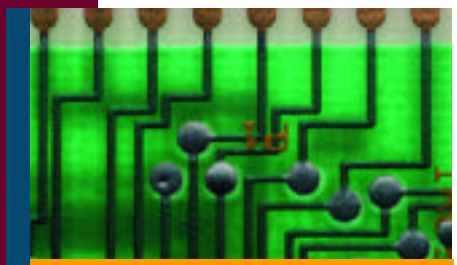


BY RICHARD TSUJIMOTO

Using MQSeries for Synchronous Processing: Part I

This article, the first in a two-part series, explores the capabilities and limitations of MQSeries for the development of synchronous messaging applications.

THE MQSeries middleware product from IBM introduces powerful messaging capabilities on a wide range of platforms, from the largest ES/9000 class processor to a desktop personal computer, running MVS, OS/390, OS/400, a number of different flavors of UNIX, Microsoft Windows, and OS/2. MQSeries also draws its popularity from other characteristics such as its application programming interface (API), which is consistent across dissimilar platforms, its support for a number of widely used communication protocols, such as TCP/IP and SNA, and its assured message delivery. But, the main strength of MQSeries is that,through its messaging capabilities,it allows applications to run independent of each other, on different platforms, at different times and speeds.

MQSeries Messaging applications are time-independent; that is, communication is conducted by sending and receiving messages to and from queues, as opposed to having a direct dialogue between each other. In other words, MQSeries is designed primarily for asynchronous message processing. But, in spite of this design characteristic, MQSeries does provide facilities that could be used to create synchronous — or at least *nearly* synchronous — messaging applications.

Given the existence of these facilities, this article will try to explore the capabilities and limitations of MQSeries for the development of synchronous messaging applications. As misguided or heretical as it may seem, it may prove enlightening to see how MQSeries performs when it is used for purposes other than those it was designed for.

ASYNCHRONOUS AND SYNCHRONOUS MESSAGING

A simple definition of asynchronous messaging is "a method for communication between programs that allows the requesting program to proceed with its own processing without having to wait for a reply to its request." In an MQSeries application, the requesting program would send messages to a queue and continue its own processing, as shown in Figure 1. Once the requesting application places its messages on the queue, its job is essentially done. A common analogy is a telephone caller who cannot

Figure 1: Asynchronous Messaging

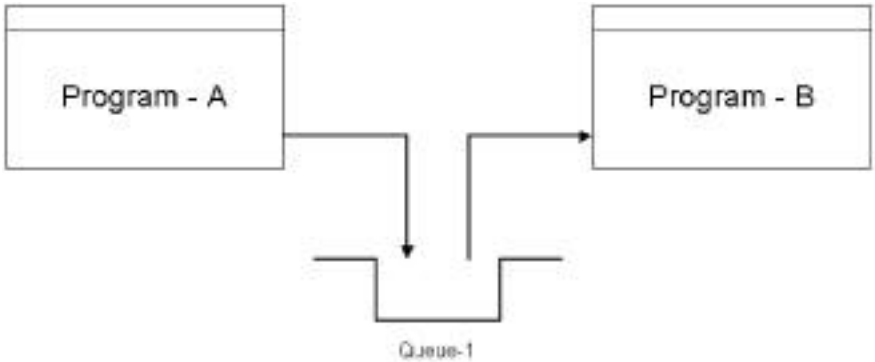


Figure 2: Synchronous Messaging

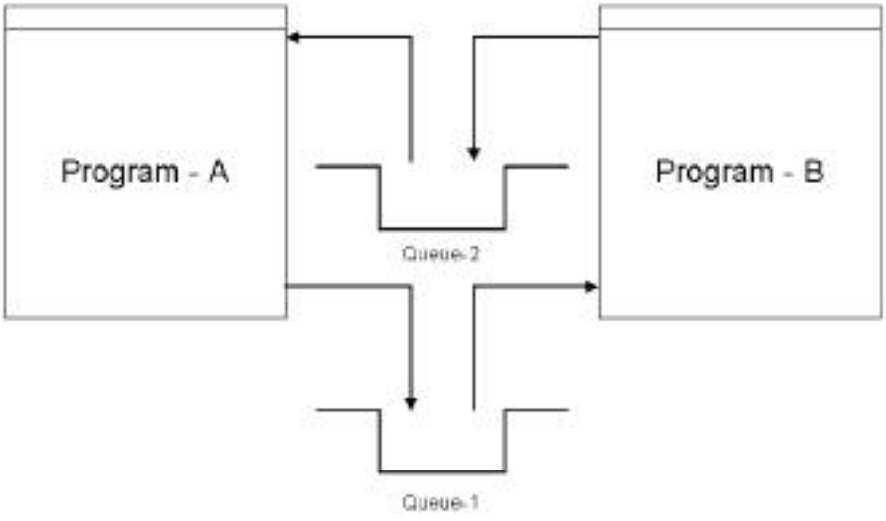
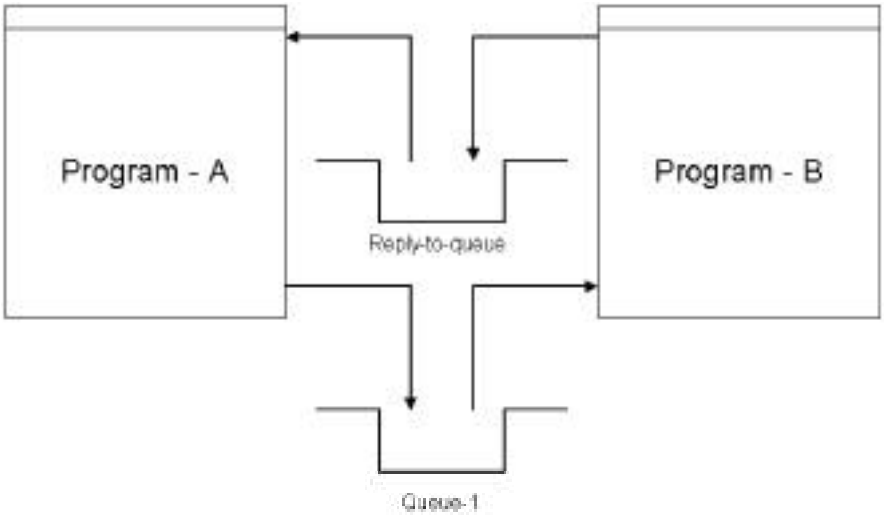


Figure 3: Response Message Put to Reply-to-Queue



As misguided or heretical as it may seem, it may prove enlightening to see how MQSeries performs when it is used for purposes other than those it was designed for.

reach the other party and is forced to leave a message on an answering machine. The receiver would then, at some point in the future, replay the answering machine and perform whatever actions the message required.

Asynchronous messaging offers several benefits. First, time-independent applications can proceed at their own pace, which may be significant if there is a large disparity in processor speeds. Second, this communication method allows for overlap and/or parallel processing so that a number of receiving applications can process messages at the same time. In this particular case, the requestor would send messages to multiple target queues, and a separate program would service each message. Third, the requestor can review replies concerning the outcome of the transmissions at some point in the future rather than having to wait for an initial connect to be established.

Conversely, synchronous messaging is time-dependent, is serialized and requires the requestor to establish a connection with the receiving application, as shown in Figure 2. In effect,the requestor must be aware of the name of its partner and to some degree, where that partner resides. Again, in the telephone analogy the caller establishes a connection and is involved in a conversation where each side must wait for the other person to finish a sentence before interjecting a thought. Though the occasional case where both sides talk at the same time may seem normal to some of us, in the digital world this is usually indicative of a malfunction.

Even though this method of communication may be more familiar and natural, it has several disadvantages. First, it forces the parties to communicate in a serial, dedicated manner. If a problem arises that delays one side, the other side is penalized as well. Second, this mode of communication requires the allocation of resources such as a telephone line that are dedicated and cannot be shared. Third, the management of programs becomes more difficult since synchronous programs need to know the name of the receiving program and its location. Moving programs from one location to another may entail changes to both the sending and receiving applications.

Another important difference between asynchronous and synchronous messaging is the message type each method employs. In essence, the message types characterize the functionality required to perform each type of messaging.

MQSERIES MESSAGE TYPES

MQSeries defines five types of messages and allows users to create application-specific message types as well. The MQSeries defined message types are as follows:

- Request
- Reply
- Datagram
- Report
- Trigger

Request messages are typically sent to a server that provides some service such as a data base query, and as the name implies, these messages expect a reply.

Reply messages are simply responses to *request* messages.

Datagram messages are sent without any expectation of a reply. A typical application of this message type is the arrival/departure information display at an airport.

Report messages are normally used to return information regarding an error condition or the occurrence of a particular event. For example, if the destination queue

at a remote machine is full, a *report* message could be generated and sent back to the requesting application.

On the surface, it may seem that a *report* message provides the same service as a *reply* message. But, the primary difference is that a report message is generated on an exception basis or in response to an event, whereas a reply is always created in response to a *request* message. Another difference is that applications control the relationship of a *request-reply* message pair and the queue manager itself does not impose any rules on its usage. But, in the case of *report* messages, applications and the queue manager itself can generate report messages. The report messages generated by the queue manager are used to facilitate management of expired messages, to confirm that a message has been put to its destination queue, and to indicate that an application actually reads the message from the queue. The latter two are known as *confirmation of arrival* (COA) and *confirmation of delivery* (COD). However, a *request-reply* message pair and a *report* message are similar in one respect. There must be sufficient information passed with the sending message to allow the *reply* or *report* message to be returned to a known destination, as shown in Figure 3. This destination is referred to as the *reply-to-queue*.

Trigger messages represent a special message type that is always generated by the queue manager itself. They form the basis of a mechanism used for automatically starting applications when message are put into a queue. This effect is referred to as *triggering*. In brief, the queue manager generates a trigger message when it determines that certain conditions are met. The *trigger* messages are used to inform a special application, known as a *trigger monitor,* that a message has arrived. The *trigger monitor* retrieves the *trigger* message and initiates the application based on the information in the message and its associated process definition. A process definition is an MQSeries object that provides information that can be used to initiate a particular application, such as the name of a CICS transaction.

MQSeries also supports the creation of application-specific message types that provide the designer with flexibility beyond the use of the defined message types. One possible use of application-specific message types would be the generation of a report message if a customer's balance exceeded a certain amount, thus identifying him/her as a potential customer for other investments, e.g., stocks.

Given the available message types, the *request-reply* message pair, the COA, COD and possibly application-specific message types all represent potential means for creating synchronous messaging applications. Each message type offers a different level of synchronization. The COA only reports the arrival of messages at the destination queue, while the COD goes one step further, reporting when an application actually retrieves the message. However, in both cases, the receiving application may fail or encounter a problem while processing the message itself, giving the sending application a potentially false sense of completeness. The *request-reply* message pair is the only mechanism that provides end-to-end serialization. Application-specific message types would, for the most part, result in some variation of the *request-reply* message pair model.

STAY TUNED

This article identified the MQSeries facilities that could be utilized in the creation of a synchronous messaging application. Having found those, the next logical step would be to design an MQSeries application that exploits them. The concluding article will present the design of a file transfer application as well as a discussion of its actual implementation and performance.

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 ts

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.