOL CopyBook, CICS, ISO ASN.1, and EDI.

For further information contact:
DataPower Technology, One Alewife Center, 4th Floor, Cambridge, MA 02140, USA.
Tel: (617) 864 0455.
URL: http://www.datapower.com/products/xi50.html.

* * *

Compuware has announced Version 3.1 of STROBE, which is designed to help users improve the efficiency of their applications.

STROBE 3.1 provides support for WebSphere Application Server, complementing its existing support for Java, enabling users to manage and improve the performance of Java and WebSphere applications. It also provides information on how Java and WebSphere applications interact with CICS, DB2, and other z/OS facilities.

For further information contact:
Compuware, One Campus Martius, Detroit, MI 48226, USA.
Tel: (313) 227 7300.
URL: http://www.compuware.com/products/strobe/default.htm.

* * *

IBM has announced CICS Interdependency Analyzer for z/OS Version 1.3, which is used with CICS Transaction Server on a mainframe to identify the resources used by CICS transactions and the relationships between them. The product also reports on WebSphere MQ, DB2, and IMS resources that are used by CICS. The main resources that are identified include those associated with transactions, programs, BMS maps, files, temporary storage queues, transient data queues, 3270 Bridge facility, Web Services, CorbaServer, and Enterprise JavaBeans.

For further information contact your local IBM representative.
URL: www.ibm.com/software/htp/cics/products/interdepanalyzer.

* * *

IBM has announced WebSphere Business Integration Modeler Version 5, which can help companies establish a more detailed map of the flow of business processes across their IT systems. This helps to identify slowdowns, and respond faster to customer demand and changing market conditions.

WebSphere Business Integration Modeler provides support for WebSphere Business Integration Server Foundation, WebSphere MQ message queueing software, and Rational Rose XDE development tools. Customers can work with existing content based on standards like XML, and extend it using WebSphere Business Integration Modeler's simulation and modeling capabilities.

For further information contact your local IBM representative.
URL: http://www-306.ibm.com/software/integration/wbimodeler/library/quicktour.html.