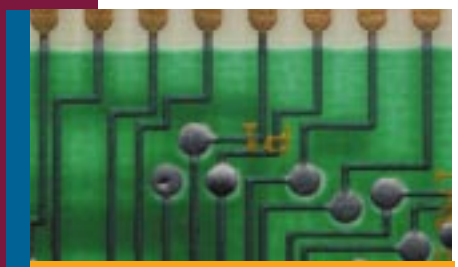


Using MQSeries for Synchronous Processing: Part II

BY RICHARD TSUJIMOTO

This concluding article examines a high-level design that uses MQSeries to replace FTP as an alternative method of synchronous data transmission.

PART I (***Technical Support***, April 1998) discussed the basic concepts of MQSeries synchronous messaging along with the resources that are required to make it possible. This concluding article describes a high-level design that was used to create a file transfer application that is also a classic example of synchronous processing. Additionally, this article discusses the events that led to the development of this application, the design requirements, testing results and post-implementation analysis.

BACKGROUND

One of my clients was deeply enmeshed in migrating legacy CICS applications from MVS/ESA 4.3 on an IBM 9672 R32 to AIX 4.1.4 on an RS/6000 SP Model 9076, using Oracle as the basis for the new applications. As with any migration effort, data files are frequently passed back and forth between the old and new platforms. The communication protocol between the two environments was TCP/IP over a 10MB ethernet link and FTP was used for transferring data. An automated job scheduler was installed on AIX to manage batch processing, including jobs using FTP.

A key feature of any job scheduler is the ability to release down-stream jobs based on the completion status of preceding jobs. However, several unfortunate incidents occurred with FTP on AIX that lead to the improper release of dependent jobs. After some research, it became apparent that FTP on AIX (or most UNIX variants for that matter) can return a successful return code even if something went wrong. In this particular case, an invalid password was used by an AIX FTP job when it attempted to transfer a file to MVS. FTP on AIX issued messages indicating a problem had been encountered, but it nonetheless returned a successful return code, without transferring any data to MVS. The converse was not a problem because FTP on MVS supports an optional parameter, e.g., PARM=(EXIT, that results in a non-zero return code when any error occurs. In effect, return code values from FTP on AIX were unreliable and an alternative method for transferring files was required. MQSeries had been installed on AIX and MVS in early 1997, but it was not being used. The client wanted to see if MQSeries could serve as a replacement for FTP.

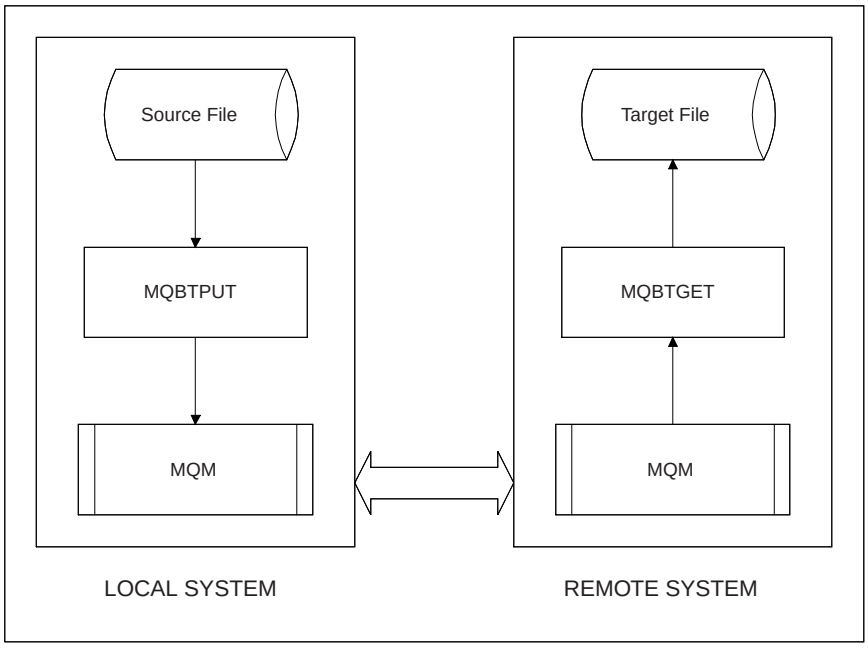
FUNCTIONAL REQUIREMENTS

The requirements for the MQSeries-based application, which will be referred to as MQM-FTP, were as follows:

- It must have a similar "look and feel" as FTP: The primary reasoning for this requirement was to minimize the learning curve for users and the impact to existing batch jobs. Hence, the method of invocation must be similar, i.e., FTP can be invoked as a line command that accepts input from switches on the command line or from a file. This also implies that the file transfer operations must be treated as an atomic operation. An entire file must be transferred to be considered a success. Otherwise, the request is a failure and the entire operation must be repeated.

- It must support a reasonable subset of FTP services: FTP supports numerous services and features, most of which the client did not require. In short, the client wanted to be able to simply

Figure 1: MQM-FTP Design Overview



transfer files between platforms. The target files would either be overwritten or appended to, or newly created.

- It must deliver data in the same sequence that it had been sent: For obvious reasons, the order of data must be preserved and steps must be take to ensure MQSeries does not change it when it delivers each record (or message) to the target destination.

- It must be able to deliver the data in a timely manner: My client accepted the fact that a message queuing system has inherent delays, such as delays due to queuing. However, the final product was still expected to be able to deliver the data in a reasonable amount of time.

- Lastly, it must be able to return reliable return codes: This was one of the primary concerns about the development of this application.

DESIGN OVERVIEW

The objective of MQM-FTP is quite simple. Basically, its purpose is to transmit the contents of a file from one platform to another (see Figure 1). The source and target sites could be similar platforms, for example, such as MVS, or they could be two dissimilar ones, such as MVS and AIX. The modules that are used in this design are as follows:

- **MQBTPUT:** This is an MQSeries application that transmits a file to another platform. This program can be invoked from both MVS/ESA and AIX.

- **MQBTGET:** This is the counterpart to MQBTPUT that will receive messages transmitted by MQBTPUT and write the messages to a destination file as records. It, too, will run on both MVS/ESA and AIX.

- **MQBTMON:** Unlike other platforms, MQSeries does not provide a batch trigger monitor for MVS/ESA. Customers can either develop their own, obtain one from another user or try to purchase one. In this particular case, a customized batch trigger monitor has been developed that is generic in design and not specific to MQM-FTP. It just so happens that MQM-FTP is the first batch MQSeries application that is controlled by MQBTMON.

TEMPORARY DYNAMIC QUEUE

This is a special type of queue that was not discussed in Part I but plays an important role in the design of MQM-FTP. Basically, a temporary dynamic queue can be created during the execution of an MQSeries application and could optionally be assigned a unique name. It has a lifetime that matches the application that initiated its creation, disappearing once the queue has been explicitly closed by the application or when the program terminates. Lastly, only non-persistent messages can be written to it.

A CLOSER LOOK UNDER THE HOOD

The following sections provide a general overview of the logic flow in MQM-FTP. There are three basic phases, the first being the most complicated. The first phase, initial "handshake," performs the setup needed to ensure the readiness of both the sending and receiving sides. The data transfer phase is basically a repetitive sequence, or loop, of messages being sent and received. Finally, the last phase confirms that the number of messages sent matches those received.

Initial "Handshake"

This phase presents the steps required to ensure that both the sender and receiver are ready for the transmission of messages (see Figure 2). For illustration purposes, the following steps describe a message being sent from MVS to MVS, but they are applicable to any other platform as well.

1. MQBTPUT processes the control card input, validating its syntax and content. It then obtains a temporary dynamic queue that has a prefix of MQBTPUT. This queue will be used for receiving control information from MQBTGET. MQBTPUT then creates a control message that contains the name of the temporary dynamic queue, the name of the target file, whether the target file will be overwritten or appended to, and so forth.

2. MQBTPUT then transmits the control message by issuing an MQPUT1 request to SY2.FTP, which is a queue alias name that is associated with a remote queue definition. The remote queue definition, in turn, describes other relationships. First, it defines the transmission queue to be used to transmit the message. Second, it defines the name of the destination queue owned by CSQ2. Third, it defines the channel to be used when the message is sent to CSQ2. MQBTPUT then issues an MQGET against the temporary dynamic queue, waiting for MQBTGET's response message.

3. The MQ channel mover sends the message over the channel using, in

Figure 2: Initial Handshake

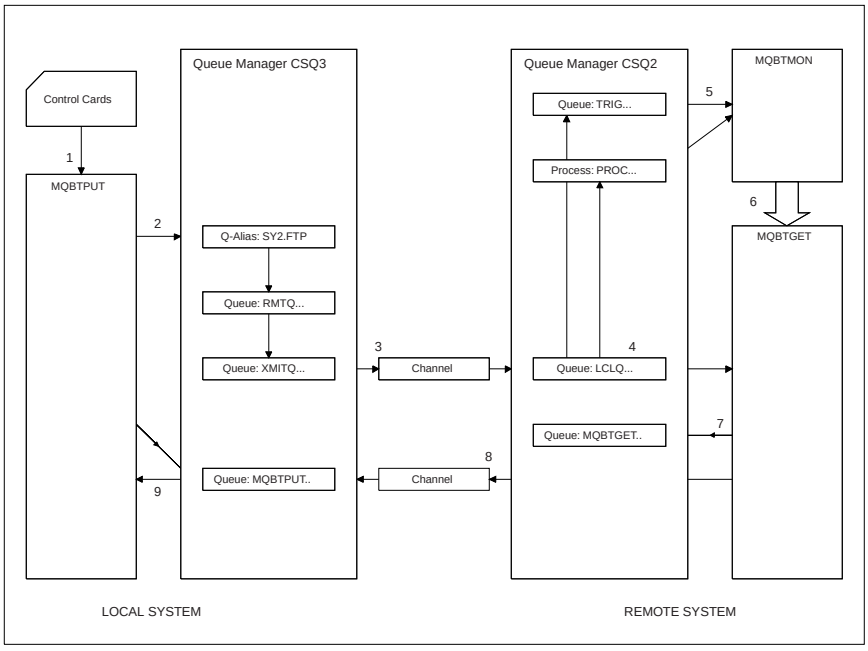
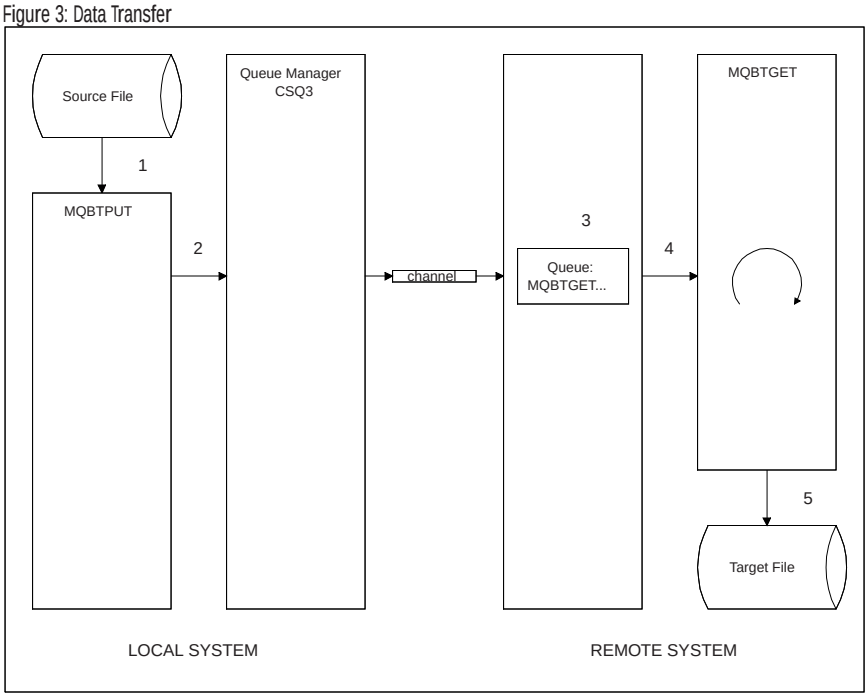


Figure 3: Data Transfer



this case, TCP/IP as the communication protocol. The control message is received by CSQ2 and placed on the *FTP control message queue*, which is called LCLQ.FTP.SY2.01. This is a special queue that is used to receive control messages that describe an MQM-FTP request.

4. The *FTP control message queue* is defined as having a trigger. In effect, whenever it receives a message, a special message called a trigger message is put on an initiation queue that is associated with it. In this case, the initiation queue is called TRIG.SY2.BATCH.01.

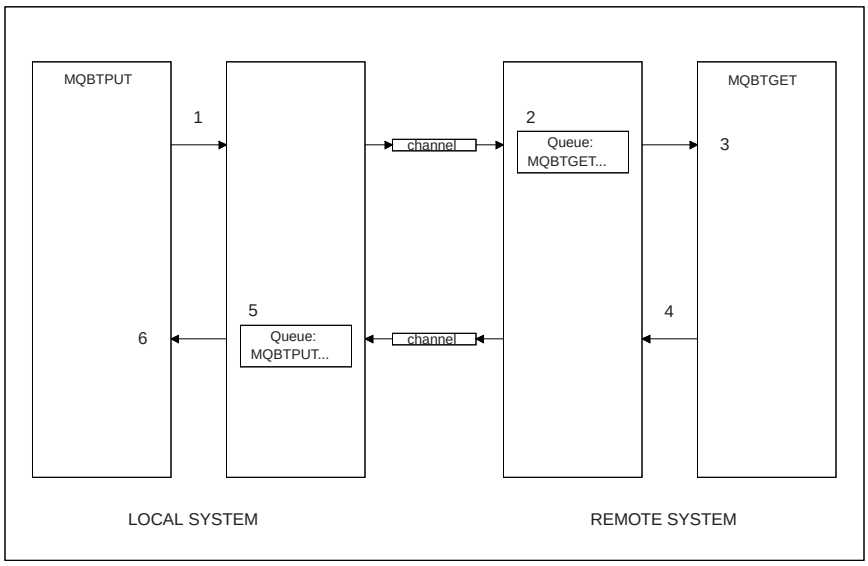
5. MQBTMON is a long-running task that waits for a trigger message to arrive on TRIG.SY2.BATCH.01. The message that arrives could be a trigger message or a special message that contains a command, e.g., QUIT. If the message is a trigger message, MQBTMON accesses the application identifier field in the Process Definition to determine which batch job should be submitted to process the FTP request that just arrived. The Process Definition called PROC.FTP.SY2.01 is associated with the *FTP control message queue*. The data in the application identifier field has been set with the string DefaultFTPappl, which is an arbitrary string that is used to identify MQM-FTP requests.

Using temporary dynamic queues made it possible to design MQM-FTP as a generic utility that required few, if any, additional MQSeries resource changes once the initial definitions were in place.

6. MQBTMON will start a job that invokes MQBTGET, the counterpart to MQBTPUT. The batch trigger monitor will modify the JCL for the job, inserting the name of the *FTP control message queue* and the name of the local Queue Manager, or CSQ2 in this case, in the OS parameter field on the JCL EXEC PGM=MQBTGET statement. MQBTGET uses this information so that it knows which Queue Manager to connect to and which queue contains the control message. MQBTGET then fetches the control message, obtaining the information necessary to gain access to the target file and MQBTPUT's reply-to-queue name.

7. MQBTGET will open the target file (creating it first if the proper allocation information is provided in the control message) and then it will obtain a temporary dynamic queue that will be used by MQBTPUT as a target queue during message transmission. The name of the temporary dynamic queue will have MQBTGET as a prefix. Once MQBTGET is finally ready to start receiving messages, it will create a response message that includes

Figure 4: Final Confirmation



the name of its dynamic temporary queue. It will then send the response message to MQBTPUT's temporary dynamic queue and issue an MQGET on its own temporary dynamic queue, waiting for the data transfer phase to begin.

8. The response message is sent back to MQBTPUT via its receiver channel where, eventually, the message is placed on MQBTPUT's temporary dynamic queue.

9. MQBTPUT's outstanding request for the next message is satisfied. It then fetches the message, verifying that the response acknowledges MQBTGET's readiness to start receiving messages. At this point, the "handshake" between the two components has completed, and the data transfer phase is ready to begin.

Data Transfer

This phase of MQM-FTP processing is fairly straightforward (see Figure 3). It is an iterative process where messages are being read from a file, transmitted and received until an end-of-file (EOF) is encountered processing the source file.

1. MQBTPUT reads the next record from the source file and issues an MQPUT request, initiating the transmission of the data. As soon as the MQPUT request is completed, MQBTPUT will read the next record. Note that the completion of the MQPUT request does not mean the message has been received by MQBTGET. Rather, a successful MQPUT request indicates that MQSeries has accepted the message and placed it on the associated transmission queue.

2. The message is sent to the remote system using the same MQSeries resources as outlined earlier (see Figure 2).

3. The data is eventually stored on MQBTGET's temporary dynamic queue.

4. MQBTGET's outstanding MQGET request is satisfied. It fetches the next message, making sure that it is data and not the final confirmation request. Optionally, the data is converted to an appropriate coded character set during the MQGET processing.

5. MQBTGET writes the message as a record to the target file.

Final Confirmation

This phase describes the steps that complete the MQM-FTP processing. See Figure 4. MQBTPUT will not know until the very end if the data transfer was successful or not.

1. When MQBTPUT encounters an EOF while reading the source file, it then prepares a Request-Type message that contains the number of records sent. MQBTPUT sends the message to the remote system. MQBTPUT then issues a final read against its temporary dynamic queue, waiting for MQBTGET's reply.

2. The message is enqueued, transmitted and eventually gets placed on MQBTGET's temporary dynamic queue.

3. MQBTGET fetches the message as part of the normal data transfer phase. It then determines that it is a Request-Type message that contains a record count. MQBTGET compares that value with its own count. If the record counts match, MQBTGET closes the target file and prepares a positive acknowledgment reply.

4. MQBTGET issues an MQPUT for the final reply, sending the message to MQBTPUT's reply-to-queue. MQBTGET then prepares to terminate, closing its temporary dynamic queue and disconnecting from the local Queue Manager.

5. The final reply message is eventually placed on MQBTPUT's temporary dynamic queue.

6. MQBTPUT reads the reply message, and prepares for termination by closing its temporary dynamic queue and disconnecting from the local Queue Manager.

TEST RESULTS AND EXPERIENCES

The process of making MQM-FTP work and then having it perform to meet a certain level of acceptance was both frustrating and exciting. Along the way, unexpected obstacles were encountered that caused delays and resulted in additional coding and testing.

ERROR HANDLING

Even though MQM-FTP was based upon a synchronous messaging model, it still felt the effects of messaging or queuing when an error occurred. The most obvious case occurred when messages could no longer be delivered to the destination queue because the destination queue had disappeared. Since temporary dynamic queues were used extensively in the design of MQM-FTP, they could disappear in an error

situation. In this case, the remaining messages ended up on the dead letter queue. An MQSeries dead letter queue is similar to the Post Office's dead letter office, except that the customer has the option of trying to remedy the problem. Hence, it became necessary to develop an application to manage the dead letter queue that basically retrieved and discarded any MQM-FTP message.

Another error situation that required special processing occurred when MQBTGET encountered a problem that made it impossible to continue — the target file ran out of disk space. MQBTGET handled this problem by basically *draining* the remaining inbound messages before it terminated.

The final problem occurred when the application became blocked or stalled due to a shortage of processing power or resource contention. MQM-FTP addressed this problem by giving the user the option of specifying a time-out value as a control card parameter. This value basically controlled the wait time for any MQGET request, whether it was issued from MQBTGET or MQBTPUT.

DATA INTEGRITY

One of the primary functional requirements was to be able to retrieve the messages in the same order they were sent. *The MQSeries for MVS/ESA Distributed Queuing Guide* explicitly states the conditions that must be met to satisfy this requirement. However, even if these conditions were strictly adhered to, in certain error situations it may not be possible to retrieve the messages sequentially. For example, if the destination queue somehow had its PUT attribute set to DISABLED, MQSeries would reroute inbound messages to the dead letter queue. Since the file transfer process was to be regarded as a single atomic operation, e.g., the entire file must be resent in the case of an error, a final confirmation message with the expected totals was sent as a safeguard.

MVS/JES

MQBTGET encountered scheduling problems on MVS because it had to contend for a finite set of initiators. This led to erratic response times. Allocating a dedicated class of initiators for MQBTGET solved the problem.

Occasionally, MQBTGET would run on the wrong LPAR. Since the LPARs were connected via a shared spool, a job could run on the wrong system unless the appropriate JES control card was specified. The solution was to ensure that MQBTGET would have a JES control card that forced it to run on the proper LPAR.

> Even though the strength of MQSeries really lies in its asynchronous messaging model, IBM is beginning to realize the importance of enhancing the synchronous model as well.

PERFORMANCE

The requirement that the file should be delivered in a timely manner is fairly ambiguous as far as requirements go. Fortunately, my client understood that messaging, by design, has some built-in delays. Nevertheless, every effort was made to optimize the performance of MQM-FTP. Since it was a given that MQM-FTP was to mimic FTP as much as possible, any failure would require the file to be resent in its entirety, just as it would using FTP. This assumption made the design a bit simpler. By being able to discard all of the data in the case of an error, the messages could be regarded as non-persistent and not within a unit of work, e.g., no syncpoints were required. These two factors enabled MQM-FTP to perform faster. First, MQSeries will try to keep non-persistent messages in memory and only write them to disk if memory becomes exhausted. Second, by not having to send batches of messages as units of work, the overhead associated with syncpoint processing (e.g., writes to the log) is eliminated.

But, these two factors by themselves were not sufficient to make MQM-FTP perform as expected. The initial timings based on these factors and using the default BATCHSZ of 50 were wholly unacceptable. The time it took to transfer 100,000 100-byte records or a 10MB file was measured in minutes. The BATCHSZ is a parameter that can be specified on sender, receiver, server, and requestor type channels and it represents the maximum number of messages that can be sent through a channel before a checkpoint is taken. When a checkpoint is taken, it typically involves taking a *snapshot* of the system's status and writing it to the active log. Taking a checkpoint is expensive, and if it occurs too frequently, it can have a detrimental affect on performance. Based upon these factors, it became apparent that two other changes may be needed. First, adjusting BATCHSZ may improve performance. Second, the MQM-FTP must block individual messages (or records in this case) into a larger message before sending it. Since it was not known what value should be use for blocking purposes, it was decided to make this a new parameter in the control card. This approach allowed us to easily change it and to determine an optimal value.

A series of volume tests were conducted where 10MB, 30MB, 60MB, and 90MB files were transferred from MVS-to-MVS, AIX-to-AIX, AIX-to-MVS and MVS-to-AIX. Interestingly enough, the results of each test showed that favorable elapsed times were achieved when the maximum message size ranged from 55,055 to 214,055 bytes. However, the times themselves were not consistent from test to test. For example, the ratio of MQM-FTP's response times to FTP's was at times longer for a 60MB file than for a 90MB file. However, given that the testing conditions were less than ideal, the anomalies could be attributed to the contention on the ethernet link with other traffic. The results were, nevertheless, quite impressive. In the case where a 10MB file was transferred, MQM-FTP's response time was very close to FTP's. And, in the case where a 90MB file was transmitted, MQM-FTP's response time was 2:24 vs. 2:16. From a statistical perspective, that ratio may seem horrific, but from a user's view point, it represents data being delivered in a timely manner.

DEVELOPING THE MVS BATCH TRIGGER MONITOR

Since IBM does not provide a batch trigger monitor, we decided to develop our own. I downloaded SupportPac MA12 from IBM's website and used it as a model. We decided to make the batch trigger monitor as generic as possible so that it could support an arbitrary number of triggered applications based upon the literal found in the Process Definition's application identifier field. Since the batch trigger monitor was a long running job, it had to be able to accept external commands to

update installation-specific data, e.g., accept and recognize a new application identifier string. The effort required to develop the batch trigger monitor proved to be fairly extensive and certainly beyond the scope of this article.

SUMMARY

The final product met all of the functional requirements and certainly performed better than expected. Using temporary dynamic queues made it possible to design MQM-FTP as a generic utility that required few, if any, additional MQSeries resource changes once the initial definitions were in place. MQM-FTP proved it was possible to use MQSeries for transmitting a significant amount of data in a (near) synchronous manner. Even though the strength of MQSeries really lies in its asynchronous messaging model, IBM is beginning to realize the importance of enhancing the synchronous model as well. IBM introduced Message Groups in MQSeries V5, which provides a means for maintaining ordering of messages. Unfortunately, this support has not yet been extended to MQSeries for MVS but it may be just a matter of time.

ACKNOWLEDGMENTS

I would like to thank Howard Simonson, IBM Global Services, who developed the AIX-based MQM-FTP programs and helped me understand and appreciate the potential power of MQSeries.

REFERENCES

IBM MQSeries: An Introduction to Messaging and Queuing - GC33-0805

IBM MQSeries: Application Programming Guide - SC33-0807

IBM MQSeries Application Programming Reference - SC33-1673

MQSeries for MVS/ESA Distributed Queue Management Guide - SC33-0806

Lewis, Rhys, Burnie Blakeley, Harry Harris, 1995, *Messaging & Queuing Using the MQI*, McGraw-Hill, New York

NaSPA member Richard Tsujimoto is an independent consultant specializing in CICS, MVS and MQSeries. He has also designed and developed software for UNIX and OS/2.