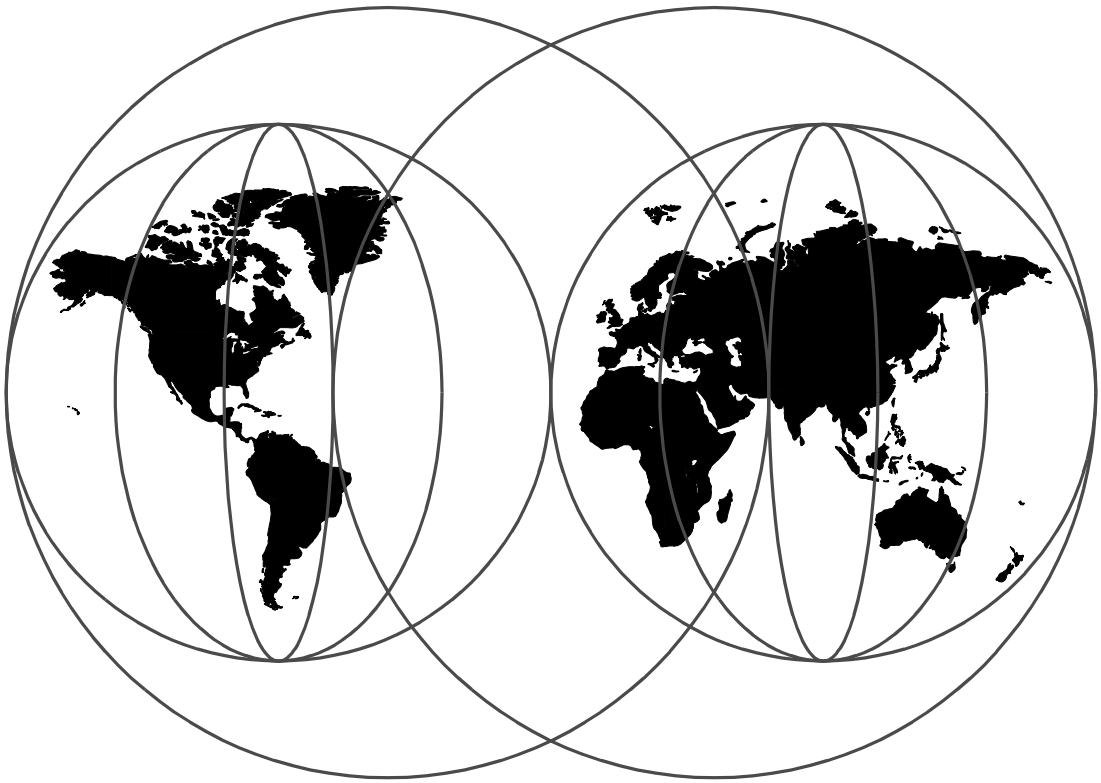
MQSeries Primer



MQSeries Enterprise Application Integration Center

Dieter Wackerow

MQSeries is IBM's award winning middleware for commercial messaging and queuing. It is used by thousands of customers in every major industry in many countries around the world. MQSeries speeds implementation of distributed applications by simplifying application development and test.

MQSeries runs on a variety of platforms. The MQSeries products enable programs to communicate with each other across a network of unlike components, such as processors, subsystems, operating systems and communication protocols. MQSeries programs use a consistent application program interface (API) across all platforms.

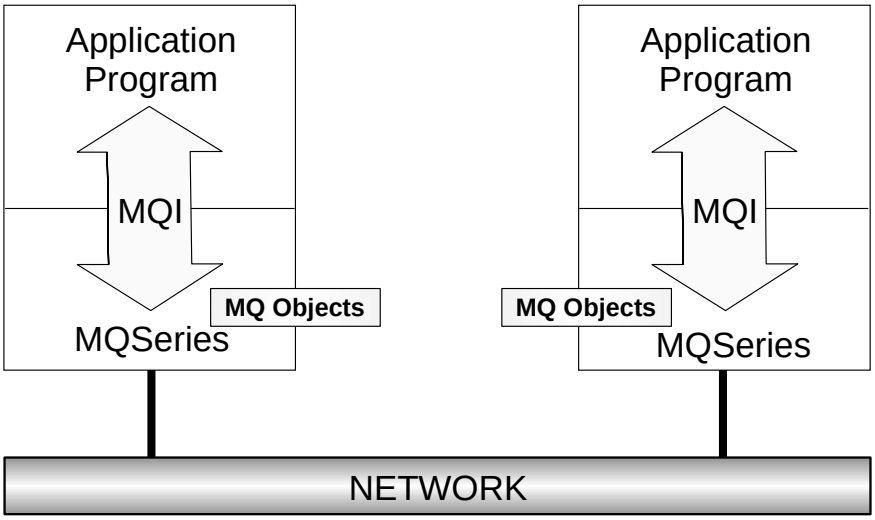


Figure 1. MQSeries at Run Time

Figure 1 shows the main parts of an MQSeries application at run time. Programs use MQSeries API calls, that is the Message Queue Interface (MQI), to communicate with a queue manager (MQM), the run-time program of MQSeries. For the queue manager to do its work, it refers to objects, such as queues and channels. The queue manager itself is an object as well.

The following provides a brief overview of MQSeries, including clients and servers.

What is Messaging and Queuing?

Message queuing is a method of program-to-program communication. Programs within an application communicate by writing and retrieving application-specific data (messages) to/from queues, without having a private, dedicated, logical connection to link them.

Messaging means that programs communicate with each other by sending data in messages and not by calling each other directly.

Queuing means that programs communicate through queues. Programs communicating through queues need not be executed concurrently.

With *asynchronous messaging*, the sending program proceeds with its own processing without waiting for a reply to its message. In contrast, *synchronous messaging* waits for the reply before it resumes processing. For the user, the underlying protocol is transparent. The user is concerned with conversational or data-entry type applications.

MQSeries is used in a client/server or distributed environment. Programs belonging to an application can run in one workstation or in different machines on different platforms. Applications can easily be moved from one system or platform to another. The programs can be written in various programming languages, including Java. The same queuing mechanism is valid for all platforms, and so are the currently 13 APIs.

Since MQSeries communicates via queues it can be referred to as using indirect program-to-program communication. The programmer cannot specify the name of the target application to which a message is sent. However, he or she can specify a target queue name; and each queue is associated with a program. An application can have one or more “input” queues and may have several “output” queues containing information for other servers to be processed, or for responses for the client that initiated the transaction.

The programmer does not have to worry about the target program being busy or not available. He or she isn’t even concerned about the server being down or having no connection to it. The programmer sends messages to a queue that is associated with an application; and the application may or may not be available at the time of the request. MQSeries takes care of the transport to the target application and even starts it, if necessary.

If the target program is not available, the messages stay in a queue and get processed later. The queue is either in the sending machine or in the target machine, depending whether the connection between the two systems can be established or not. Applications can be running all day long or they can be triggered, that is, automatically started when a message arrives or after a specified number of messages have arrived.

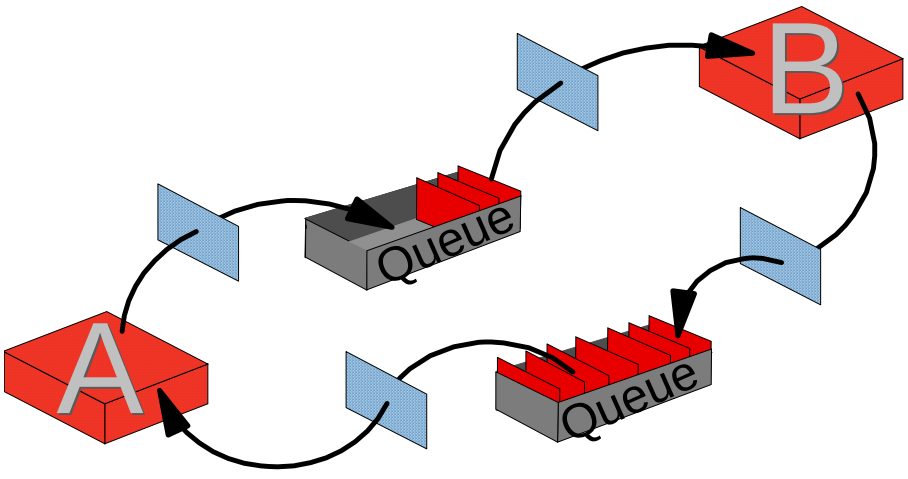


Figure 2. Messages and Queues

Figure 2 on page 4 shows how two programs, A and B, communicate with each other. We see two queues; one is the “output” queue for A and at the same time the “input” queue for B, while the second queue is used for replies flowing from B to A.

The squares between the queues and the programs represent the Message Queuing Interface (API) the program uses to communicate with MQSeries’ run-time program, the queue manager. As said before, the API is a simple multi platform API consisting of 13 calls. The API will be discussed later.

About Messages

A message consists of two parts:

1. Data that is sent from one program to another
2. The message descriptor or message header

The message descriptor identifies the message (message ID) and contains control information, also called attributes, such as message type, expiry time, correlation ID, priority, and the name of the queue for the reply.

A message can be up to 4 MB or 100 MB long, depending on the MQSeries version you use. MQSeries Version 5 (for distributed platforms) supports a maximum message length of 100 MB.

Message Segmenting and Grouping

In MQSeries Version 5, messages can be *segmented* or *grouped*. Message segmenting can be transparent to the application programmer. If permitted, the queue manager segments a large message when it does not fit in a queue. On the receiving end, the application has the option to either receive the entire message in one piece or each segment separately. This may depend on the buffer size available for the application.

A second method of segmenting leaves the programmer in control so that he or she can split a message according to logical boundaries or buffer size available for the program. The programmer puts each segment as a separate physical message; thus several physical messages build one logical message. The queue manager ensures that the order of the segments is maintained.

To reduce traffic over the network, you can also group several small messages together and build one larger physical message. This message is then sent to the destination and is there disassembled. Message grouping also guarantees that the order the messages are sent in is preserved.

Distribution Lists

Using MQSeries Version 5, you can send a message to more than one destination queue with one MQPUT call. This is done with a dynamic *distribution list.* A distribution list can be a file that is read at the time an application starts. It can be modified any time. It contains a list of queue names and the queue managers that own them. A message sent to multiple queues belonging to the same queue manager is sent over the network only once and so reduces network traffic. The

receiving queue manager replicates the messages and puts them into the destination queues. This function is called *late fan-out.*

Message Types

MQSeries knows four types of messages:

Datagram: A message containing information for which no response is expected.
Request: A message for which a reply is requested.
Reply: A reply to a request message.
Report: A message that describes an event such as the occurrence of an error or a confirmation on arrival or delivery.

Persistent and Non-Persistent Messages

Application design determines whether a message must reach its destination under any circumstances, or if it can be discarded when it cannot get there in time. MQSeries differentiates between *persistent* and *non-persistent* messages. *Delivery of persistent messages is assured*; they are written to logs to survive system failures. In an AS/400 these logs are Journal Receivers. Non-persistent messages cannot be recovered after a system restart.

The Message Descriptor

The table below contains some interesting attributes of the message descriptor. We mention them here because they explain some of the functions the queue manager provides for you.

Version	Return address
Message ID / Correllation ID	Format
Persistent / non-persistent	Sender application and type
Priority	Report options / Feedback (COA, COD)
Date and time	Backout counter
Lifetime of a message	Segmenting / grouping information

Figure 3. Some Attributes of the Message Descriptor

- The *version* of the message descriptor depends on the MQSeries version and platform you use. For the functions introduced with Version 5 additional fields were needed to keep information about segments and their order and distribution list information, to name a few. This enlarged structure carries the version number 2. Other queue managers who don't support these functions ("Version 1 queue managers") treat the additional information as data.

- *Message* and/or *correlation ID* are used to identify a specific request or reply message. The programmer can move a value in one or both fields or have MQSeries create a unique ID for him or her. Before the programmer puts the request message in the queue he or she can save the ID(s) and use them in a subsequent get operation for the reply message. The program that receives the request message copies this information into the reply message. This allows the originating program (the one that gets the reply) to instruct MQSeries to look for a specific message in the queue instead of getting the first one in the queue.

- We discussed *persistent* and *non-persisten*t messages earlier. Persistent messages always arrive at their destination, even when the system fails. They are “hardened”, that is, saved on disk. You can make a specific message persistent or all messages on a particular queue.

- You can assign a *priority* to a message and so control the order in which it is processed.

- The queue manager stores *time* and *date* when the MQPUT occurred in the message header. The time is in GMT and the year has four digits and so is Y2K compliant.

- You can also specify an *expiration date*. When this date is reached and an MQGET is issued, then the message will be discarded. There is no “daemon” that checks queues for expired messages. Expired messages can stay in a queue for weeks, until a program attempts to read it.

- The return address is very important for request/reply messages. You have to tell the server program where to send the reply message. Clients and servers have a one-to-many relationship and usually the server program cannot find out from the user data where the request message came from. Therefore, the client provides the *reply-to queu*e and *reply-to queue manager* in the message header. The server uses this information when it performs the MQPUT API call.

- In the *format* field, the sender can specify a value that the receiver can use to decide whether data conversion can be done or not. It is also used to indicate that there is an additional header (extension) present.

- The message also carries information about the sending application (program name and path) and the platform it is running on.

- *Report options* and *feedback* code are used to request information, such as confirmation on arrival or delivery, from the receiving queue manager. For example, the queue manager can send a report message to the sending application when it puts the message in the target queue or when the application gets it off the queue.

- Each time a message is backed out, the *backout counter* is increased. An application can check this counter and act on it, for example, send the message to a different queue where the reason for the backout is analyzed by an administrator.

- Message segmenting and grouping has been mentioned earlier. The queue manager uses the message header to store information about the physical message; for example, if it is a message group, the first or last segment, or which one in between.

About the Queue Manager

The heart of MQSeries is the message queue manager (MQM), MQSeries’ run-time program. Its job is to manage queues and messages for applications. It provides the Message Queuing Interface (MQI) for communication with applications. Application programs invoke functions of the queue manager by issuing API calls. For example, the MQPUT API call puts a message on a

queue to be read by another program using the MQGET API call. This scenario is shown in Figure 4.

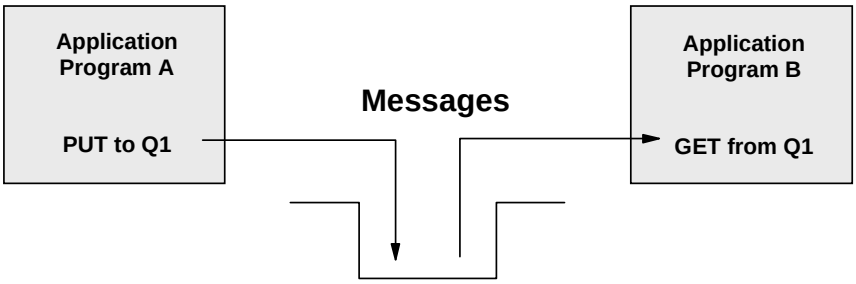


Figure 4. Program-to-Program Communication - One System

A program may send messages to another program that runs in the same machine as the queue manager (shown above), or to a program that runs in a remote system, such as a server or a host. The remote system has its own queue manager with its own queues. This scenario is shown in Figure 5.

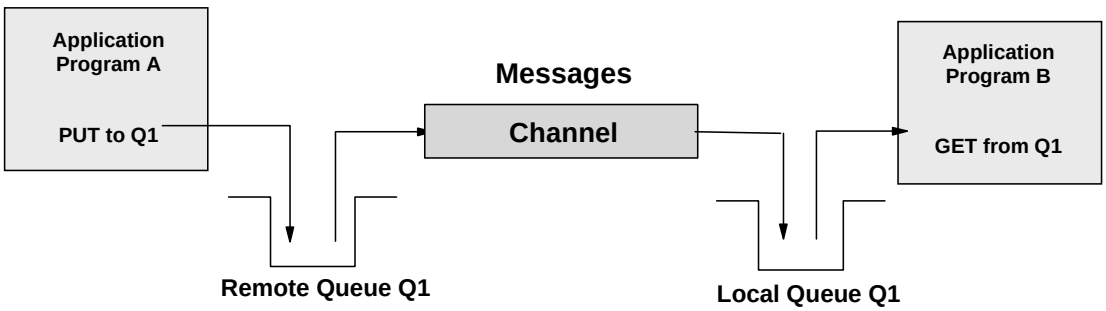


Figure 5. Program-to-Program Communication - Two Systems

The queue manager transfers messages to other queue managers via *channels* using existing network facilities, such as TCP/IP, SNA or SPX. Multiple queue managers can reside in the same machine. They also need channels to communicate.

Application programmers do not need to know where the program to which they are sending messages runs. They put their messages on a queue and let the queue manager worry about the destination machine and how to get the messages there. MQSeries knows what to do when the remote system is not available or the target program is not running or busy.

For the queue manager to do its work, it refers to objects that are defined by an administrator, usually when the queue manager is created or when a new application is added. The objects are described in "About Queue Manager Objects" on page 11. The functions of a queue manager can be summarized as follows:

- It manages queues of messages for application programs.

- It provides an application programming interface, the Message Queue Interface (MQI).
  Note: The Networking Blueprint identifies three communication styles:
  1. Common Programming Interface - Communications (CPI-C)
  2. Remote Procedure Call (RPC)
  3. Message Queue Interface (MQI)

- It uses existing networking facilities to transfer messages to other queue managers when necessary.

- It coordinates updates to databases and queues using a two-phase commit. Gets and puts from/to queues are committed together with SQL updates, or backed out if necessary.

- It segments messages (if necessary) and assembles them. It also can group messages and send them as one physical message to their destination where they are automatically disassembled.

- It can send one message to more than one destination with one API call using a user-defined dynamic distribution list, thus reducing network traffic.

- It provides additional functions that allow administrators to create and delete queues, alter the properties of existing queues, and control the operation of the queue manager. MQSeries for Windows NT Version 5.1 provides graphical user interfaces; other platforms use the command line interface or panels.

MQSeries clients do not have a queue manager in their machines. Client machines connect to a queue manager in a server. The queue manager manages the queues for all clients attached to it.

In contrast to MQSeries clients, each workstation that runs MQSeries for Windows (Version 2) has its own queue manager and queues. MQSeries for Windows is a single-user queue manager and is not intended to function as a queue manager for other MQSeries clients. This product is designed for a mobile environment.

Note: MQSeries for Windows and MQSeries for Windows NT are two different products.

About Queue Manager Clusters

With MQSeries for MVS/ESA and Version 5.1 for distributed platforms, you can join queue managers together in clusters. Queue managers that form a cluster can run in the same machine or in different machines on different platforms. Usually, two of those “cluster queue managers” maintain a repository that contains information about all queue managers and queues in the cluster. This is called a full repository. The other queue managers maintain only a repository of objects they are interested in, a partial repository. The repository allows any queue manager in the cluster to find out about any cluster queue and who owns it. The queue managers use special cluster channels to exchange information.

Clustering also permits multiple instances of a queue (with the same name) on different queue managers. This allows for workload distribution, that is, the queue manager can send messages to different instances of an application.

In normal distributed processing, we send messages to a specific queue owned by a specific queue manager. All messages destined for that queue manager are placed in a transmission queue on the sender's side. This transmission queue has the same name as the destination queue manager. The message channel agents move the messages across the network and place them into the destination queues. Figure 6 shows the relationship between a transmission (Xmit) queue and the target queue manager.

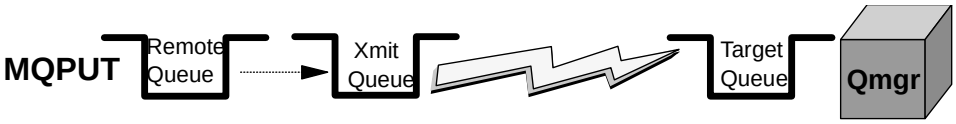


Figure 6. MQPUT to a Remote Queue

With clustering, you send a message to a queue with a specific name somewhere in the cluster, in Figure 7 represented by a cloud. You specify the name of a target queue, not the name of a remote queue definition. Clustering does not require remote queue definitions. They are only useful when you send a message to a queue manager that is not a member of the cluster. You can also specify a queue manager and direct the message to a specific queue, but very often it is left to the queue manager to determine where the queue is (or the queues are) and to which one to send the message.

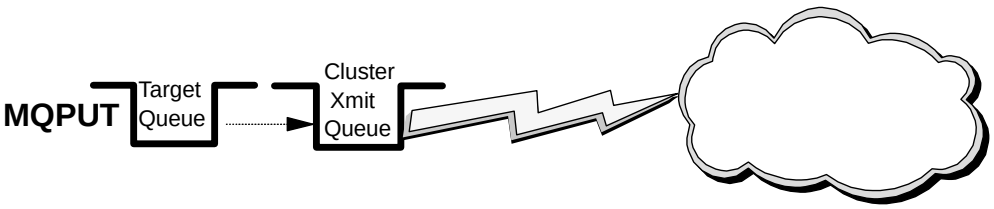


Figure 7. MQPUT to a Cluster Queue

The vision of an MQSeries cluster is as the place where multiple instances of a queue can exist. They come and go as an administrator requires in order to satisfy changing availability and throughput requirements. This has to be achieved completely dynamically and without placing the administrator under a great burden to configure and control. In addition, the programmer does not have to think about multiple queues; he or she just treats them as if writing to a single queue.

This is not to say that there is no burden on the programmer or administrator. Enhanced levels of availability and exploitation of parallelism do require some planning. The administrator or system designer must ensure that there is enough redundancy in the configuration to meet their needs. The application designer must ensure that messages are capable of being processed in multiple places.

You create multiple instances of a queue by defining a queue with the same name on multiple queue managers that belong to the cluster. You must also name the cluster when you define the queue. Without this attribute the queue would only be known locally. When the application specifies only the queue name, where is the message sent?

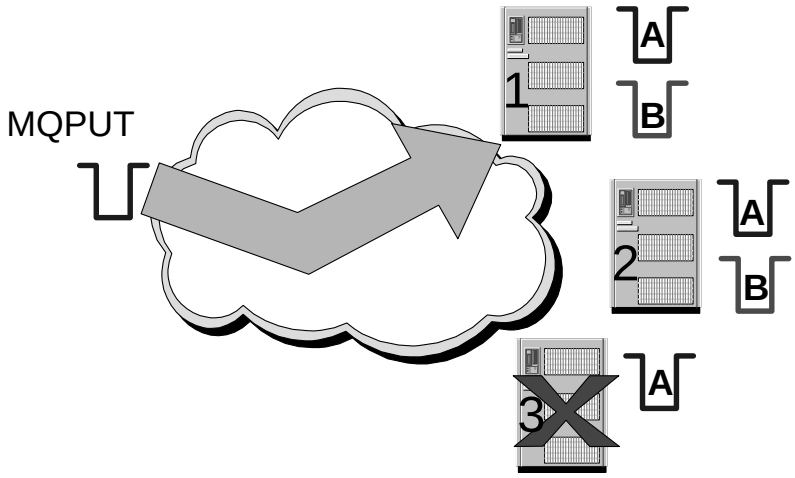


Figure 8. Accessing Cluster Queues

Figure 8 gives you an idea. MQSeries distributes the messages round-robin. You can, however, change this default action by writing your own workload balancing exit routine.

Figure 8 shows messages put in one of the three cluster queues named A. Each of the three queue managers on the right owns a queue with this name. By default, the first message is placed in queue A on queue manager 1, the next in queue A on queue manager 2, the third goes to queue manager 3 and the fourth message to the queue on queue manager 1 again.

In another scenario involving queue B, we notice that the third queue manager is down and the third instance of queue B is not available. The sending queue manager becomes aware of this problem because it subscribed to information about all queue manager and queues it is interested in, that is, where it sends messages. As soon as it finds out that there is a problem with the third instance of B, it distributes messages to the first two instances only. Special messages about changes of the status of cluster objects are instantly published to all queue managers that subscribed to that object.

About Queue Manager Objects

This section introduces you to queue manager objects, such as queues and channels. The queue manager itself is an object, too. Usually, an administrator creates one or more queue managers and their objects. A queue manager can use objects of the following types:

1. Queues
2. Process definitions
3. Channels

The objects are common across different MQSeries platforms. There are other objects that apply to MVS systems only, such as the buffer pool, PSID, and storage class. AS/400 MQ objects are known to the OS/400 operating system as object type *USRSPC (user space) in the QMQMDATA library.

Queues

Message queues are used to store messages sent by programs. There are local queues that are owned by the local queue manager, and remote queues that belong to a different queue manager. Queues are described in more detail in the section “About Message Queues” on page 13.

Channels

A channel is a logical communication link. In MQSeries, there are two different kinds of channels:

1. *Message channels*
   A message channel connects two queue managers via message channel agents (MCAs). Such a channel is unidirectional. It comprises two message channel agents, a sender and a receiver, and a communication protocol. An MCA is a program that transfers messages from a transmission queue to a communication link, and from a communication link into the target queue. For bidirectional communication you have to define two channel pairs consisting of a sender and a receiver. Message channel agents are also referred to as movers.

2. *MQI channe*ls
   A Message Queue Interface (MQI) channel connects an MQSeries client to a queue manager in its server machine. Clients don’t have a queue manager of their own. An MQI channel is bidirectional.

Figure 9 shows both channels types. You see four machines, two clients connected to their server machine via MQI channels, and the server connected to another server or a host via two unidirectional message channels. Some channels can be defined automatically by MQSeries. There are different types of message channels, depending on how the session between the queue managers is initiated and for what purpose they are used.

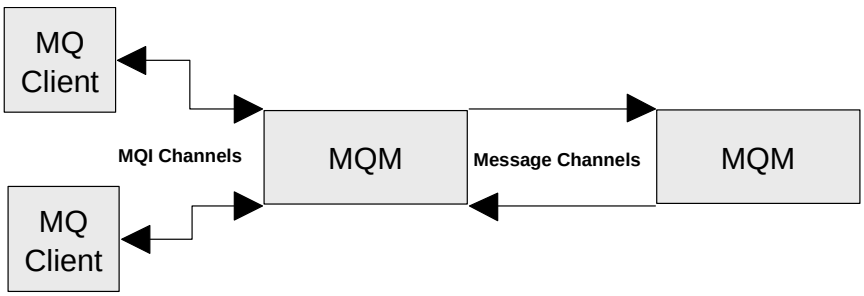


Figure 9. MQSeries Channels

To transmit non-persistent messages, a message channel can run at *two speeds*: fast and normal. Fast channels improve performance, but (non-persistent) messages can be lost in case of a channel failure.

A channel can use the following transport types: SNA LU 6.2, TCP/IP, NetBIOS, SPX and DEC Net. Not all are supported on all platforms.

MQSeries for Windows Version 2 uses message channels to connect to other machines. Since this product is designed as a single user system, it does not support MQI channels. This product supports only TCP/IP.

Process Definitions

A process definition object defines an application to a queue manager. For example, it contains the name of the program (and its path) to be triggered when a message arrives for it.

About Message Queues

Queues are defined as objects belonging to a queue manager. MQSeries knows a number of different queue types, each with a specific purpose. The queues you use are located either in your machine and belong to the queue manager to which you are connected, or in your server (if you are a client). Figure 10 lists different queue types and their purposes. More detailed information is below.

Local queue	is a real queue
Remote queue	structure describing a queue
Transmission queue (xmitq)	local queue with special purpose
Initiation queue	local queue with special purpose
Dynamic queue	local queue created "on the fly"
Alias queue	if you don't like the name
Dead-letter queue	one for each queue manager
Reply-to-queue	specified in request message
Model queue	model for local queues
Repository queue	holds cluster information

Figure 10. Queue Types

Local Queue

A queue is local if it is owned by the queue manager to which the application program is connected. It is used to store messages for programs that use the same queue manager. For example, program A and program B each has a queue for incoming messages and another queue for outgoing messages. Since the queue manager serves both programs, all four queues are local.

Note: Both programs do not have to run in the same workstation. Client workstations usually use a queue manager in a server machine.

Cluster Queue

A cluster queue is a local queue that is known throughout a cluster of queue managers, that is, any queue manager that belongs to the cluster can send messages to it without the need of a remote definition or defining channels to the queue manager that owns it.

Remote Queue

A queue is “remote” if it is owned by a different queue manager. A remote queue definition is the local definition of a remote queue. A remote queue is not a real queue. It is a structure that contains some of the characteristics of a queue hosted by a different queue manager.

The application programmer can use the name of a remote queue just as he or she can use the name of a local queue. The MQSeries administrator defines where the queue actually is. Remote queues are associated with a transmission queue.

Notes: - A program cannot read messages from a remote queue.
- You don’t need a remote queue definition for a cluster queue.

Transmission Queue

This is a local queue with a special purpose. A remote queue is associated with a transmission queue. Transmission queues are used as an intermediate step when sending messages to queues that are owned by a different queue manager.

Typically, there is only one transmission queue for each remote queue manager (or machine). All messages written to queues owned by a remote queue manager are actually written to the transmission queue for this remote queue manager. The messages will then be read from the transmission queue and sent to the remote queue manager.

Using MQSeries clusters, there is only one transmission queue for all messages sent to all other queue managers in the cluster.

Transmission queues are transparent to the application. They are used internally by the queue manager. When a program opens a remote queue, the attributes of the queue are obtained from the transmission queue. Therefore, the results of a program writing messages to a queue will be affected by the transmission queue characteristics.

Dynamic Queue

Such a queue is defined "on the fly" when the application needs it. Dynamic queues may be retained by the queue manager or automatically deleted when the application program ends. Dynamic queues are local queues. They are often used in conversational applications, to store intermediate results. Dynamic queues can be:

- Temporary queues that do not survive queue manager restarts
- Permanent queues that do survive queue manager restarts

Alias Queue

Alias queues are not real queues but definitions. They are used to assign different names to the same physical queue. This allows multiple programs to work with the same queue, accessing it under different names and with different attributes.

Model Queue

A model queue is not a real queue. It is a collection of attributes that are used when a dynamic queue is created.

Initiation Queue

An initiation queue is a local queue to which the queue manager writes a trigger message when certain conditions are met on another local queue, for example, when a message is put into an empty message queue or in a transmission queue. Such a trigger message is transparent to the programmer. Two MQSeries applications monitor initiation queues and read trigger messages, the trigger monitor which starts applications and the channel initiator which starts the transmission between queue managers.

Note: Applications do not need to be aware of initiation queues, but the triggering mechanism implemented through them is a powerful tool to design and write asynchronous applications.

Reply-to-Queue

A request message must contain the name of the queue into which the responding program must put the reply message. This can be considered the “return address”. The name of this queue together with the name of the queue manager that owns it is stored in the message header. This is the responsibility of the application program.

Dead-Letter Queue

A queue manager must be able to handle situations when it cannot deliver a message. Here are some examples:

- The destination queue is full.
- The destination queue does not exist.
- Message puts have been inhibited on the destination queue.
- The sender is not authorized to use the destination queue.
- The message is too large.
- The message contains a duplicate message sequence number.

When the above conditions are met, the messages are written to the dead-letter queue. Such a queue is defined when the queue manager is created. It will be used as a repository for all messages that cannot be delivered.

Repository Queue

Repository queues have existed since Version 5.1 and Version 2.1 for OS/390. They are used in conjunction with clustering and hold either a full or a partial repository of queue managers and queue manager objects in a cluster (or group) of queue managers.

Creating a Queue Manager

You may create as many queue managers as you like and have them running at the same time. You create a queue manager with the command `crtmqm`; to make it the default, specify the parameter /q.
The following command creates the default queue manager MYQMGR (in a Windows NT environment):

```
crtmqm /q MYQMGR
```

Note: Queue manager names are case-sensitive.

There are default definitions for objects every queue manager needs, such as model queues. These objects are created automatically. Most certainly, you will have to create other objects that pertain to the applications you run. Usually, those application specific objects are kept in a script file, such as mydefs.in. You apply them to a newly created queue manager with the command:

```
runmqsc < mydefs.in
```

MQSeries for Windows NT Version 5.1 provides a graphical user interface to create and manipulate queue managers and their objects.

A dead-letter queue is not automatically created. To create one when you create the queue manager, specify it as shown in the following example:

```
crtmqm /q /u system.dead.letter.queue MYQMGR
```

To start the queue manager issue the command:

```
strmqm
```

Manipulating Queue Manager Objects

MQSeries for distributed platforms provides the utility RUNMQSC to create and delete queue manager objects and to manipulate them. The queue manager must be running when you use the utility. RUNMQSC works in two ways:

- You can type the commands.
- You can create a file containing a list of commands and use this file as input.

The commands in Figure 11 on page 17 start the default queue manager (which is already running, as the response indicates) and create the local queue QUEUE1 for it. Another command alters the queue manager properties to define a dead-letter queue.

To start the utility in an interactive mode, type `runmqsc`. To end it, type `end`. Another way to create MQSeries objects is by using an input file instead of typing the commands; for example:

```
runmqsc < mydefs.in > a.a
```

where mydefs.in is the script file that contains the commands and a.a is the file that will contain the responses from the RUNMQSC utility, so that you can check if any error occurred. The output can either appear in the window or can be redirected to a file.

```
C:\strmqm
MQSeries queue manager running.

runmqsc
84H2001,6539-B42 (C) Copyright IBM Corp. 1994, 1997. ALL RIGHTS RESERVED
Starting MQSeries Commands.

define qlocal('QUEUE1') replace descr ('test queue')
     1 : define qlocal('QUEUE1') replace descr ('test queue')
AMQ8006: MQSeries queue created.
alter qmgr deadq(system.dead.letter.queue)
     2 : alter qmgr deadq(system.dead.letter.queue)
AMQ8005: MQSeries queue manager changed.
end
     3 : end
2 MQSC commands read.
0 commands have a syntax error.
0 commands cannot be processed.

C:\
```

Figure 11. Manipulating Objects Using Control Commands

Clients and Servers

MQSeries distinguishes clients and servers. Before you install MQSeries on a distributed platform you have to decide if the workstation will be an MQSeries client, an MQSeries server, or both. With MQSeries for Windows a new term was introduced, the leaf node (described later). There are two kinds of clients:

- Slim client or MQSeries client
- Fat client

Fat clients have a local queue manager; slim clients don't.

When a slim client cannot connect to its server it cannot work, because the queue manager and queues for a slim client reside in the server. Usually, an MQSeries client is a slim client. Several of these clients share MQSeries objects, and the queue manager is one of them, in the server to which they are attached.

Note: The MQSeries Client for Java is a slim client.

In some cases it may be advantageous to have queues in the end user's workstation, especially in a mobile environment. That allows you to run your application when a connection between

client and server does not (temporarily) exist.

You may install client and server software in the same system and use it as an end user's workstation. If your operating system is Windows NT you can install MQSeries for Windows NT V5.1 or MQSeries for Windows V2.1 (also called MQWin). If your operating system is Windows 95 use MQWin V2.1. This product has been designed for end users and uses fewer resources.

The difference between an end user's workstation that is a client and one that has a queue manager is the way messages are sent. The queues reside either in the end user's workstation or in the server.

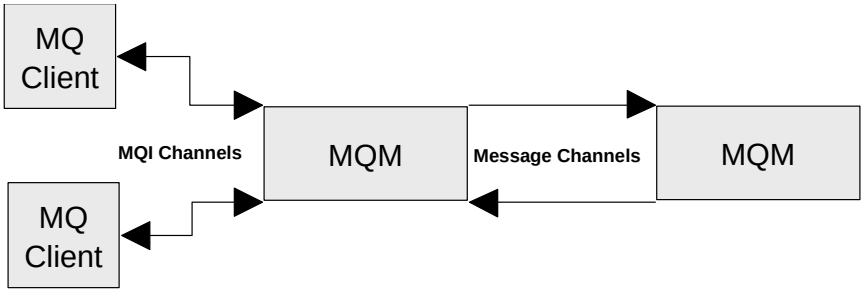


Figure 12. MQI and Message Channels

Figure 12 shows again the use of MQI and message channels.

- MQI channels connect clients to a queue manager in a server machine. All MQSeries objects for the client reside in the server. MQI channels are faster than message channels.

- A message channel connects a queue manager to another queue manager. The queue manager can reside in the same or in a different machine.

The following summarizes the three workstation types:

MQSeries Client

A client workstation does not have a queue manager of its own. It shares a queue manager in a server with other clients. All MQSeries objects, such as queues, are in the server. Since the connection between client and server is synchronous, the application cannot work when the communication is broken. You could refer to such workstations as "slim" clients.

MQSeries Server

A workstation can be a client and a server. A server is an intermediate node between other nodes. It serves clients that have no queue manager and manages the message flow between its clients, itself and other servers. In addition to the server software you may install the client software, too. This configuration is used in an application development environment.

Leaf Node

MQSeries for Windows was designed for use by a single user. It has its own "small footprint" queue manager with its own objects. However, it is not an intermediate node between other nodes. It is called a leaf node. You could also refer to it as a "fat" client. This product is able to queue outbound messages when connection to a server or host is not available, and inbound messages when the appropriate application is not active.

How MQSeries Works

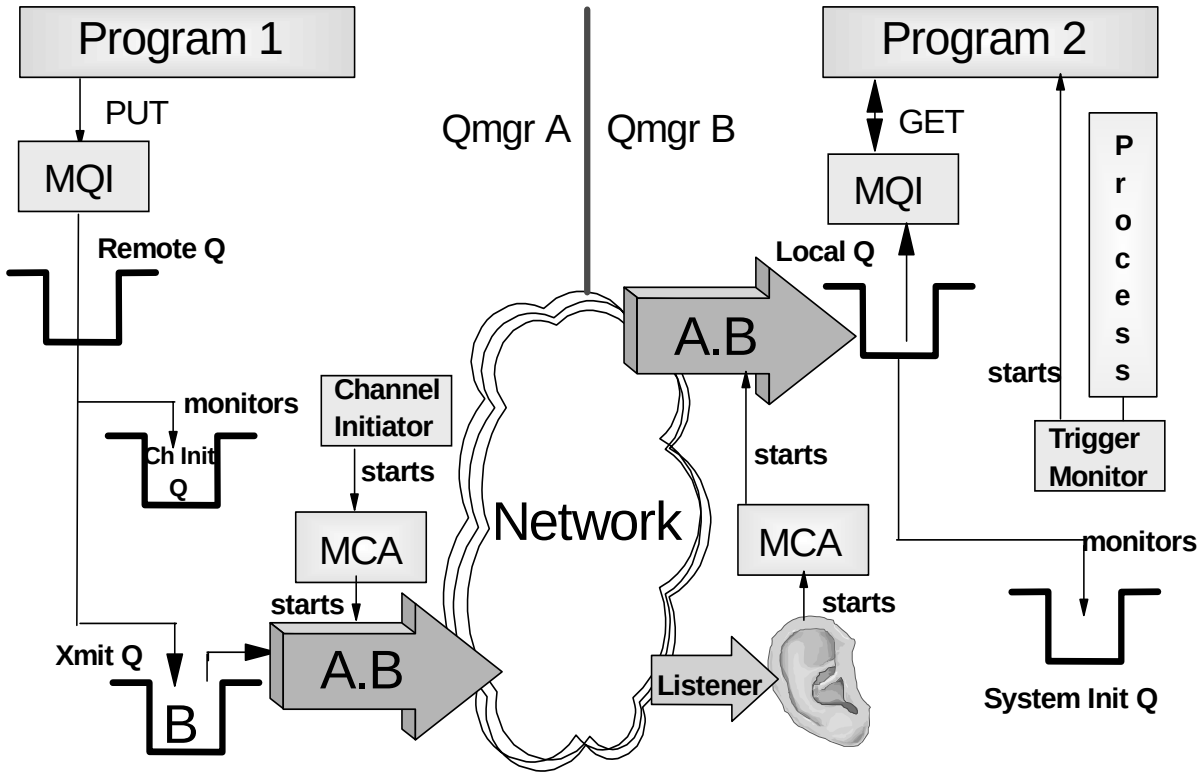


Figure 13. MQSeries - Parts and Logic

Figure 13 shows the parts and architecture of MQSeries. The application program uses the Message Queue Interface (MQI) to communicate with the queue manager. The MQI is described in more detail later. The queuing system consists of the following parts:

- Queue Manager (MQM)
- Listener
- Trigger Monitor
- Channel Initiator
- Message Channel Agent (MCA) or mover

When the application program wants to put a message on a queue it issues an MQPUT API call. This invokes the MQI. The queue manager checks whether the queue referenced in the MQPUT is local or remote. If it is a remote queue, the message is placed into the transmission (xmit) queue. The queue manager adds a header that contains information from the remote queue definition, such as destination queue manager name and destination queue name.

Note: Each remote queue must be associated with an xmit queue. Usually, all messages destined for one remote machine use the same xmit queue.

Transmission is done via channels. Channels can be started manually or automatically. To start a channel automatically, the xmit queue must be associated with a channel initiation queue. Figure 13 on page 19 shows that the queue manager puts a message into the xmit queue and another message into the channel initiation queue. This queue is monitored by the *channel initiator.*

The channel initiator is an MQSeries program that must be running in order to monitor initiation queues. When the channel initiator detects a message in the initiation queue, it starts the message channel agent (MCA) for the particular channel. This program moves the message over the network to the other machine, using the sender part of the unidirectional message channel pair.

On the receiving end, a *listener* program must have been started. The listener, also supplied with MQSeries, monitors a specified port, by default, the port dedicated to MQSeries, 1414. When a message arrives, it starts the *message channel agent*. The MCA moves the message into the specified local queue using the receiver part of the message channel pair.

Note: Both channel definitions, sender and receiver, must have the same name. For the reply, you need another message channel pair.

The program that processes the incoming message can be started manually or automatically. To start the program automatically, an initiation queue and a process must be associated with the local queue, and the *trigger monitor* must be running.

When the program starts automatically, the MCA puts the incoming message into the local queue and a trigger message into the initiation queue. This queue is monitored by the trigger monitor. This program invokes the application program specified in the process definition. The application issues an MQGET API call to retrieve the message from the local queue.

Communication between Queue Managers

In this section, we discuss what you have to define to send messages to a queue manager that resides in another system. We use message channels for communication between queue managers as shown in Figure 12 on page 18.

The logic is illustrated in Figure 14 on page 22 and the necessary MQSeries definitions are shown in Figure 15 on page 22.

Each machine has a queue manager installed and each queue manager manages several local queues. Messages destined for a remote queue manager are put into a *remote queue*. A remote queue is not a real queue; it is the definition of a local queue in the remote machine. A remote queue is associated with a transmission (xmit) queue, which is a local queue. Usually, there is one xmit queue for each remote queue manager.

A transmission queue is associated with a message channel. Message channels are unidirectional, meaning that you have to define two channels for a conversational type of communication. Also, you have to define each channel twice, once in the system that sends the message (sender channel) and once in the system that receives the message (receiver channel). Each channel pair (sender and receiver) must have the same name. This scenario is elucidated in Figure 14 on page 22. Next, let us find out how we get this to work.

How to Define a Connection between Two Systems

Figure 14 on page 22 shows the required MQSeries objects for connecting two queue managers. In each system we need:

- A remote queue definition that mirrors the local queue in the receiver machine and links to a transmission queue (Q1 in system A and Q2 in system B).

- A transmission queue that holds all messages destined for the remote system until the channel transmits them (QMB in system A and QMA in system B).

- A sender channel that gets messages from the xmit queue and transmits them to the other system using the existing network (QMA.QMB in system A and QMB.QM.A in system B).

- A receiver channel that receives messages and puts them into a local queue (QMB.QMA in system A and QMA.QMB in system B); receiver channels can be started automatically by the queue manager when Channel Auto Definition (CHAD) is enabled.

- A local queue from which the program gets its messages (Q2 in system A and Q1 in system B).

In each system, you must define the appropriate queue manager objects. The objects are defined in the two script files shown in Figure 15 on page 22.

Notes:

When you use clustering you don’t have to define transmission queues. There is only one transmission queue per queue manager, and that is created automatically when the queue manager is created.

You also don’t have to define channels, neither sender or receiver channels; they are automatically created when needed.

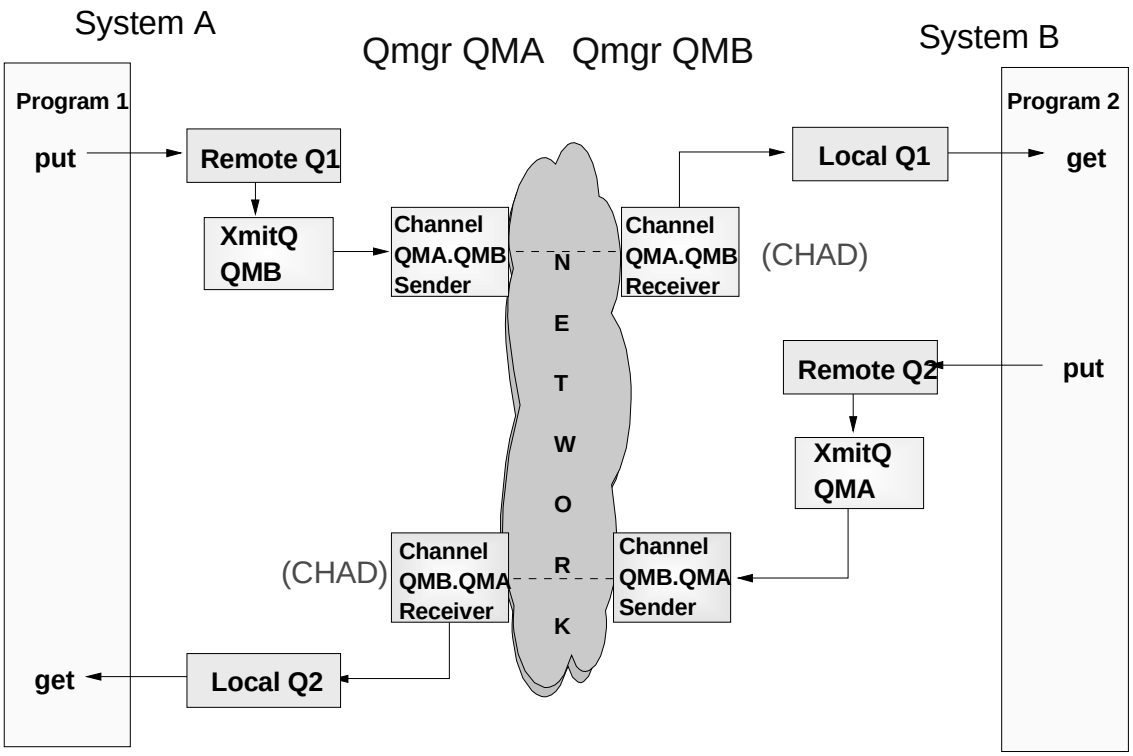


Figure 14. Communication between Two Queue Managers

System A (QMA)	System B (QMB)
DEFINE QREMOTE(Q1) + RNAME(Q1) RQMNAME(QMB) + XMITQ(QMB)	DEFINE QLOCAL(Q1)
DEFINE QLOCAL(QMB) + USAGE(xmitq)	
DEFINE CHANNEL(QMA.QMB) + CHLTYPE(sdr) + XMITQ(QMB) + TRPTYPE(tcp) + CONNAME(9.24.104.123)	DEFINE CHANNEL(QMA.QMB) + CHLTYPE(rcvr) + TRPTYPE(tcp)
DEFINE QLOCAL(Q2)	DEFINE QREMOTE(Q2) + RNAME(Q2) RQMNAME(QMA) + XMITQ(QMA)
	DEFINE QLOCAL(QMA) + USAGE(xmitq)
DEFINE CHANNEL(QMB.QMA) + CHLTYPE(rcvr) + TRPTYPE(tcp)	DEFINE CHANNEL(QMB.QMA) + CHLTYPE(sdr) + XMITQ(QMA) + TRPTYPE(tcp) CONNAME(ABC1)

Figure 15. MQSeries Objects Defining Connection between Two Queue Managers

How to Start Communication Manually

First, the objects have to be known to the queue managers. You use RUNMQSC to create the objects. Make sure that the queue manager is running. Next, start the listeners and the channels. You need to start only the sender channel in each system. MQSeries starts the receiver channel. The commands to start listener and channel for queue manager QMA are:

```
strmqm QMA
start runmqlsr -t tcp -m QMA -p 1414
runmqsc
start channel (QMA.QMB)
end
```

With the first command you start queue manager QMA. The next command starts the listener. It listens on behalf of QMA on port 1414. As transmission protocol TCP/IP is used. The third command starts runmqsc in interactive mode. The channel QMA.QMB is started under control of runmqsc. For the other queue manager you issue equivalent commands. You also have to start the applications in both systems.

How to Start Communication Automatically

You can use the channel initiator to start channels. Instead of the commands shown above enter the following commands (for Windows NT, UNIX and OS/2):

```
start runmqlsr -t tcp -m QMA -p 1414
start runmqchi
```

With the first command you start the listener and with the second the channel initiator program. The channel initiator monitors a channel initiation queue and starts the proper channel to read in the message. The default initiation queue is SYSTEM.CHANNEL.INITQ.

You may also start the channel initiator from RUNMQSC (Windows NT, UNIX and OS/2). The command is:

```
start chinit
```
--OR--
```
start chinit initq(SYSTEM.CHANNEL.INITQ)
```

To have the transmission queue triggered, add three more parameters (below shown in bold):

```
DEFINE QLOCAL(A.TO.B) REPLACE +
       USAGE(xmitq) +
       TRIGGER
       TRIGTYPE(every) +
       INITQ(SYSTEM.CHANNEL.INITQ) +
       DESCR('Xmit Queue')
```

The queue manager can trigger the process that starts the channel program in three ways:

- When the first message is put into the transmission queue
- Every time a message is put into the xmit queue
- When the queue contains a specified number of messages

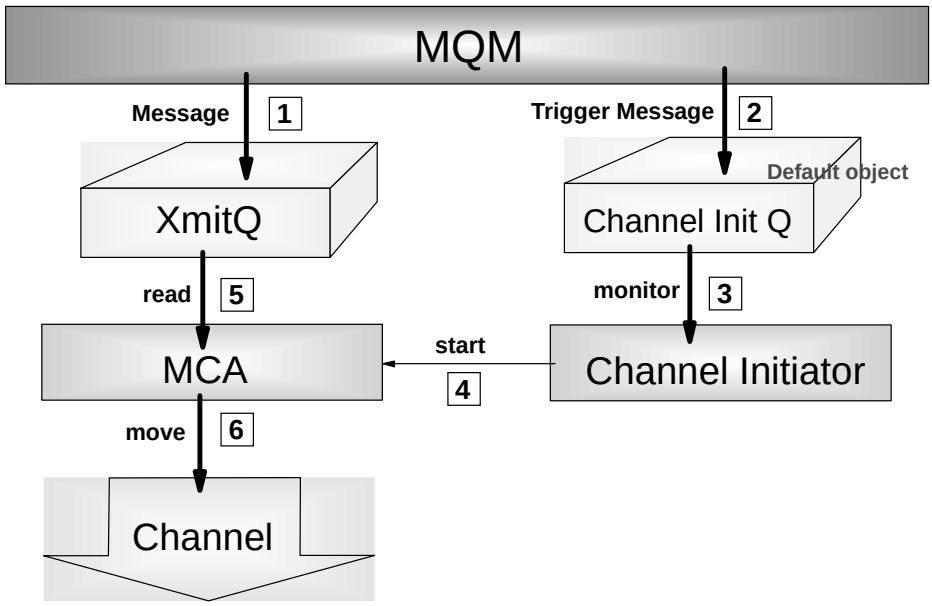


Figure 16. Triggering Channels

Figure 16 shows the logic behind triggering:

1. The program issues an MQPUT to a remote queue and a message is placed into the transmission queue.

2. When the queue manager puts a message into the transmission queue, it checks the trigger type specified in the queue definition. Depending on that definition and on how many messages are in the queue, it may put an additional message in the channel initiation queue. This “trigger message” is transparent to the user.

3. Since the channel initiator was started earlier, for example, at boot time, it monitors the channel initiation queue and removes the trigger message.

4. The channel initiator starts the message channel agent (also called mover).

5. The channel program gets the message off the transmission queue and invokes any channel exit routines, if specified.

6. The message is then moved over the network to its destination.

How to Trigger Applications

This section describes how to trigger an application program that runs in the server machine. Both triggering and triggered applications can run under the same or different queue managers.

Note: MQSeries for Windows V2.1 does not support triggering.

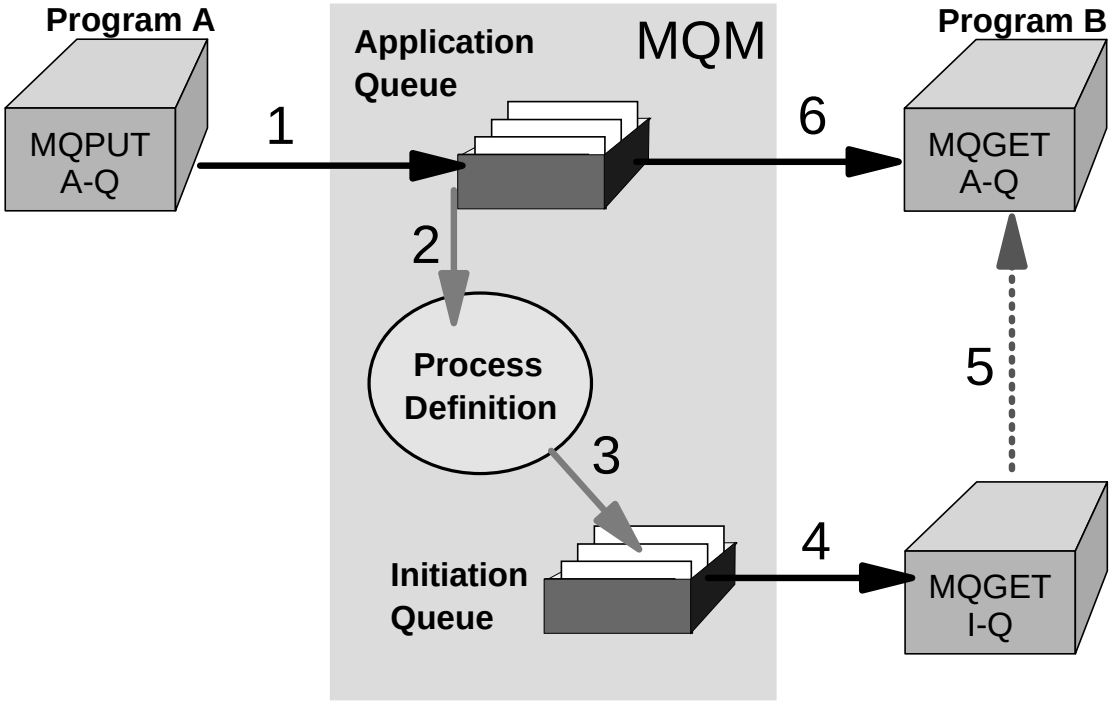


Figure 17. Triggering an Application

Figure 17 shows the logic of triggering. Here Program A sends a message to A-Q to be processed by Program B. The MQSeries triggering mechanism is as follows:

1. Program A issues an MQPUT and puts a message into A-Q for Program B.

2. The queue manager processes this API call and puts the message into the application queue.

3. It also finds out that the queue is triggered. It creates a trigger message and looks in the Process Definition to find the name of the application and puts it in the trigger message. The trigger message is put into the initiation queue.

4. The trigger monitor gets the trigger message from the initiation queue and starts the program specified.

5. The application program starts running and issues an MQGET to retrieve the message from the application queue.

The definitions necessary to trigger an application are as follows:

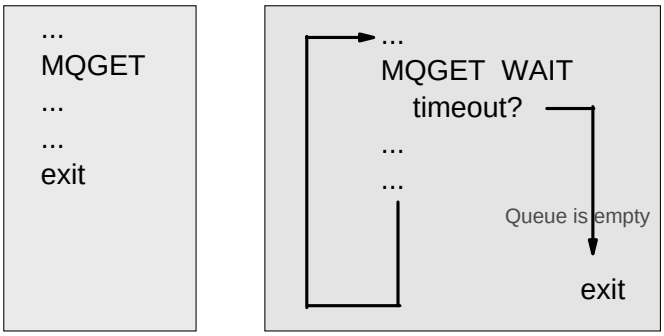
- The target queue must have “triggering” specified as shown in bold below:

```
DEFINE QLOCAL(A-Q)  REPLACE +
       TRIGGER
       TRIGTYPE(first) +
       INITQ(SYSTEM.DEFAULT.INITIATION.QUEUE ) +
       PROCESS(proc1)
       DESCR('This is a triggered queue')
```

- The process definition associated with the target queue can be this:

```
DEFINE PROCESS(proc1)  REPLACE +
       DESCR('Process to start server program') +
       APPLTYPE(WINDOWSNT) +
       APPPLICID('c:\test\myprog.exe')
```

What trigger type to use depends on how the application is written. You have three choices:



- EVERY Every time a message is put in the target queue a trigger message is also put in the initiation queue. Use this when your program exits after processing one message or transaction, as shown above on the left.

- FIRST A trigger message is put in the initiation queue only when the target queue has been empty. Use this when the program exits only then when there are no more messages in the queue, as shown on the right.

- n messages A trigger message is put in the initiation queue when there are n messages in the target queue. For example, you can start a batch program when the queue holds 1000 messages.

Communication between Client and Server

Below we discuss what you have to do to define and test the connection between an MQ client and its MQ server. A more detailed description is provided in the publication *MQSeries Clients* , GC33-1632.

How to Define a Client/Server Connection

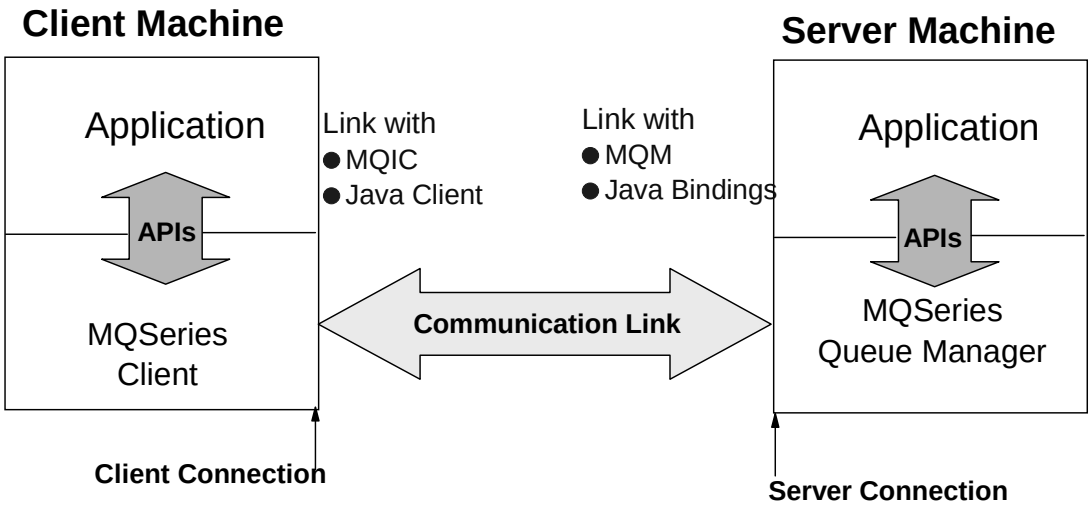


Figure 18. Client/Server Connection

Figure 18 shows that the MQSeries Client product is installed in the client machine. We said before that clients and servers are connected with MQI channels. An MQI channel consists of a sender/receiver pair, called Client Connection (CLNTCONN) and Server Connection (SVCONN) channel.

You have to know what transmission protocol is used (for example, TCP/IP), the port the listener listens to (1414 is the default), and the address of the systems to which you want to connect. For an address you can specify an LU name, a host name or machine name, or a TCP/IP address.

The client connection channel is defined as an environment variable, such as:

set MQSERVER=CHAN1/TCP/9.24.104.206(1414)

where:

- MQSERVER is the name of the environment variable.
- CHAN1 is the name of the channel to be used for communication between client and server. This channel is defined in the server. MQSeries will automatically create it should it not exist.
- TCP denotes that TCP/IP is to be used to connect to the machine with the address following the parameter.
- 1414 is the default port number for MQSeries. You may omit this parameter if the listener on the server side uses this default, too.

The definition of the server is as follows:

```
DEFINE CHANNEL('CHAN1') CHLTYPE(SVRCONN) REPLACE +
       TRPTYPE(TCP) MCAUSER(' ')
```

For the MQSeries Client for Java, the environment variables are set in the applet code. An applet can run in any machine, such as a network station, and it has no access to environment variables. The example below shows what statements to include in your Java program:

```
import com.ibm.mq.*;

MQEnvironment.hostname = "9.24.104.456";
MQEnvironment.channel  = "CHAN1";
MQEnvironment.port     = 1414;
```

How a Client/Server Connection Works

Now we describe how to trigger an application program that runs in the server machine. Since there are MQI channels of the type server connection between clients and server, all clients use the queue manager in the server machine. When a client puts a message on a queue it has to be read and processed by a program. This program can be started when the server starts or the queue manager can start it when needed by using the MQSeries triggering mechanism.

Figure 19 on page 29 shows two clients connected to a server. Both clients request services from the same program (Appl S1). Since that application runs in the same system as the queue manager, we have only local queues. Some queues are specifically for a particular client, for example, QA1 is the reply queue for client A and QA2 is the reply queue for client B. Other queues are used by both clients and server. For example, QS1 is used as output queue for both clients and as input queue for the server program.

Next, we describe the MQSeries objects and API call sequences in both client and server.

How a Client Sends a Request

The client starts a program that puts a message on a queue. For this function five MQSeries API calls are executed:

- MQCONN to connect to the queue manager in the server
- MQOPEN to open the message queue QS1 for output
- MQPUT to put a message in the queue
- MQCLOSE to close the queue QS1
- MQDISC to disconnect from the queue manager

Of course, the program can put many messages in the queue before it closes it and disconnects. Closing the queue and disconnecting from the queue manager can be done when the application ends because there are no more messages to process.
The MQSeries client code that runs in the client machine processes the API calls and routes them to the machine defined in the environment variable.

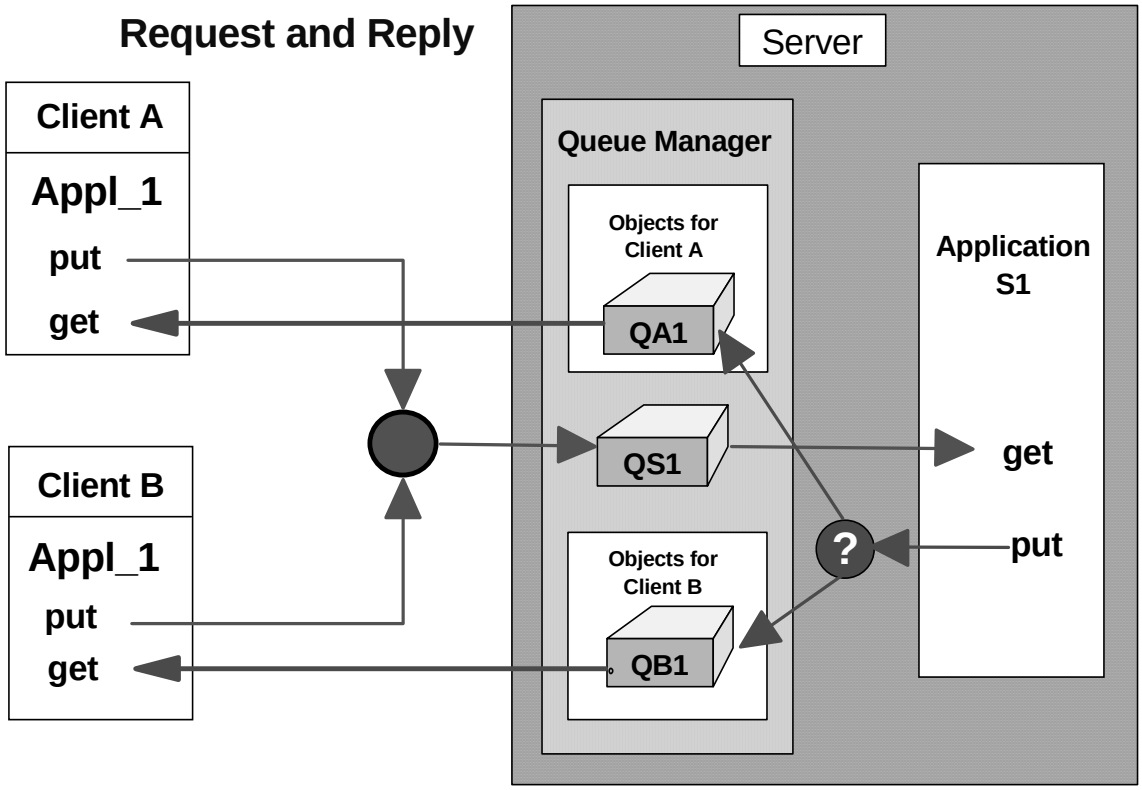


Figure 19. Clients and Server Communicating

How the Server Receives a Request

In the server machine, the following queue manager objects are needed:

- A channel of the type server connection.
- A local queue, QS1, into which the clients put their messages.
- An initiation queue into which the queue manager puts a trigger message when a request for queue QS1 arrives. You can use the default initiation queue.
- A process definition that contains the name of the program to be started when the trigger event occurs (S1).
- One or more queues in which the program puts the reply messages (QA1 and QB1).

In the server machine, two programs have to be started: the listener and the trigger monitor. The listener listens for messages on the channel and puts them on the queue QS1. Since QS1 is triggered, the MQM puts a trigger message on the trigger queue each time a message is put on QS1. When a message is placed on the trigger queue, the trigger monitor starts the program defined in the process.

The server program S1 connects to the queue manager, opens the queue QS1 and issues an MQGET to read the message.

How the Server Sends a Reply

After processing a request the server puts the reply in the reply queue for the client. To do this it has to open the output queue (QA1 or QB1) and issue an MQPUT.

Since several clients use the same server application, it is advisable to give the server a "return address," that is, the names of the queue and the queue manager that will receive the reply message. These fields are in the header of the request message, containing the reply-to-queue manager and reply-to-queue (here, QA1 or QB1). It is the responsibility of the client program to specify these values.

Usually, the server program stays active and waits for more messages, at least for a certain time. For how long can be specified in the wait option of the MQGET API.

How the Client Receives a Reply

The client program knows the name of its input queue, here QA1 or QB1. The application can use two modes of communication:

- *Conversational*
  If the application uses this mode of communication with the server program, it waits for the message to arrive before it continues processing. This means, the reply queue is open and an MQGET with wait option has been issued.

  The client application must be able to deal with two possibilities:

  - The message arrives in time.
  - The timer expires and no message is there.

- *True asynchronous*
  When using this mode, the client does not care when the request message arrives. Usually, the user clicks a push button in a menu window to activate a program that checks the reply queue for messages. If a message is present, this or another program can process the reply.

The Message Queuing Interface (MQI)

A program talks directly to its local queue manager. It resides in the same processor or domain (for clients) as the program itself. The program uses the Message Queuing Interface (MQI). The MQI is a set of API calls that request services from the queue manager.

Note: When the connection between a client and its server is broken, no API calls can be executed, since all objects reside in the server.

There are 13 APIs. They are shown in Figure 20 on page 31.

MQCONN	Connect to a queue manager
MQDISC	Disconnect from a queue manager
MQOPEN	Open a specific queue
MQCLOSE	Close a queue
MQPUT	Put a message on a queue
MQGET	Get a message from a queue
MQPUT1	MQOPEN + MQPUT + MQCLOSE
MQINQ	Inquire properties of an object
MQSET	Set properties of an object
MQCONNX	Standard or fastpath bindings
MQBEGIN	Begin a unit of work (database coordination)
MQCMIT	Commit a unit of work
MQBACK	Back out

Figure 20. MQSeries APIs

The most important ones are MQPUT and MQGET. The other calls are used less frequently. Comments regarding several APIs follow:

MQCONN establishes a connection with a queue manager using the standard bindings.

MQCONNX establishes a connection with a queue manager using fastpath bindings. Fastpath puts and gets are faster, but the application must be well behaved, that is, well tested. Application and queue manager run in the same process. When the application crashes it takes the queue manager down with it. This API call is new in MQSeries Version 5.

MQBEGIN begins a unit of work that is coordinated by the queue manager and that may involve external XA-compliant resource managers. This API has been introduced with MQSeries Version 5. It is used to coordinate transactions that use queues (MQPUT and MQGET under syncpoint) and database updates (SQL commands).

MQPUT1 opens a queue, puts a message on it and closes the queue. This is a combination of MQOPEN, MQPUT and MQCLOSE.

MQINQ requests information about the queue manager or one of its objects, such as the number of messages in a queue.

MQSET changes some attributes of an object.

MQCMIT specifies that a syncpoint has been reached. Messages put as part of a unit of work are made available to other applications. Messages retrieved as part of a unit of work are deleted.

MQBACK tells the queue manager to back out all message puts and gets that have occurred since the last syncpoint. Messages put as part of a unit of work are deleted. Messages retrieved as part of a unit of work are reinstated on the queue.

Notes:
- MQDISC implies the commit of a unit of work. Ending the program without disconnecting from the queue manager causes a rollback (MQBACK).
- MQSeries for AS/400 does not use MQBEGIN, MQCMIT or MQBACK. The commit control operation codes of the AS/400 language are used.

A Code Fragment

The code fragment below shows the APIs to put a message on one queue and get the reply from another queue.

Note: The fields CompCode and Reason will contain completion codes for the APIs. You can find them in the Application Programming Reference

Comments:

1 This statement connects the application to the queue manager with the name MYQMGR. If the parameter QMName does not contain a name, then the default queue manager is used. MQ stores the handle of the queue manager in the variable HCon. This handle must be used in all subsequent APIs.

2 To open a queue the queue name must be moved into the object descriptor that will be used for that queue. This statement opens QUEUE1 for output only (open option MQOO_OUTPUT). The handle to the queue and values in the object descriptor are returned. The handle Hobj1 must be specified in the MQPUT.

3 MQPUT places the message assembled in a buffer on a queue. Parameters for MQPUT are:
•The handle of the queue manager (from MQCONN)
•The handle of the queue (from MQOPEN)
•The message descriptor
•A structure containing options for the put (refer to the Application Programming Reference)
•The message length
•The buffer containing the data

4 This statement closes the output queue. Since the queue is predefined no close processing takes place (MQOC_NONE).

5 This statement opens QUEUE2 for input only using the queue-defined defaults. You could also open a queue for browsing, meaning that the message will not be removed.

```
MQHCONN  HCon;                  // Connection handle
MQHOBJ   HObj1;                 // Object handle for queue 1
MQHOBJ   HObj2;                 // Object handle for queue 2
MQLONG   CompCode, Reason;      // Return codes
MQLONG   options;
MQOD     od1 = {MQOD_DEFAULT};  // Object descriptor for queue 1
MQOD     od2 = {MQOD_DEFAULT};  // Object descriptor for queue 2
MQMD     md  = {MQMD_DEFAULT};  // Message descriptor
MQPMO    pmo = {MQPMO_DEFAULT}; // Put message options
MQGMO    gmo = {MQPMO_DEFAULT}; // Get message options
:
// 1 Connect application to a queue manager.
strcpy  (QMName,"MYQMGR");
MQCONN  (QMName, &HCon, &CompCode, &Reason);

// 2 Open a queue for output
strcpy  (od1.ObjectName,"QUEUE1");
MQOPEN  (HCon,&od1, MQOO_OUTPUT, &Hobj1, &CompCode, &Reason);

// 3 Put a message on the queue
MQPUT   (HCon, Hobj1, &md, &pmo, 100, &buffer, &CompCode, &Reason);

// 4 Close the output queue
MQCLOSE (HCon, &Hobj1, MQCO_NONE, &CompCode, &Reason);

// 5 Open input queue
options = MQOO_INPUT_AS_Q_DEF;
strcpy  (od2.ObjectName, "QUEUE2");
MQOPEN  (HCon, &od2, options, &Hobj2, &CompCode, &Reason);

// 6 Get message
gmo.Options = MQGMO_NO_WAIT;
buflen  = sizeof(buffer - 1);
memcpy (md.MsgId, MQMI_NONE, sizeof(md.MsgId);
memset (md.CorrelId, 0x00, sizeof(MQBYTE24));
MQGET (HCon, Hobj2, &md, &gmo, buflen, buffer, 100, &CompCode, &Reason);

// 7 Close the input queue
options = 0;
MQCLOSE (HCon, &Hobj2,options, &CompCode, &Reason);

// 8 Disconnect from queue manager
MQDISC  (HCon, &CompCode, &Reason);
```

Figure 21. A Code Fragment

6 For the get, the nowait option is used. The MQGET needs the length of the buffer as an input parameter. Since there is no message ID or correlation ID specified, the first message from the queue is read. You may specify a wait interval (in milliseconds) here. You can check the return code to find out if the time has expired and no message arrived.

7 This statement closes the input queue.

8 The application disconnects from the queue manager.