MQSeries

GC33-0805-01

# An Introduction to Messaging and Queuing

IBM

MQSeries

GC33-0805-01

# An Introduction to Messaging and Queuing

**Note!**

Before using this information and the products it supports, be sure to read the general information under "Notices" on page v.

**Second Edition (June 1995)**

This edition applies to the IBM MQSeries announcement of November 1994.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the addresses given below.

At the back of this publication is a page titled "Sending your comments to IBM." If you want to make comments, but the methods described are not available to you, please address your comments to:

IBM United Kingdom Laboratories Limited, Information Development,
Mail Point 095, Hursley Park, Winchester, Hampshire, England, SO21 2JN.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

# Contents



# Figures

contents

# Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates.

Any reference to an IBM licensed program or other IBM product in this publication is not intended to state or imply that only IBM's program or other product may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, 500 Columbus Avenue, Thornwood, New York 10594, U.S.A.

## Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

| | | | |
|---|---|---|---|
| AIX | CICS/MVS | IBM | OS/400 |
| AIX/6000 | CICS/VSE | MQSeries | RISC System/6000 |
| AS/400 | DB2 | MVS/ESA | System/390 |
| CICS | ES/9000 | OS/2 | |

The following terms are trademarks of Digital Equipment Corporation:

| | | |
|---|---|---|
| Digital | VAX | VMS |

The following terms are trademarks of Sun Microsystems, Inc.:

| | | |
|---|---|---|
| Sun | SunOS | Solaris |

The following terms are trademarks of other companies:

| | |
|---|---|
| AT&T | American Telephone and Telegraph Corporation |
| HP-UX | Hewlett-Packard Company |
| SCO | Santa Cruz Operation, Inc |
| Tandem | Tandem Computers Incorporated |
| UNIX | X/Open Company, Limited |
| UnixWare | Novell, Inc |
| Windows | Microsoft Corporation |

# Introducing Messaging and Queuing

***IBM MQSeries: a family of products for cross-network communication***

This book introduces the IBM MQSeries products, and the programming style—*Messaging and Queuing*—that underlies them. Briefly, the MQSeries products enable programs to talk to each other across a network of unlike components—processors, operating systems, subsystems, and communication protocols—using a simple and consistent application programming interface.

Why "Messaging and Queuing"?

*Messaging:* because programs communicate by sending each other data in messages rather than by calling each other directly.

*Queuing:* because the messages are placed on queues in storage, so that programs can run independently of each other, at different speeds and times, in different locations, and without having a logical connection between them.

***What's in this book***

On pages 1 and 2 there's a brief discussion of the nature of the problem addressed by the MQSeries products.

On pages 3 through 12 you can find a description of how Messaging and Queuing works.

Some of the key benefits of Messaging and Queuing are explored on pages 13 through 17.

Examples of some of the ways in which the MQSeries products can be used begin on page 19.

A small amount of technical detail about messages and message queues is provided beginning on page 29.

***Who this book is for***

This book is for anyone who is new to Messaging and Queuing.

# Meeting the interoperability challenge

***Business applications are made up of related programs***

It's commonplace for a business application to be designed as a group of related programs, each of which handles a single, well-defined component of the whole. Often, the programs that make up a business application run in a single environment (such as the OS/2 environment) on single or multiple processors. And sometimes they run in multiple, unlike environments.

***Related programs are distributed around the network to make better use of resources...***

Many businesses go a step further and distribute programs around the data-processing network, rather than run them all on one processor. For example, a single application could be distributed between an AIX/6000 environment on a RISC System/6000 processor and an OS/400 environment on an AS/400 processor.

There are many advantages in this approach, most of which are related to making better use of resources:

It's often a good idea to put a program near the data it's processing, so that network traffic is kept to a minimum. ("*My database is in Washington, so if I put my customer accounts program in Miami, my network is going to be kept busy just moving data between the two.*")

Load balancing—rescheduling and relocating the workload to complete it as efficiently as possible—is another reason for distributing an application. ("*My London branch is running at capacity, but I've got an underused machine in Rome. Why can't I just move some of the work to Rome?*")

Sometimes "rightsizing"—moving an application from one large machine to several smaller machines—is the trigger. ("*We've installed a midrange processor in each of our branches, so we need to share the workload among them.*")

***...but heterogeneous networks make distribution difficult***

Of course, when you distribute a single application, whether to unlike environments on a single processor or to different nodes of a network, you must have a way of getting the parts of the application to communicate with each other. This can be challenging enough when the components of the network are from a single vendor, when there are no variations in the operating systems you're using, when programs are written in a single language, and when there's a single communication protocol. How much more challenging the problem becomes when the network components are from a variety of vendors, when different operating systems are in use at different nodes of the network, when the programs you're trying to connect are written in different languages, and when multiple communication protocols are being used. And yet, the *interoperability* challenge—the challenge of getting programs to communicate across unlike environments—could be the one facing you today.

## meeting the interoperability challenge

***The composition of a network is likely to remain fluid and unpredictable...***

So how do networks become so diverse in the first place? Perhaps, for your company, buying equipment and services from multiple vendors is simply a matter of policy. Or perhaps you have incompatible computer systems after changes to your business: if your company has merged with or acquired another, if it has relocated or restructured in some way, the chances are that you've gathered an assortment of network components in the process. Or perhaps you need to connect your applications with those of your business partners—your suppliers, your distributors, even your own customers—so that business procedures can be integrated and data can be shared. As the boundaries between businesses themselves are becoming less distinct, the need to connect computer systems across geographic and organizational boundaries is growing.

***...and some resource constraints are appearing***

As more and more businesses attempt to associate related programs from unlike environments, some constraints are emerging. In particular:

- The time-dependent relationship between two communicating programs can impose artificial requirements: typically, one program is executed while the other waits. Even though they take turns to run, both programs have to be available to maintain the conversation. Increasingly, businesses want to be able to run related programs *independently* of each other.
- As the use of distributed applications has mushroomed, so too has the number of communication sessions in a network. Businesses that use a lot of distribution techniques can find themselves having to support thousands of dedicated network sessions. Such numbers may be manageable while the network is problem-free, even though long startup times are inevitable. But if the network fails, having a large number of sessions can cause performance problems, simply because of the time it can take to restart a part or the whole of the network.

***In summary, interoperability is a challenge...***

Most businesses have networks of diverse hardware and software. However, related programs in different parts of a network must be able to communicate in a way unaffected by variations in hardware, in operating systems, in programming languages, and in communication protocols. Moreover, businesses need to be able to run related programs independently of each other. And all this needs to be achieved with an overall reduction in the number of sessions on the network.

Complex though the problem may be, it needs a solution that works in the same way—and equally well—between programs on a single processor (in both like and unlike environments) and between programs at different nodes of a varied network.

***...that Messaging and Queuing can meet***

***MQSeries*** products provide just such a solution.

# How Messaging and Queuing works

***Programs communicate by putting messages on message queues***

Messaging and Queuing enables programs to communicate across a network, *without* having a private, dedicated, logical connection to link them. And it does this in a way that's simple, elegant, and proven: programs communicate by putting messages on message queues, and by taking messages from message queues.

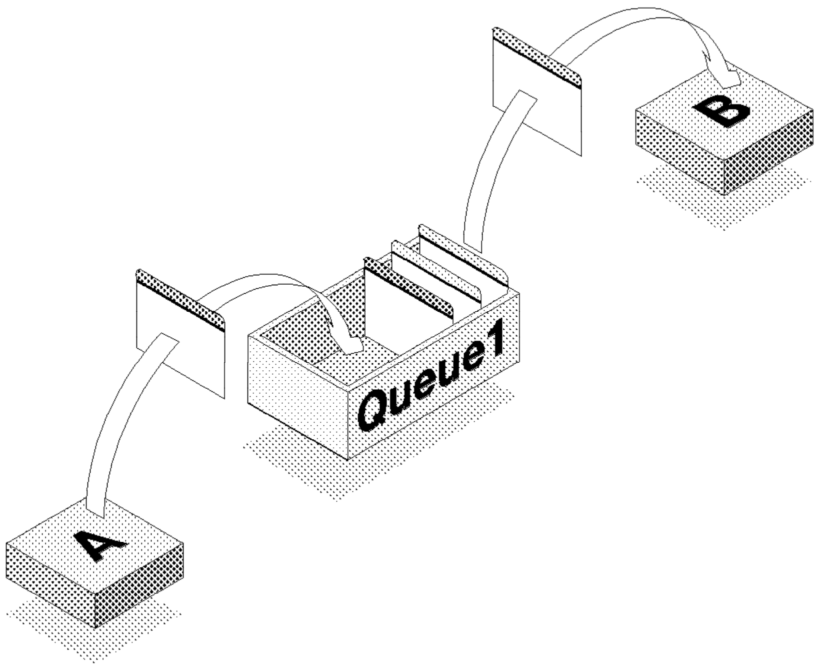


*Figure 1. Program A sends a message to program B via Queue 1*

In Figure 1, programs A and B could be running on the same processor in a single environment, on the same processor in different environments, or on different processors in different environments.

## how Messaging and Queuing works

***Communication can be one-way or two-way***

Messages can be one-way (from program A to program B, for example), or they can be reciprocated (a message from program A can cause program B to issue a reply message, for example). However, two-way communication isn't compulsory: if no response is required, none is sent.

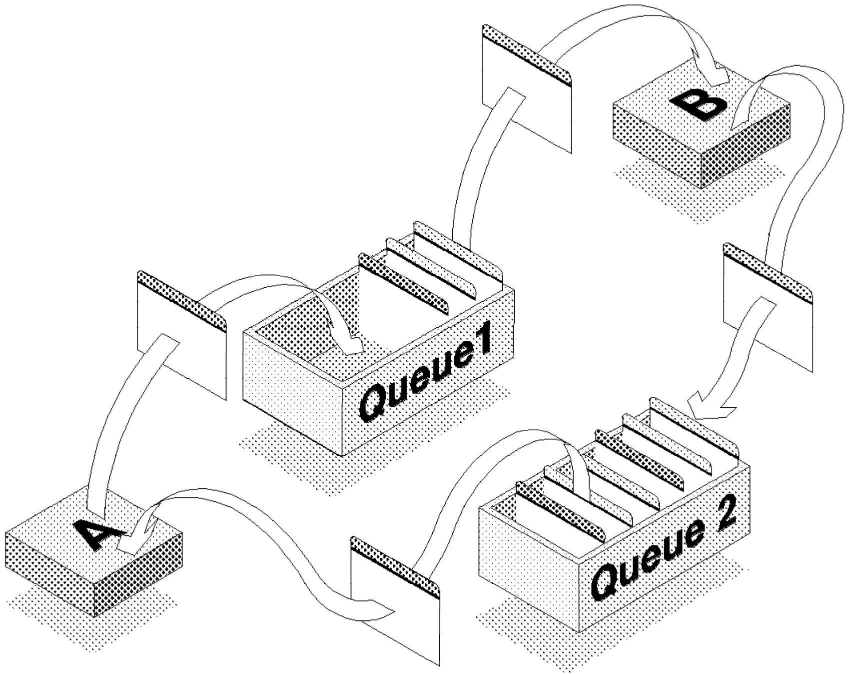


*Figure 2. Two-way communication between programs is optional. Program A communicates with program B via Queue 1. If program B needs to communicate with program A (whether to reply to a message from program A or for some unrelated reason), it puts a message on Queue 2.*

***Three characteristics of Messaging and Queuing***

Three key facts about Messaging and Queuing differentiate it from other communication styles:

1) Communicating programs can run at different times.
2) There are no constraints on application structure.
3) Programs are insulated from network complexities.

Let's examine each of these characteristics in more detail.

***1) Communicating programs can run at different times***

Programs do not talk to each other directly across the network, but indirectly by putting messages on message queues. And because there is no direct contact between programs, they don't have to be running at the same time. The target program can be busy at the time a message is put on the appropriate queue. The fact that a message has arrived doesn't affect the program's current processing, nor does it mean that it has to deal with that message immediately. In fact, the target program doesn't have to be running at all at the time the message is put on the queue. The target program can start running three hours or three weeks later, if that suits the business need.

***Either program can be busy or unavailable***

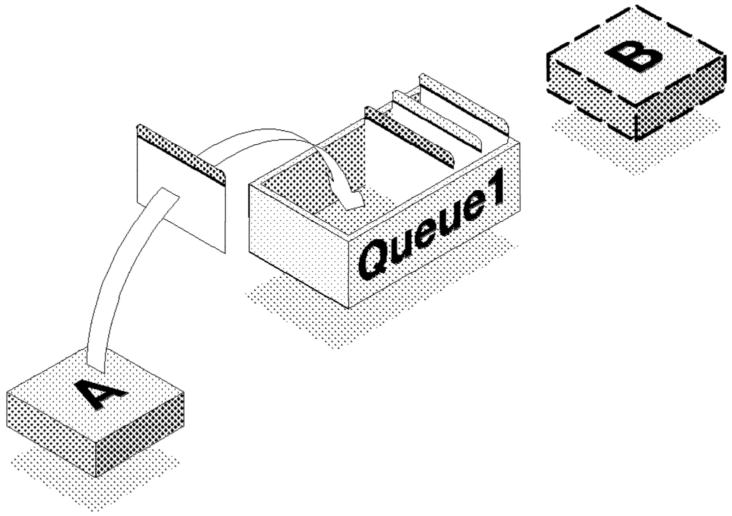


*Figure 3. Either program can be busy or unavailable. In this example, program B is unavailable. The message intended for program B is retained in Queue 1 until the program is available again. (Message queues exist independently of the programs that use them, so both programs could be inactive at the same time; the message on its way from one program to the other is nonetheless retained in Queue 1 until program B is ready for it.)*

***2) There are no constraints on application structure***

The one-to-one relationship between communicating programs shown so far, and the message-flow patterns between those programs—from program A to program B, and possibly from program B back to program A—are fairly simple. But the MQSeries products can support application structures and message-flow patterns that are much more complex than this.

***There can be a one-to-many relationship between programs...***

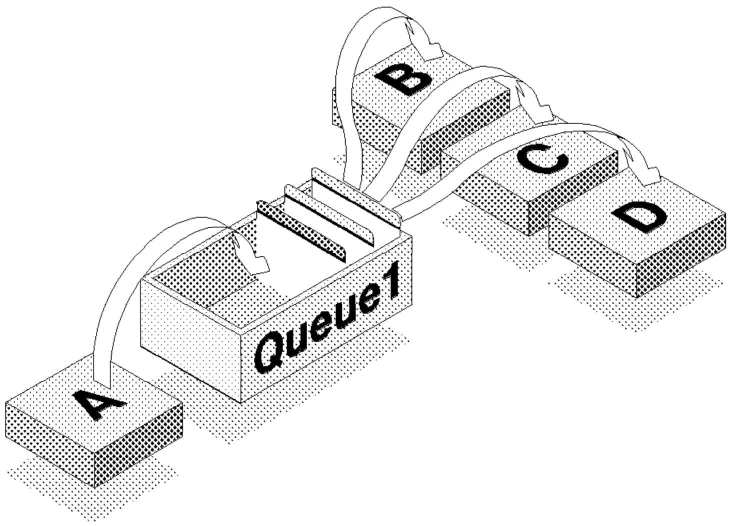


*Figure 4. A one-to-many relationship between programs. In this example, program A distributes work among programs B, C, and D. Message flow could be two-way if necessary, with any of programs B, C, and D sending messages to program A.*

In Figure 4, programs B, C, and D could be:

Three copies of a single program that are running concurrently for load-balancing purposes. That is, program A is generating work more quickly than a single instance of the target program could process it.

Three different programs taking messages from a single queue. For example, program B could be journaling while program C is processing: two activities that are usually performed in sequence are overlapped to do the work more quickly.

***... or a many-to-one relationship***

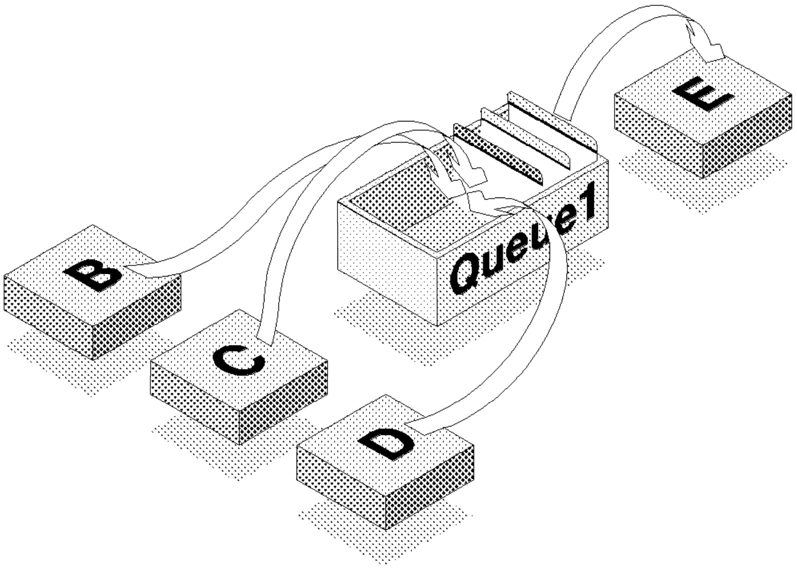


*Figure 5. A many-to-one relationship between programs. Program E can collect work from programs B, C, and D. Messages can flow from program E back to any of programs B, C, and D if necessary.*

In Figure 5, programs B, C, and D could be multiple clients of a single server, program E. However, because there is no direct connection between the server and the clients, the server is able to take messages from the queue either on a first in, first out (FIFO) basis or according to their priority. Thus, if program E (the server) knows that messages from program C, for example, are high priority, it can elect to take those messages ahead of messages from other programs. Without Messaging and Queuing, the client programs in a traditional client-server relationship must be processed in strict rotation.

**Note:** Not all products support priority ordering.

***And the one-to-one, one-to-many, and many-to-one relationships can be combined...***

The three basic program relationships—one-to-one, one-to-many, and many-to-one—can be brought together in a single application structure, and message flow between any two programs can be either one-way or two-way.

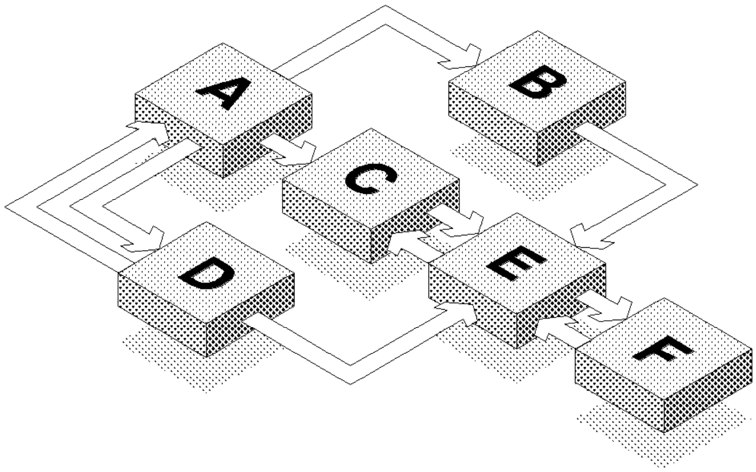


*Figure 6. Three basic program-to-program relationships combined. This application structure incorporates:*

*A one-to-one relationship (between programs E and F)*
*A one-to-many relationship (between programs A, B, C, and D)*
*A many-to-one relationship (between programs B, C, D, and E)*

*Message flow at any point can be one-way (as between A and B, for example) or two-way (as between C and E, for example).*

***... such that any application structure is possible***

Any combination of these "building-block" relationships is possible, which means that application structure has no constraints.

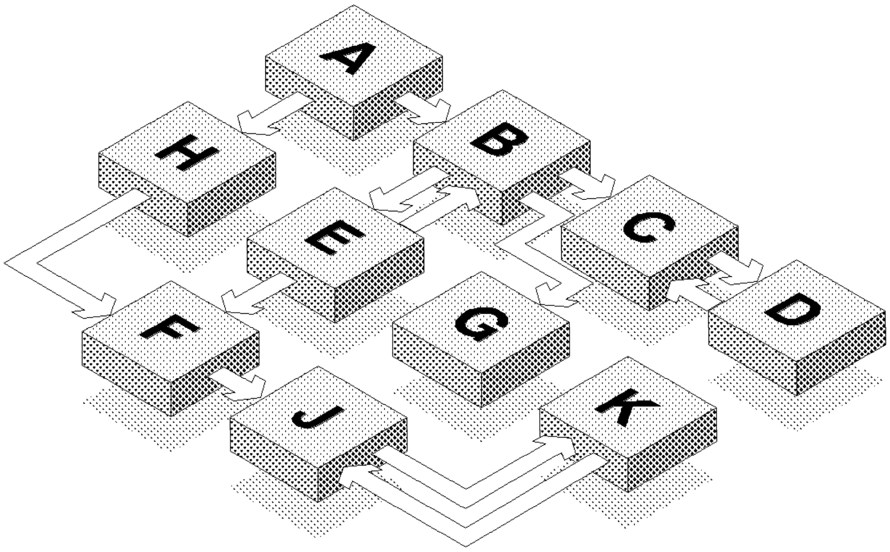


*Figure 7. There are no constraints on application structure.*

*Information can be relayed from program to program (from A to B to C to D, or from A to H to F to J to K).*

*Work can be distributed from one application to many (from B to E, G, and C).*

*Results can be collected by one program from many (by F from H and E).*

*Message flow can be one-way (A to B or F to J) or two-way (B to E or C to D).*

Furthermore, the programs shown in Figure 7 could be anywhere in the network: they could be running on a single processor, or dispersed throughout a network of unlike components, possibly in different geographical locations and time zones. MQSeries products make it possible for the programs of a distributed application to communicate using the same technique, regardless of their relative locations.

***3) Programs are insulated from network complexities***

Program A communicates with program B by placing a message on program B's message queue. Program B receives the communication by taking the message from the queue. All the activity associated with making this happen—maintaining message queues, maintaining the relationships between programs and queues, handling network restarts, and moving messages around the network—is the province of the MQSeries products. Programs do not talk directly to other programs, and they are not involved in the complexities of cross-network communication.

## how Messaging and Queuing works

***An example of a distributed application shows how***

To illustrate how the MQSeries products do their work, suppose that you have two programs, one to process customer orders, the other to bill the customer for the goods. For organizational reasons, these programs aren't running on the same computer. The Customer Orders program (CO) is running on a Tandem processor in a branch office in Chicago; the Customer Billing program (CB) is running on an ES/9000 mainframe at company headquarters in Düsseldorf. At some point, information has to be exchanged between CO and CB, so that a bill can be produced when goods are ordered. Using the interface supplied by the MQSeries products, the programs CO and CB can communicate at any time.

***Queue managers ensure that messages reach their target queues***

Here's how it works. Program CO tells program CB about a customer order by putting a message on CB's message queue (Queue 1). When CB takes the message from Queue 1, it uses the information in the message to produce a bill for the goods. CB might also want to send a response to CO (just an acknowledgement, perhaps, or confirmation that the goods have been charged for). CB would do this by putting a message on CO's message queue (Queue 2).

The heavy-duty work—moving messages from CO to Queue 1, and from CB to Queue 2—is managed by some "middleware", the *queue managers*.

***Programs use the Message Queue Interface (MQI)***

A program talks directly to its local queue manager (the one on the same processor as the program itself) using the *Message Queue Interface (MQI)*. The MQI is a set of calls that programs use to ask for the services of a queue manager. There are only two basic operations: put a message on a queue (using the MQPUT call), and take a message from a queue (using the MQGET call).

The MQI and the queue managers are part of MQSeries.

***There is a queue manager on each processor in the network***

As you can see in Figure 8, there's a queue manager on each of the two processors on which communicating programs are running. In our example, the queue manager on the Tandem processor could be provided by the MQSeries for Tandem Guardian product, and that on the ES/9000 processor by the MQSeries for MVS/ESA product. When communication across a network is required, the queue managers at the different nodes in the network communicate with each other. When communication between programs at a single node is required, it can be handled by a single queue manager. The programs themselves operate in ignorance of the network, and bear no part of the networking burden.

***Messaging and Queuing shields programs from network complexities***

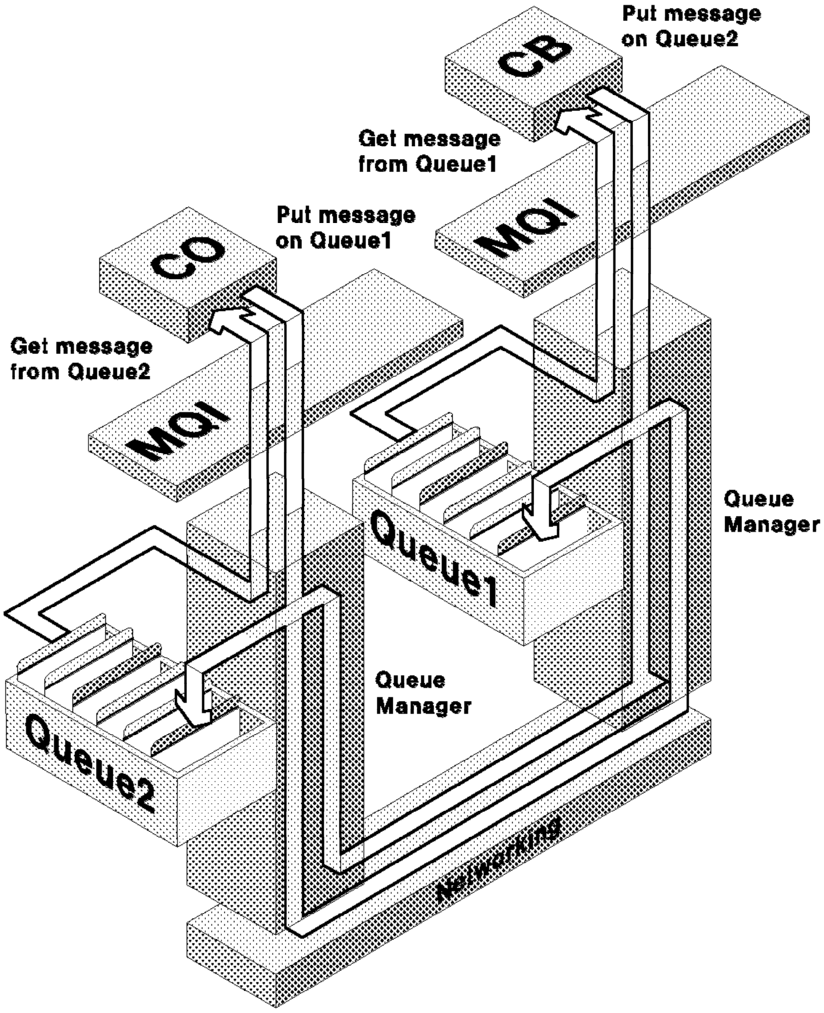


*Figure 8. Queue managers are "middleware". Programs CO and CB invoke the services of queue managers, via the MQI, to move messages across the network to their target queues.*

***Messaging and Queuing is an established technique...***

You might be surprised to learn that Messaging and Queuing is not new. In fact, it's a mature technique that's been used in a number of IBM and non-IBM products over many years. What *is* new about the MQSeries products is that, for the first time, a simple messaging interface (the MQI) is generally available to application programmers on the key industry platforms.

***...that is central to the IBM Networking Blueprint...***

The IBM Networking Blueprint, published in March 1992, identifies three key communication styles. These are:

Message Queue Interface (MQI)
Common Program Interface for Communications (CPI-C)
Remote Procedure Call (RPC)

MQI, CPI-C, and RPC are companion interfaces: you can use any combination of the three styles between communicating programs.

***...and is supported on major platforms***

There are already many products under the MQSeries "umbrella". In particular, Messaging and Queuing is implemented on the following platforms:

AT&T GIS UNIX
Digital VMS VAX
HP-UX
Tandem Guardian (C30)
Tandem Guardian Himalaya (D20)
SunOS
Sun Solaris
SCO UNIX
UnixWare
IBM AS/400 (OS/400)
IBM Operating System/2 (OS/2)
IBM RISC System/6000 (AIX)
IBM S/370 and S/390 environments including:
- MVS/ESA
- CICS/MVS
- CICS/VSE
- IMS
- TSO/E
- Batch

Client environment on AIX
Client environments on personal computer systems (OS/2, DOS, or Windows)

As you can see from this list, Messaging and Queuing isn't aimed solely at international corporations with high-end processors and the latest in transaction-processing systems. On the contrary, Messaging and Queuing is relevant to any business, large or small, where related programs must be able to communicate across a network, regardless of environmental variations.

## Some benefits of Messaging and Queuing

***Benefits derive from the key features:***

Three key features of the Messaging and Queuing style of programming are:

1) Communicating programs can run at different times.
2) There are no constraints on application structure.
3) Programs are insulated from network complexities.

From these derive all the benefits of Messaging and Queuing. Some of those benefits are explored in the following pages.

***Programmers can concentrate on business programming***

Programmers who don't have to write communication code can concentrate on the design and coding of business applications. Given that the most experienced programmers in any enterprise usually take on the most difficult assignments (and most would classify communication code as a "difficult assignment"), the MQSeries products allow you to redirect the energies of some of your best programmers.

***The MQI is easy to use***

Designing and writing application code to handle cross-network communication is one of the most difficult parts of the application programmer's job. By contrast, the MQI is simple to master and easy to use. There are just two basic calls—MQPUT and MQGET—that programs use to put messages on queues and take them from queues. The remaining calls (fewer than 10) are used infrequently or are optional. For example, there is one call (MQCONN) to set up contact with a queue manager, and another (MQDISC) to terminate contact; there are calls to open a message queue (MQOPEN) and to close a message queue (MQCLOSE); and there is one call for requesting information about a message queue (MQINQ).

Finally, not only is the MQI easy to use, but it's consistent from one environment to another.

***Constraints on program-to-program relationships are removed Constraints on program-to-program relationships are removed***

One of the characteristics of existing program-to-program communication techniques is that, in any pair or group of communicating programs, only one program can be processing at any one time. This fact can put constraints on the way you structure a distributed application.

By contrast, because Messaging and Queuing programs operate independently of each other, this constraint on the design of an application no longer applies. All the programs that make up a distributed application (and there can be any number) can run concurrently. Moreover, you can change the structure of a distributed application by adding or removing programs, and you can change the sequence in which those programs operate, without dismantling the whole application. The MQSeries products enable the relationships between programs to be more adaptable than ever before.

## some benefits of Messaging and Queuing

***Programs can be scheduled to make best use of resources***

Programs that communicate via the MQSeries products do not have a time-dependent relationship. That is, the various programs that make up a distributed application can run *at different speeds* and even *at different times* if necessary. Programs are able to operate independently because, if the application logic allows, they don't have to wait for responses from other programs before they can continue. This frees you to design and schedule programs to suit the logic of the work they are doing, rather than the logic of the communication method. Communicating applications can therefore run concurrently, or with a partial overlap, or with no overlap at all. This brings all the potential benefits of distributed processing within reach of application developers.

To show how the MQSeries products can improve resource usage, let's take a simple example: one program that generates 20 different questions, and a second program that answers those questions.

> The two programs could alternate their work, so that one question is answered before the next one is asked. This is the likely approach if the logic of the questions dictates that one question be answered before the next one can be formulated.
>
> If the relationship between the questions allows, the two programs could run at the same time: all 20 questions could be asked without waiting for the answers. This is one way of removing the processing "slack" that invariably occurs between communicating programs.
>
> For even more dramatic reductions in total processing time, the questions could be answered by 20 different programs, or by 20 instances of the same program, running at the same time. In other words, the processing of the 20 questions can be overlapped, so that they are all answered in the time it takes to ask and answer the "worst-case" question.

***Fewer network sessions are needed***

When programs communicate via the MQSeries products, private, dedicated, network sessions are not needed. Instead, sessions are between queue managers. As a consequence, the network traffic is concentrated on fewer sessions, and the total number of sessions in a network can be greatly reduced. This reduction in the number of sessions makes the network both easier to manage and faster to restart.

***Programs are less vulnerable to network failures***

Messaging and Queuing programs communicate by putting messages on queues and taking messages from queues. MQSeries products on each processor in a network are responsible for ensuring that a message reaches its target queue, regardless of the whereabouts of that queue. Therefore, cross-network communication sessions are established between queue managers rather than between individual programs. If a link between processors fails, it's the job of the queue managers to recover from the failure. Programs on the affected processors are not brought to a halt by such an event. In fact, they need not even be aware that it has happened.

***Better use can be made of all network resources***

In most networks, resources are not used as fully as they could be. A session is established, but is used for only a small percentage of the time; processors are available, but their workload has troughs as well as peaks; programs are installed, but are often waiting for responses from other programs before they can continue. Networks that use MQSeries products have fewer sessions, and use them more fully. Programs are able to run their natural course because they aren't forced to wait for responses from other programs, and can be scheduled to make best use of processor time.

***Code is easier to move and reuse***

Because they are independent of communication protocols, and because they use the same interface (the MQI) in all environments, Messaging and Queuing programs can be moved around the network from one node to another, or duplicated around the network, more readily than programs that contain environment-specific communication code.

***Business change is more readily accommodated***

Programs that use Messaging and Queuing have clearly defined inputs and outputs (messages) and a standard interface to other programs that does not vary as the programs themselves are changed and moved. Consequently, programs are easier to update than they would otherwise be, and so can be adapted more readily to meet new business requirements. Thus, even though a change to a business process may be necessary, you can be confident that its impact on the relevant application can be contained, and that cross-network communication will continue to work.

***Message delivery can be assured***

The data in a message can be as valuable as any other business asset. For example, the loss of a message that carries a funds transfer could mean financial losses for your business and your customers. For this reason, you can take the precaution of declaring valuable messages as *persistent*. A persistent message is written to nonvolatile storage, from where it can be reinstated after a system restart.

As well as assuring you that messages will be delivered, MQSeries products also ensure that the messages will be delivered once only, thus protecting, for example, against a possible double transfer if a message is carrying a funds transfer notice.

**Note:** Not all MQSeries products support non-persistent messages.

## some benefits of Messaging and Queuing

***Changes to resources can be coordinated***

There are times when two or three data updates must *all* work, or they must all be undone (backed out). If, for example, you're transferring money from one account to another, you need to ensure that an amount is not credited to one account unless it's also debited from the other account. The programs carrying out this task must therefore be coordinated: each must be able to undo any changes if the other is not successful, so that the job isn't left half done.

In environments that have a syncpoint manager, the MQSeries products allow programs to defer commitment of changes made at a single node of the network until all parties can commit. This is known as *syncpoint* processing, a concept you'll be familiar with if you are a CICS, IMS, or DB2 user.

***A gradual transition is possible***

The good news is that you don't have to get new equipment or throw away your existing applications. You can adopt the Messaging and Queuing style of programming gradually, and at a pace that suits your business. Here are four approaches to implementing the MQSeries products, beginning with the approach that has the least impact on existing environments:

1) You can use a technique of *interception* to introduce Messaging and Queuing to well-established code that you don't want to alter (for example, if the program is not modular and the logic is too involved, or you don't have the source of the program at all). Suppose, in the 20 questions application mentioned earlier, there is a single program (for which you don't have the source code) that answers all 20 questions. Question 12 changes, so you need to update the code that answers it: you write a new Messaging and Queuing program to process question 12 only, and you alter the program that generates the questions so that it redirects question 12, using the MQI, to the new program. Thereafter, question 12 does not reach the original program. You can repeat this technique as required for each of the questions until the original program is no longer called.

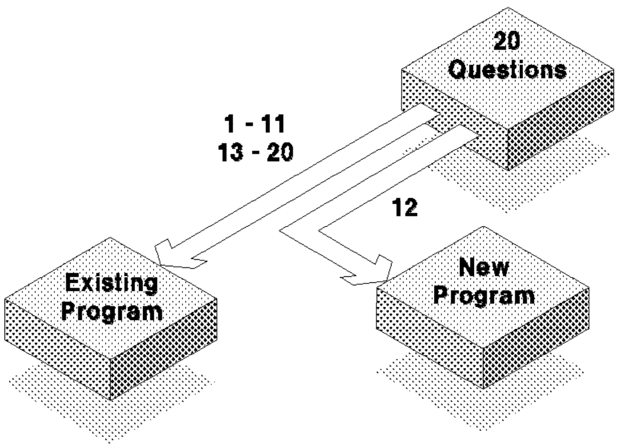


*Figure 9. A technique of interception. New Messaging and Queuing programs can take over the functions of existing programs gradually.*

2) To alter or extend the function of an existing application, you can code Messaging and Queuing programs that coexist with existing programs. The old and the new might need to be link-edited, but no other alterations need to be made.

3) You can make selective alterations to a program by replacing existing, communication-related calls with MQI calls. This approach might also require changes to the overall logic of the application, because related programs will now be able to run independently of each other.

4) You can design and write completely new business applications that make full use of the benefits of the MQSeries products.

# Some Messaging and Queuing examples

***Messaging and Queuing in the "real" world***

Here are some examples of the ways in which the MQSeries products can be used in the "real" world. The examples are taken from industries that are widely understood; even if you aren't working in these industries yourselves, you are almost certainly familiar with their workings. From the chosen examples, you might be tempted to conclude that Messaging and Queuing is only for large enterprises with top-of-the-range processors. Messaging and Queuing can be applied in a vast range of business environments, and is useful in *any* enterprise, large or small, where there is a need to get programs to talk to each other.

In each of the examples, key points about Messaging and Queuing have been highlighted. However, the following are common to all potential uses of the MQSeries products:

- No Messaging and Queuing program includes communication code.
- Network sessions are between MQSeries products at each node of the network, not between the programs themselves. As a result, the network is likely to carry fewer sessions.
- The relative locations of two communicating programs are unimportant.
- The MQSeries products accommodate dynamically changing and unpredictable message flows as easily as they accommodate unchanging and predictable message flows.
- Both high volumes of message traffic and high performance can be sustained.

***An example from the insurance business***

***Independently operating but related programs***

Insurance agents throughout the country ask for insurance quotations using an online, menu-driven system. This system is provided by a traditional client-server application, with client programs (the insurance agents) sending requests for quotations to a central server program. The server does some calculations using data from a central insurance database, then sends a quotation to the requesting agent. Using Messaging and Queuing, the client programs put request messages on a single queue, from which the server program takes them.

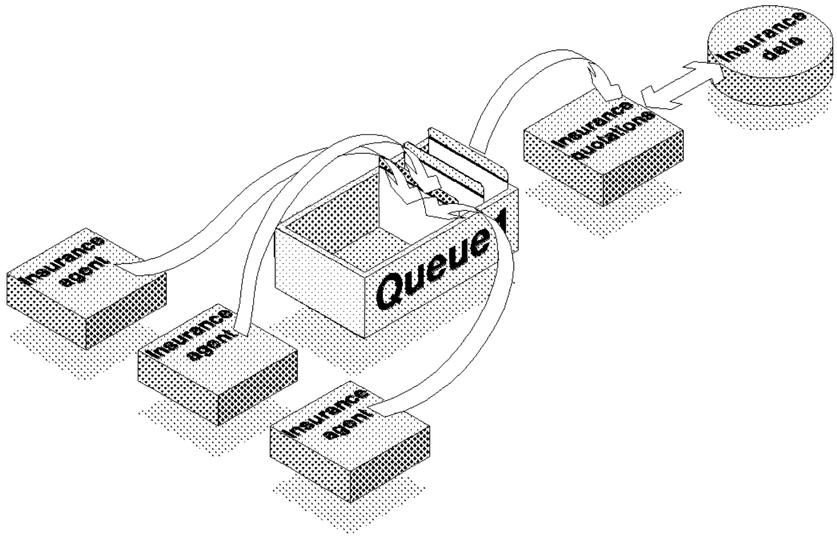


*Figure 10. Independently operating but related programs. A client-server relationship, with client and server programs running independently of each other.*

***...and some key points***

The primary advantages of using Messaging and Queuing in this example are:

- The client programs and the server can all run at the same time; the server doesn't need to communicate with the clients in turn, nor is any program suspended simply because another program is running.
- There can be any number of client programs, and the number can vary dynamically under the control of a locally written "monitor program". For example, insurance agents could request quotations using portable personal computers as they travel around the country. A client program could be running while an agent is interacting with it, but could be stopped by the monitor program whenever interaction ceases. (It's generally true of the MQSeries products that available resources are used fully, and that resources that are not being used do not need to be available.)
- The network is almost certain to be heterogeneous, because it is linking multiple, independent insurance agencies. Each agency has selected its hardware and software, and written its application programs, in isolation from the other agencies in this network. However, provided both server and clients use the MQI to communicate, differences in processors, in operating environments, and in the programs themselves can be disregarded.
- If any of the links becomes unavailable, client and server programs can continue working. This is especially important in the case of the server, which can continue to process messages as they arrive from other clients. No special programming of the server program is necessary to make this possible.

***An example from manufacturing industry***

## ***Output-only devices***

In many businesses, output-only devices are constantly updated with instructions or information. Examples include:

- Printing devices
- Displays of information, such as stock-exchange prices or flight arrivals and departures
- Factory-floor robotics, where a steady stream of information controls the operation of machinery

Ideally, such information flows would be one-way because, in general, no response is required from the target program. In most program-to-program communication models, however, communication has to be two-way because the source program is suspended until the target program replies. Using Messaging and Queuing, the requirement for a one-way information flow can be met.

## some Messaging and Queuing examples

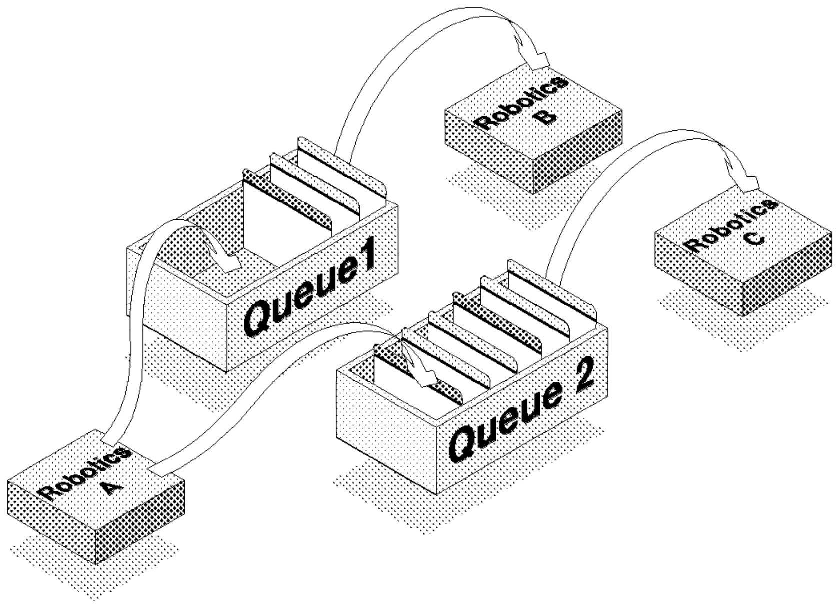


*Figure 11. One-way message flows for an output-only device. The Robotics A program is in control of an automated manufacturing process. It puts messages on Queue 1 for the Robotics B program, which directs some welding machinery, and on Queue 2 for the Robotics C program, which controls a paint sprayer.*

***...and some key points***

Some of the main advantages of Messaging and Queuing for this type of application are:

The source program is not suspended between the transmission of one instruction message and the next.

The predictable, one-way message flow characteristic of applications such as these yields excellent performance.

Source and target programs can run at the same time, but at different speeds.

*An example from the retail industry*

## *A solution to the classic “batch window” problem*

A department store writes its sales figures to a file throughout the day’s trading. Overnight, a report of the day’s sales is produced using this file of data as input. The report must be on the Sales Manager’s desk before the next day’s trading begins, which presents the store’s data-processing department with two challenges: the first is that the amount of time available for producing the report is limited to the “window” of time between the end of business on one day and the start of business on the next; the second is that the actual start and finish times for this activity are fixed.

Instead of operating in sequence and communicating via a file, the two programs could run independently of each other and communicate using Messaging and Queuing.

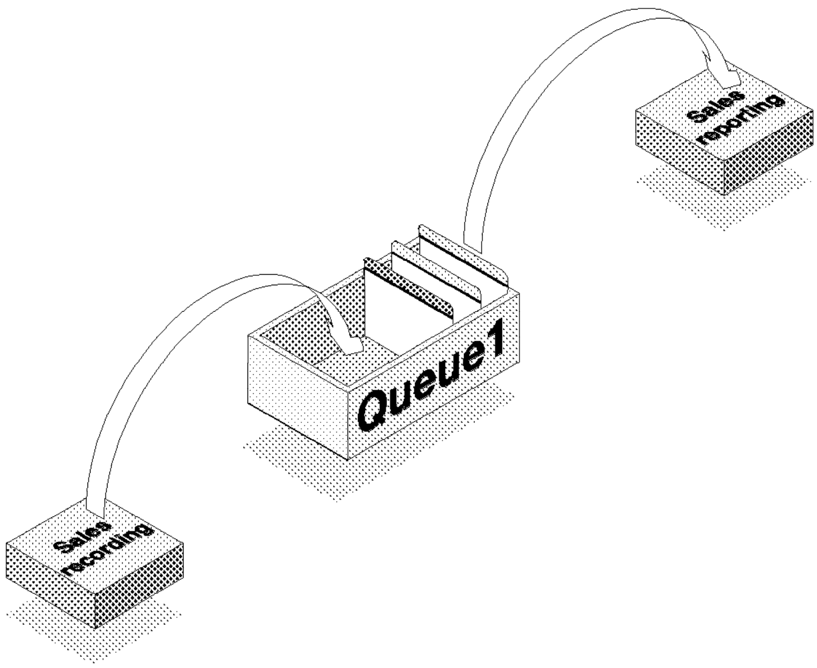


*Figure 12. Addressing the “batch window” problem. The recording and reporting of sales figures need no longer be strictly sequential processes, but can be overlapped.*

***...and some key points***

The particular advantages of Messaging and Queuing in this example are:

- The two programs can operate independently of each other:
  - They can run in sequence.
  - They can start running at the same time.
  - The reporting program can overlap its processing with that of the sales-recording program.
  - The reporting program can run intermittently throughout the day to use spare processing power.

  Greater flexibility in the scheduling of the programs increases the available time in which the report can be produced, allowing the business to use its processors more efficiently and improving the chances of meeting the deadline for the sales report. The business is no longer constrained by the time window between the end of one day's processing and the start of the next.

  In the banking, securities, and other industries, this is known as the "batch window" problem because data accumulated (batched) throughout the day must be processed, in the background, between the close of business on one day and the start of business on the next. The Messaging and Queuing solution to the batch window problem is of special interest to the securities business, where the batch window during which a day's transactions can be processed is limited (by legal requirements) to the time between the close of trading and the close of business (typically 90 minutes) on any one day.

- Messages carrying sales data are written to a message queue, which is owned by an MQSeries product. The programs themselves do not need to be involved in file control, nor do they need to be concerned with the recoverability of the data. Message delivery can be assured[1], and messages are delivered once only.

- In this example, the function of the message queue is similar to that of a file. However, all the data doesn't have to be accumulated before it can be processed, because a message queue can be read from while it is being written to: the first message on the queue is instantly available for processing. And, if the processing of the two programs is overlapped, less storage in total is likely to be required for the sales data.

- The overhead of converting from the existing approach, where programs communicate via an intermediate file, to the Messaging and Queuing style, where programs communicate by putting messages on a message queue, is small.

- The system is extendable: as new stores are opened or acquired, the same sales-recording techniques can be used. Equally, the store's suppliers could be linked to the store's network, so that inventory control could benefit from the ready availability of the data.

---

[1] Assured delivery means that, once the message has been written to nonvolatile storage, it is retained until successfully delivered, subject to the reliability of the nonvolatile storage.

***An example from the travel industry***

***Load balancing***

This example is taken from the airline business, where seat reservations are made by an airline in response to requests from multiple travel agencies. In data-processing terms, this is a traditional client-server relationship, with multiple clients (the travel agencies) requesting flight bookings from a single server (the airline). The server program takes booking requests from a single queue of messages (either as they arrive on the queue or according to their priority), and updates a database of seat reservations as appropriate. The server also sends a reply message to the relevant client program. The reply could inform the client that the requested seat has been reserved or, if the seat is unavailable, it could include some suggestions for alternative seats or flights.

When the number of requests on the queue reaches a particular level, a “monitor” program could start additional instances of the server program.

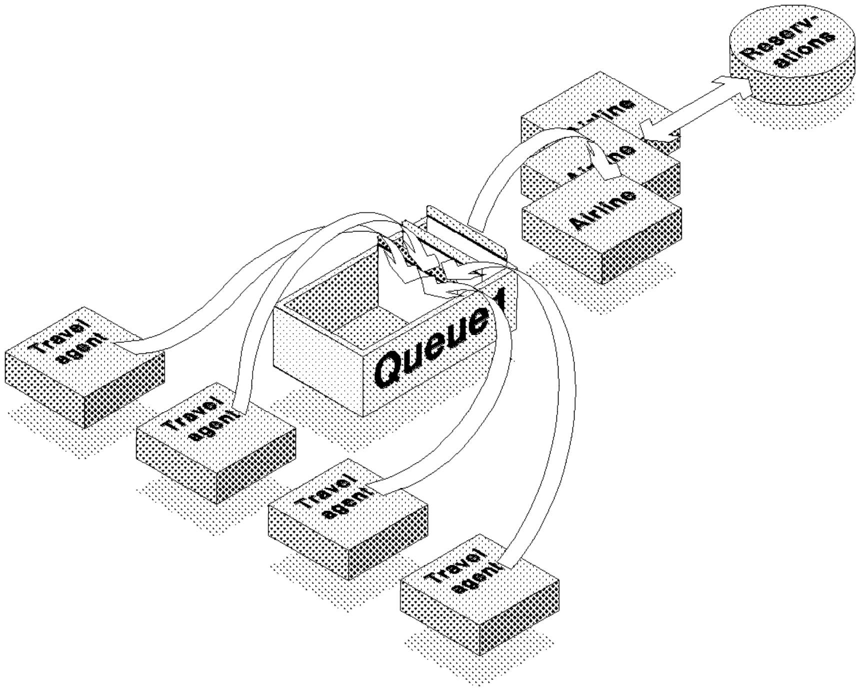


*Figure 13. Load balancing. Large and fluctuating volumes of message traffic can be processed by multiple instances of a server program.*

***...and some key points***

The advantages of Messaging and Queuing in this example are:

Multiple instances of the server program can be started during peak-load periods, and those program instances can be stopped as the volume of requests subsides. In this way, the booking system can offer its customers a consistently high level of service.

Clients and server can run at the same time: there is no question of any client program being suspended while the server is processing a seat reservation. Furthermore, the client programs can all send messages to the server's message queue at any time, regardless of whether the server program is busy.

Requests can be serviced as they arrive on the server's message queue, or according to priority. For example, a particular travel agent, or a particular category of message from any client program, could be accorded a higher priority than some others by the server program.

**Note:** Not all products support priority ordering.

If the connection between one of the client nodes and the server node is unavailable, the remaining client-server relationships are not affected.

The client programs do not have to be designed and coded in the same way, nor do they have to run in identical environments.

***An example from the banking world***

## ***Synchronization points in transaction processing***

Increasing use is being made of electronic point of sale (EPOS) terminals in shops, at service stations, and elsewhere. A transaction is recorded at the EPOS and stored until it can be sent to a handling agency, usually a bank. The bank must ensure that both halves of the transaction—the subtraction of money from the customer's account and its addition to the retailer's—occur. In environments that have a *syncpoint manager* (a "neutral" program that synchronizes updates to resources), related changes to multiple resources can be coordinated: either all related changes occur, or they are all undone.

Messaging and Queuing programs running on a single processor can participate fully in syncpoint control, even though they may be running asynchronously and communicating using the MQI: changes to the contents of message queues can be coordinated with changes to other resources.

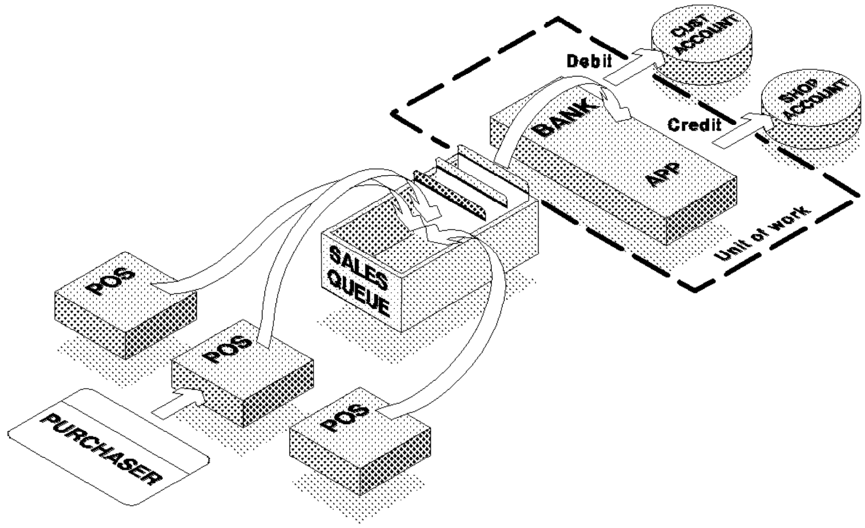


*Figure 14. Syncpoint participation. A customer sale is recorded at an EPOS terminal and sent to to the sales message queue. The bank application gets a message from the sales queue and instructs the Debit program to remove funds from a customer account. The Credit program places the funds into the shop's account. Updates to the resources constitute a single unit of work.*

***...and some key points***

Some points to note about the role of Messaging and Queuing in syncpoint control:

Changes to a message queue "resource" are treated in the same way as changes to other resources. That is, MQGET or MQPUT calls issued under syncpoint control are not completed until the current unit of work ends successfully. If the unit of work cannot be committed, all resources return to their state before the unit of work was started.

Messages can be defined as persistent, so that they can be reinstated if the queue manager is restarted. (Syncpointing, by contrast, insures against failure or restart of the program or any other resource manager participating in the unit of work.)

Queue managers do not participate in syncpoint control across multiple nodes of a network. This avoids the risk of many different data resources, possibly distributed around the world, all being "locked" until the current unit of work ends: if one of the resources couldn't be updated for some reason, or if part of the network were to be unavailable, all the other resources involved in that unit of work would be tied up until the problem could be resolved. So that this doesn't happen, queue managers participate in syncpoint control within a single node only.

## Messages and message queues

***Some detail about messages and message queues***

Finally, for the benefit of the application programmers and designers among you, let's take a look at two of the key players in the MQSeries products—messages and message queues. For a more detailed explanation of the Messaging and Queuing concepts, see the *MQSeries Message Queue Interface Technical Reference*, SC33-0850.

***Messages...***

A message is mainly a string of bits and bytes (data such as account numbers, account balances, booking requests, names and addresses, images of documents — anything at all, in fact) that one program wants to send to another. This data is called the *application data*. Of course, a message needs to include other information, such as its destination and possibly a return address. This type of data is called the *message descriptor*.

***...are defined by programs***

The program sending the message defines the application data and supplies it. Such data can include character strings, bit strings, binary integers, packed-decimal integers, floating-point numbers—whatever you want, basically.

***...are of four types***

There are four types of message:

A *request* message is used by one program to ask another program for something (usually data). A request message needs a reply.

A *reply* message is used in response to a request message.

A *one-way* message, as you would expect, doesn't need a reply, though it can carry data.

A *report* message is used when something unexpected occurs. For example, if the data in a reply message is not usable, the receiving program might issue a report message.

Messages are labeled in this way so that the target program can know what's expected of it, without having to examine the message in detail. Queue managers have no interest in the message type.

***...can have two identifiers***

Every message has a *message identifier*. This is a unique, 24-byte string assigned by the queue manager. A message can also have a *correlation identifier*. The correlation identifier, also a 24-byte string, is a field that you can use for any purpose, but you're recommended to use it if your program needs to identify related messages. The correlation identifier is assigned by the program. It could contain a customer account number, for example, or the identifier of the initial message in a sequence, or just a random selection of characters that means something to the program. Thus, the correlation identifier is constant in a group of related messages, but the message identifier is different for each message.

## messages...

***...can be retrieved out-of-order***

Although this is a standard queuing system (when a message arrives, it goes to the back of the queue), you can ask for a message with a particular identifier to be given to you next rather than the message that happens to be next in the queue.

***...can be defined as persistent***

Messages that must be delivered, come what may, can be defined as *persistent*. Persistent messages are logged so that they can be reinstated after a system restart. However, there is an overhead in logging messages, so this classification should be reserved for valuable messages only. Other (nonpersistent) messages are fast moving and, by definition, of little or no value. If they go astray, the effect of their loss is usually not felt at all, or is quickly made good. For example, messages that update a screen of information at 20-second intervals might come into this category, because the loss of one such message would be made good 20 seconds later.

**Note:** Not all MQSeries products support non-persistent messages.

***...can include a format name***

The sender of a message tells the receiver how to interpret the message data by supplying the name of a data structure (an 8-byte *format name*) in the message's control information. Obviously, the format must be one that's understood by both parties. For example, if the message is requesting information about a customer, the format name might be "QCUST", while the format name in the corresponding reply message might be "CUSTINFO". Neither the format nor its name is imposed by the queue manager.

***...can include a return address***

The sender of a request message must specify the name of the queue to which the reply should be sent. This is called the *reply-to queue*. The reply-to queue is the key to a flexible application structure, because it allows you to choose where the reply goes: it isn't necessarily going to be processed by the sender of the original request.

***...can be copied from a queue***

A message doesn't have to be removed from the queue when you ask for it. You can ask for a copy of the message, so that the original stays on the queue. You might want to do this if two programs need to do something with a single message. (At some point, one of them should remove it from the queue to stop the queue filling up with old messages.)

***...never include the name of the target program***

Programs never interact directly with other programs. The name of the program that is to process a message is never recorded in that message.

***Message queues...***

At its simplest, a message queue is an area of storage set aside by the queue manager to hold messages on their way from one program to another. By default, it works like a physical queue: first in is first out. Whether the queue is in main store only, or both in main store and on DASD, is for your system-support personnel to decide. Persistent messages go to queues on DASD.

***...can be local or remote***

A program interacts with a queue manager, and queues belonging to that queue manager are *local* to the program. Queues belonging to other queue managers (whether they're on the same node of the network or on the other side of the world) are *remote* queues.

***...can be redefined and relocated***

A message queue can be known by one name to the programs that use it, and by a different name to the queue manager that owns it: the queue manager knows that the two names identify a single queue. Therefore, even if a queue is moved and the name by which it is known to the queue manager changes, the name by which programs know the queue can remain unaltered. Programs can continue to read from and write to the queue, regardless of its new name and location.

***...can be limited in size***

When an MQSeries product is installed, your system-support personnel define the resources that are going to be used. These are (chiefly) queue manager instances, message queues, and the relationships between them. When queues are defined, they are given a number of attributes. One of these controls the number of messages a queue can hold, and another can be used to limit the length of a message. The two values together define the maximum size of the queue.

***...but are not limited in number***

The number of queues managed by a queue manager is not related to the number of programs that put messages on them or take messages from them. Rather, the number of queues managed by a queue manager is a system-design and system-management consideration.

## for more information...

| | |
|---|---|
| ***For more information...*** | This completes our introduction to the Messaging and Queuing style of programming implemented by the MQSeries products. For a more detailed description of Messaging and Queuing, see the MQSeries publication *MQSeries Message Queue Interface Technical Reference*, SC33-0850. If you would like to know more about the MQSeries products, contact your IBM representative. |

## Sending your comments to IBM

**An Introduction to Messaging and Queuing**

**GC33-0805-01**

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book. Please limit your comments to the information in this book and the way in which the information is presented.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

You can send your comments to IBM in any of the following ways:

By mail, use the Readers' Comment Form.

By fax:

- From outside the U.K., after your international access code use 44 1962 870229
- From within the U.K., use 01962 870229

Electronically, use the appropriate network ID:

- IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
- IBMLink: WINVMD(IDRCF)
- Internet: idrcf@winvmd.vnet.ibm.com

Whichever you use, ensure that you include:

The publication number and title
The page number or topic to which your comment applies
Your name and address/telephone number/fax number/network ID.

# Readers' Comments

## An Introduction to Messaging and Queuing

**GC33-0805-01**

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Be sure to print your name and address below if you would like a reply.

| | |
|---|---|
| Name | Address |
| Company or Organization | |
| Telephone | Email |

**An Introduction to Messaging and Queuing**
**GC33-0805-01**

IBM

**You can send your comments POST FREE on this form from any one of these countries:**

| | | | | | |
|---|---|---|---|---|---|
| Australia | Finland | Iceland | Netherlands | Singapore | United States of America |
| Belgium | France | Israel | New Zealand | Spain | |
| Bermuda | Germany | Italy | Norway | Sweden | |
| Cyprus | Greece | Luxembourg | Portugal | Switzerland | |
| Denmark | Hong Kong | Monaco | Republic of Ireland | United Arab Emirates | |

If your country is not listed here, your local IBM representative will be pleased to forward your comments to us. Or you can pay the postage and send the form direct to IBM (this includes mailing in the U.K.).

1 Cut along this line

2 Fold along this line

By air mail
*Par avion*

IBRS/CCRI NUMBER: PHQ - D/1348/SO

NE PAS AFFRANCHIR

NO STAMP REQUIRED

IBM

REPONSE PAYEE
GRANDE-BRETAGNE

IBM United Kingdom Laboratories Limited
Information Development Department (MP 095)
Hursley Park
WINCHESTER, Hants
SO21 2ZZ United Kingdom

3 Fold along this line

*From:* Name ______________________
Company or Organization ______________________
Address ______________________
______________________
EMAIL ______________________
Telephone ______________________

1 Cut along this line

4 Fasten here with adhesive tape

Printed in U.S.A.

GC33-0805-01