Enterprise Computing
Einführung in das Betriebssystem z/OS

Prof. Dr. Martin Bogdan
Dr. rer. nat. Paul Herrmannn
Prof. Dr.-Ing. Wilhelm G. Spruth

WS 2009/2010

Teil 11

MQSeries

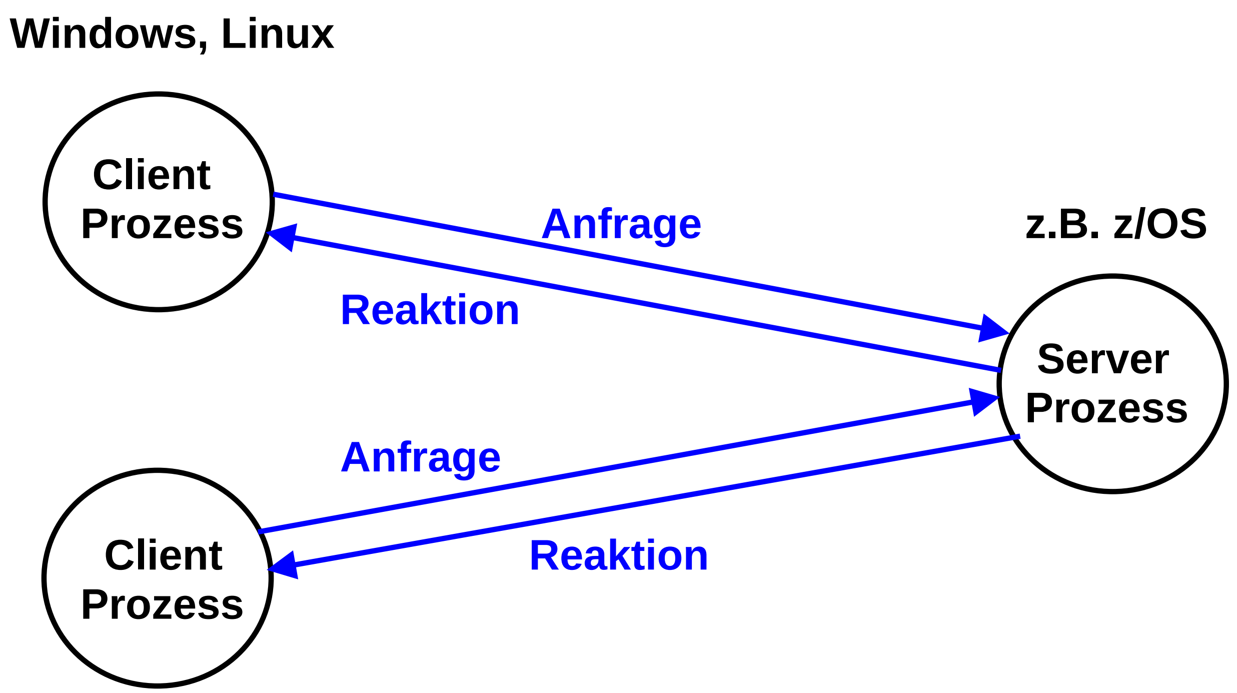
Überblick

Queues and Channels

Trigger

MQ – CICS Bridge

Literature: Dieter Wackerow: MQSeries Primer. www.redbooks.ibm.com/redpapers/pdfs/redp0021.pdf
also available at http://www.informatik.uni-leipzig.de/cs/Literature/index.html

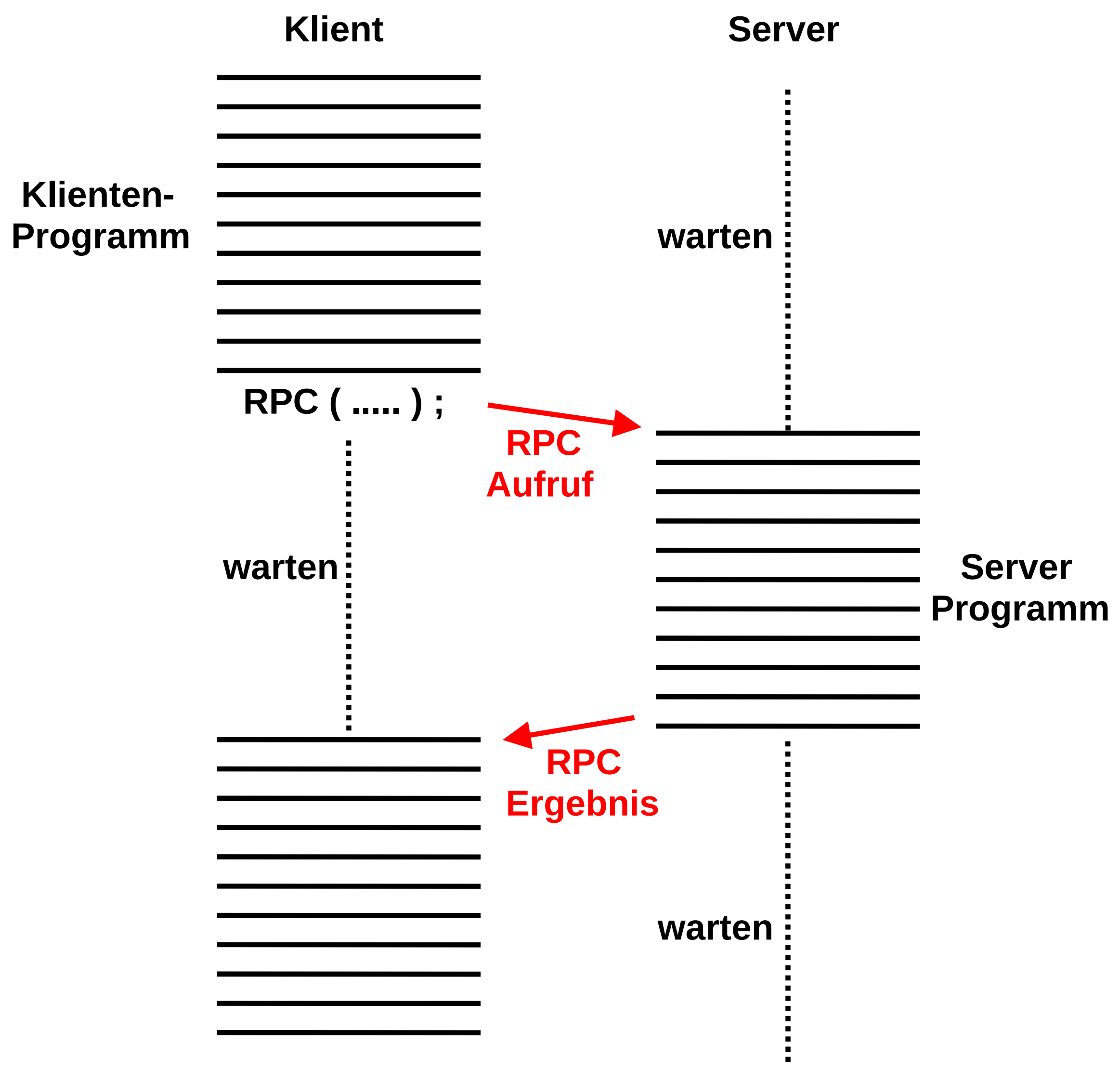


Client/Server-Modell

Prozesse auf einem Klienten-Rechner nehmen die Dienstleistungen eines Servers in Anspruch. Ein Server bietet seine Dienste (Service) einer Menge a priori unbekannter Klienten (Clients) an.

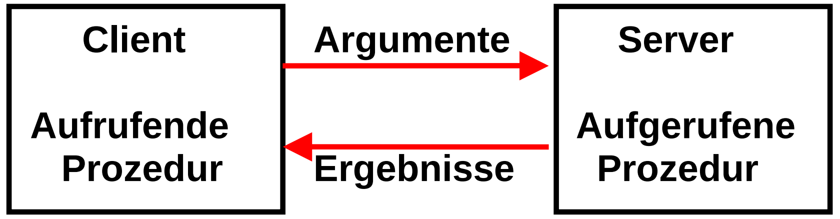
Klient:	**Nutzer eines ServerDienstes**
Server:	**Rechner, der Dienst-Software ausführt**
Dienst (Service):	**Software-Instanz, die auf einem oder mehreren Server Rechnern ausgeführt wird**
Interaktionsform:	**Anfrage / Reaktion (Request/Reply)**

Ein Rechner kann gleichzeitig mehrere Serverdienste (Services) anbieten.



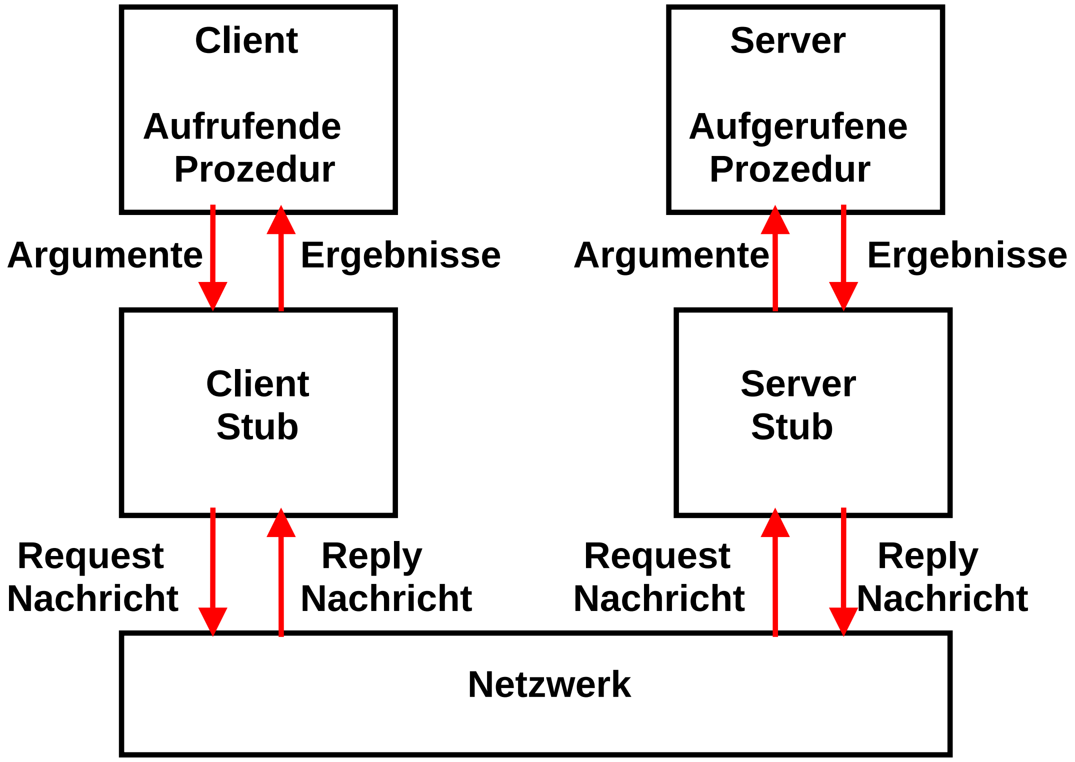
Klient blockiert während der Ausführung des Server Programms

Synchroner RPC



Local Procedure Call

Ein aufrufender Prozess führt eine aufgerufene Prozedur in seinem eigenen Adressraum aus



Remote Procedure Call

Client und Server laufen als zwei separate Prozesse. Sie können (Ausnahmefall), müssen aber nicht, auf dem gleichen Rechner laufen.

Die beiden Prozesse kommunizieren über Stubs. Stubs sind Routinen, welche lokale Prozedur Aufrufe auf Netzwerk RPC Funktionsaufrufe abbilden.

Beispiele für Remote Procedure Calls

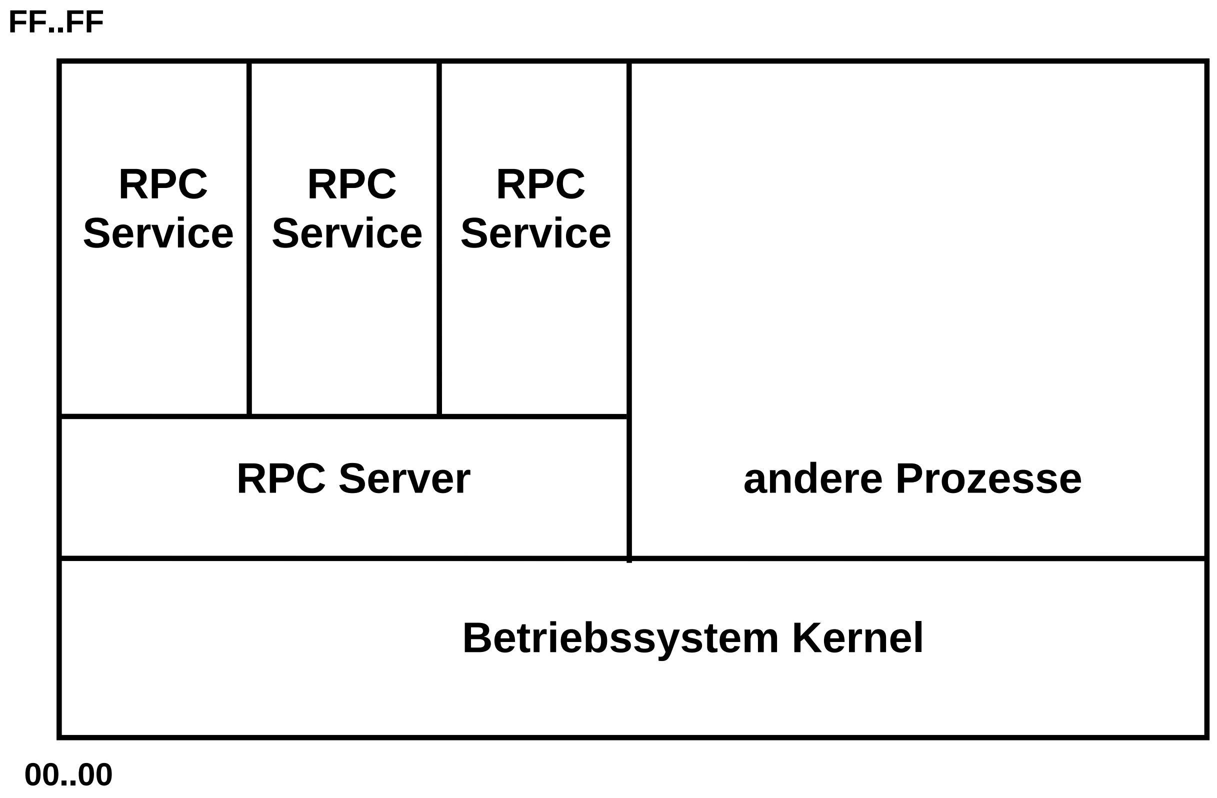
Klassische RPCs

- **Sun RPC**
- **DCE RPC**

Moderne RPCs

- **Corba**
- **RMI**

- **Web Services RPC (SOAP RPC)**



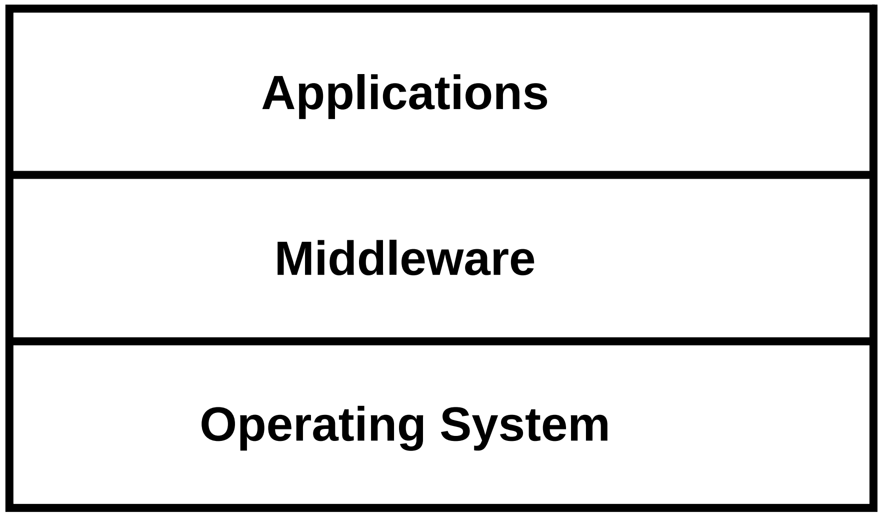
Die RPC Services können entweder in getrennten virtuellen Adressenräumen laufen, oder alternativ als Threads innerhalb eines virtuellen Adressenraums implementiert werden.

Ein RPC Service wird auch als *Implementation* bezeichnet.

Häufig hunderte unterschiedlicher RPC Services auf dem gleichen Rechner.

Der RPC Server stellt Verwaltungsdienste für seine RPC Services zur Verfügung, z.B. das *Binding*.

Auf einem Rechner können mehrere RPC Server laufen, die z.B. unterschiedliche Arten von RPC Services gruppieren.



Middleware

Software, which is located between applications and the operating system, is called middleware.

Some middleware examples are transaktion processing monitors like CICS, IMS/DC, Tuxedo or the SAP/R3 transaktion processing monitor.

Other examples are RPC, DCE, CORBA, RMI as well as Message oriented Middleware(MOM).

Message Based Queuing (MBQ)
Message oriented Middleware (MOM)

MBQ and MOM (both are synonyms) are methods for program to program communication. Programs are enabled to send and receive information without any direct connection between them. Programs communicate by depositing messages in message-queues, and by retrieving messages from message-queues.

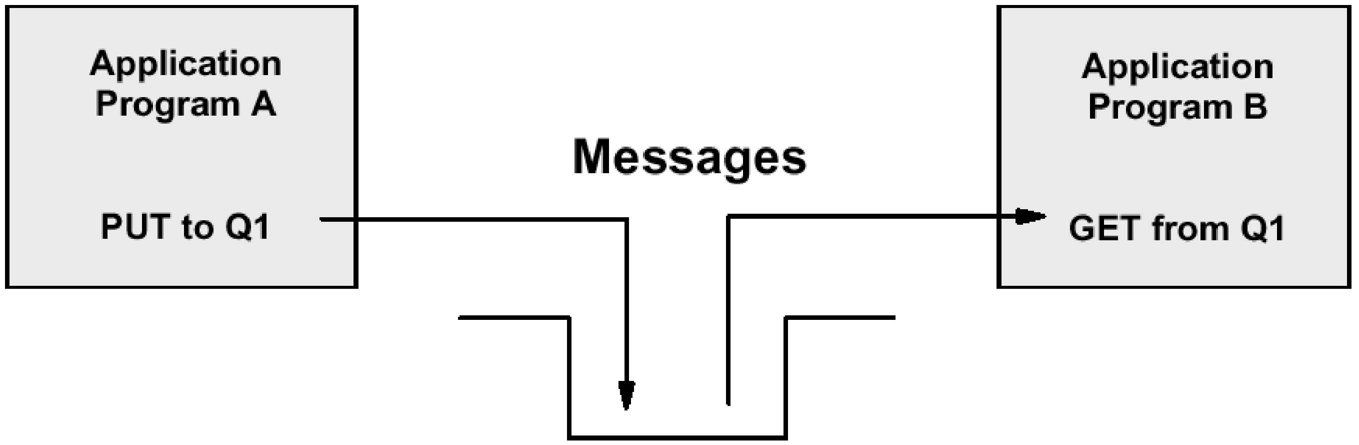
Leading MOM manufacturers are:

• **BEA**	**MessageQ**
• **IBM**	**MQSeries, now renamed WebSphere MQ**
• **Microsoft**	**MS Message Queue Server (MSMQ)**
• **Oracle**	**Advanced Queuing (AQ),**

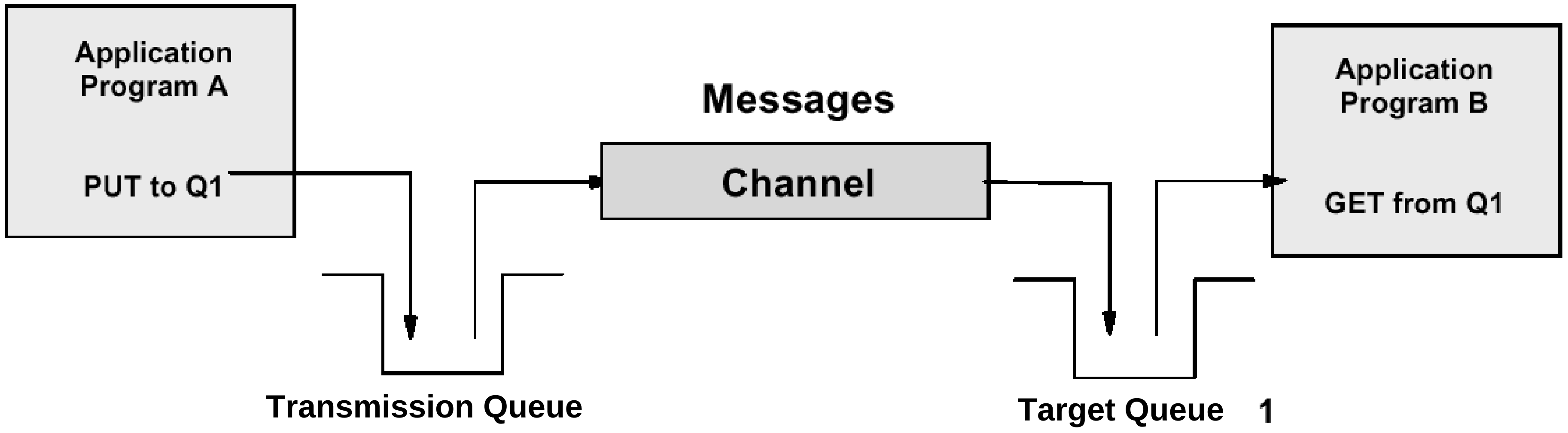
Message Based Queuing is attractive for integration of heterogenious hardware, software and application environment.

SMTP (internet mail) and X.500 electronic mail are primitive message oriented applications.

IBM MQSeries, now renamed WebSphereMQ, is the undisputed market leader. The product is available for nearly all operating system platforms, e.g.: AIX, DG/UX, HP-UX, Windows, OS/390, z/OS, OS/400, Psion Pervasive SOD, Sequent, Sinix, Solaris, Tandem, TPF, TrueUnix, VMS/VAX.

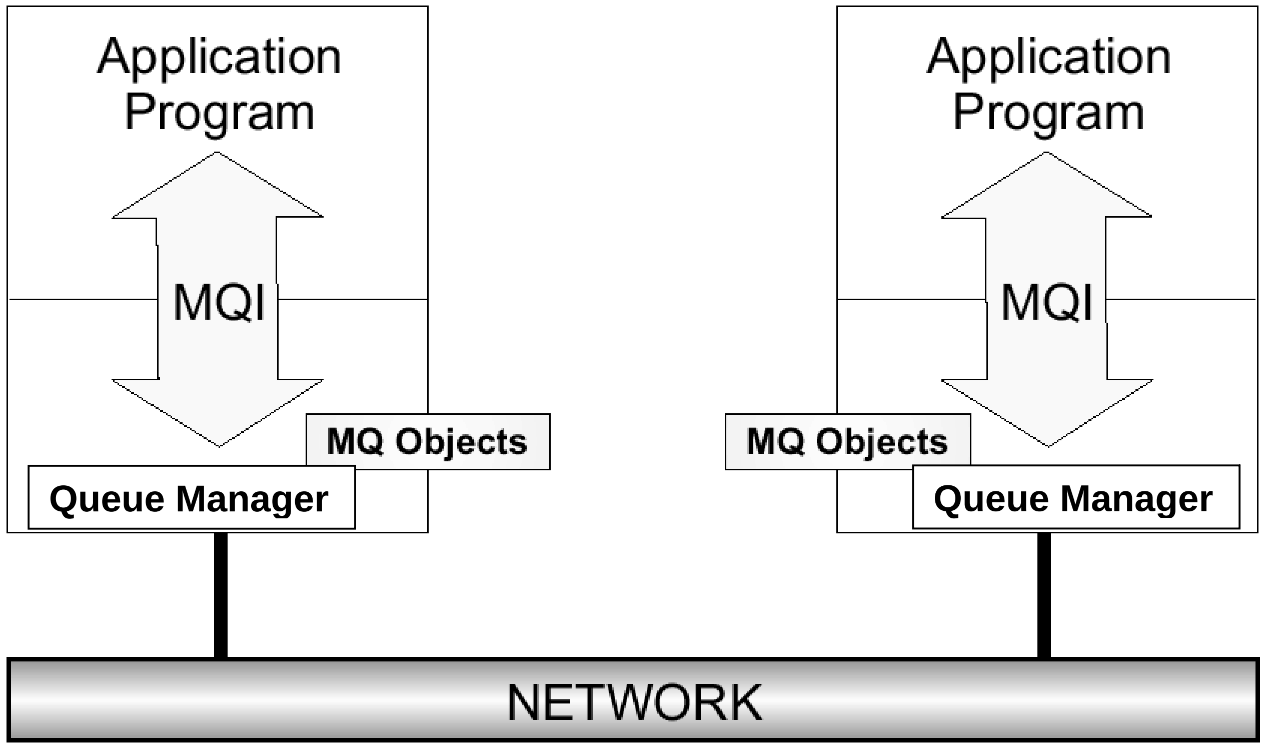


Program-to-Program Communication - One System



Program-to-Program Communication - Two Systems

A program may send messages to another program that runs in the same machine as the queue manager, 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.



MQSeries 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.

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.

The figure above shows the main parts of an MQSeries application at run time. Programs use MQSeries API calls, 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.

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.

MQI - MQSeries API

MQCONN stellt eine Verbindung mit einem Queue-Manager mit Hilfe von Standard-Links her.

MQGET

MQPUT

MQCLOSE

MQPUT1 öffnet eine Queue, legt eine Message darauf ab und schließt die Queue wieder. Dieser API-Call stellt eine Kombination von MQOPEN, MQPUT und MQCLOSE dar.

MQDISC schließt die Übergabe einer Arbeitseinheit ein. Das Beenden eines Programms ohne Unterbrechung der Verbindung zum Queue-Manager verursacht ein "Rollback" (MQBACK).

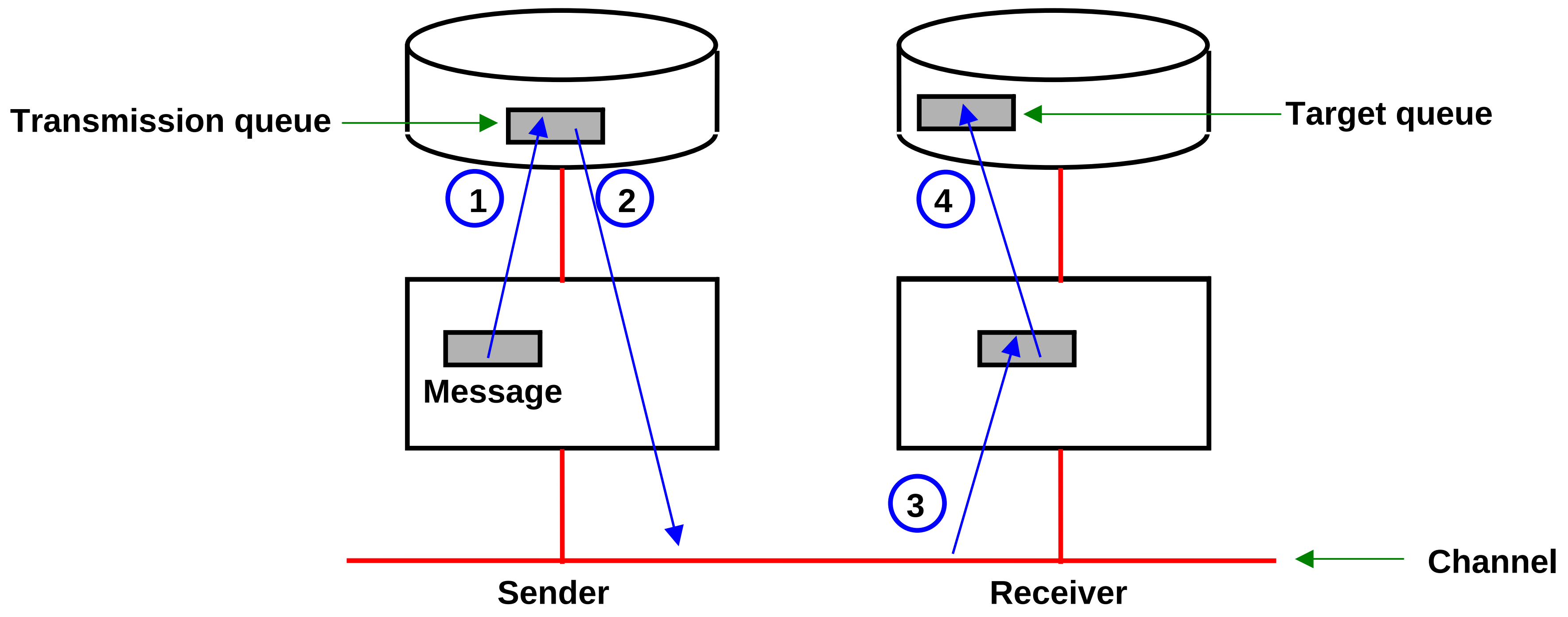
MQBEGIN startet eine Arbeitseinheit, die durch den Queue-Manager koordiniert wird und externe XA-kompatible Ressource-Manager enthalten kann.

MQINQ fordert Informationen über den Queue-Manager oder über eines seiner Objekte an, wie z.B. die Anzahl der Nachrichten in einer Queue.

MQSET verändert einige Attribute eines Objekts.

MQCMIT gibt an, dass ein Syncpoint erreicht worden ist.

MQBACK teilt dem Queue-Manager alle zurück gekommenen PUT´s- und GET´s-Nachrichten seit dem letzten Syncpoint mit.

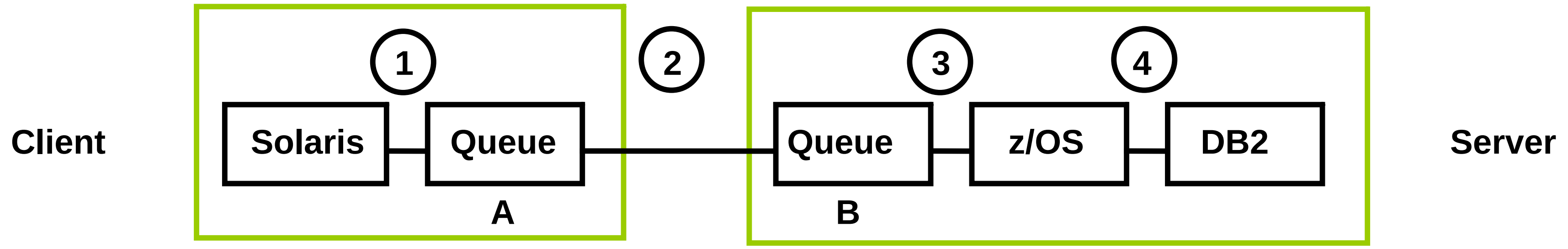


Persistent and Non-Persistent Messages

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. Non-persistent messages cannot be recovered after a system restart.

Persistent messages implement communication with ACID Properties. The message is stored persistently by the sender before being sent. The receiver stores the message persistently, before any action is takten on it, and confirms receipt to the sender. Persistent messages are transmitted with ACID properties.

Persistent Messaging Operation



Steps 1 - 3 : If successfuk, send confirmation

Step 4: If z/OS to DB2 unsuccessful, roll back to Queue B, then recover

A message queue is used to store messages sent by programs. They are defined as objects belonging to the queue manager.

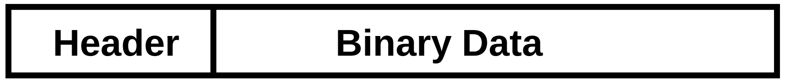
When an application puts a message on a queue, the queue manager ensures that the message is:

- **Stored safely**
- **Is recoverable**
- **Is delivered once, and once only, to the receiving application**

This is true even if a message has to be delivered to a queue owned by another remote queue manager This property is known as the assured delivery property of WebSphere MQ.

About Messages

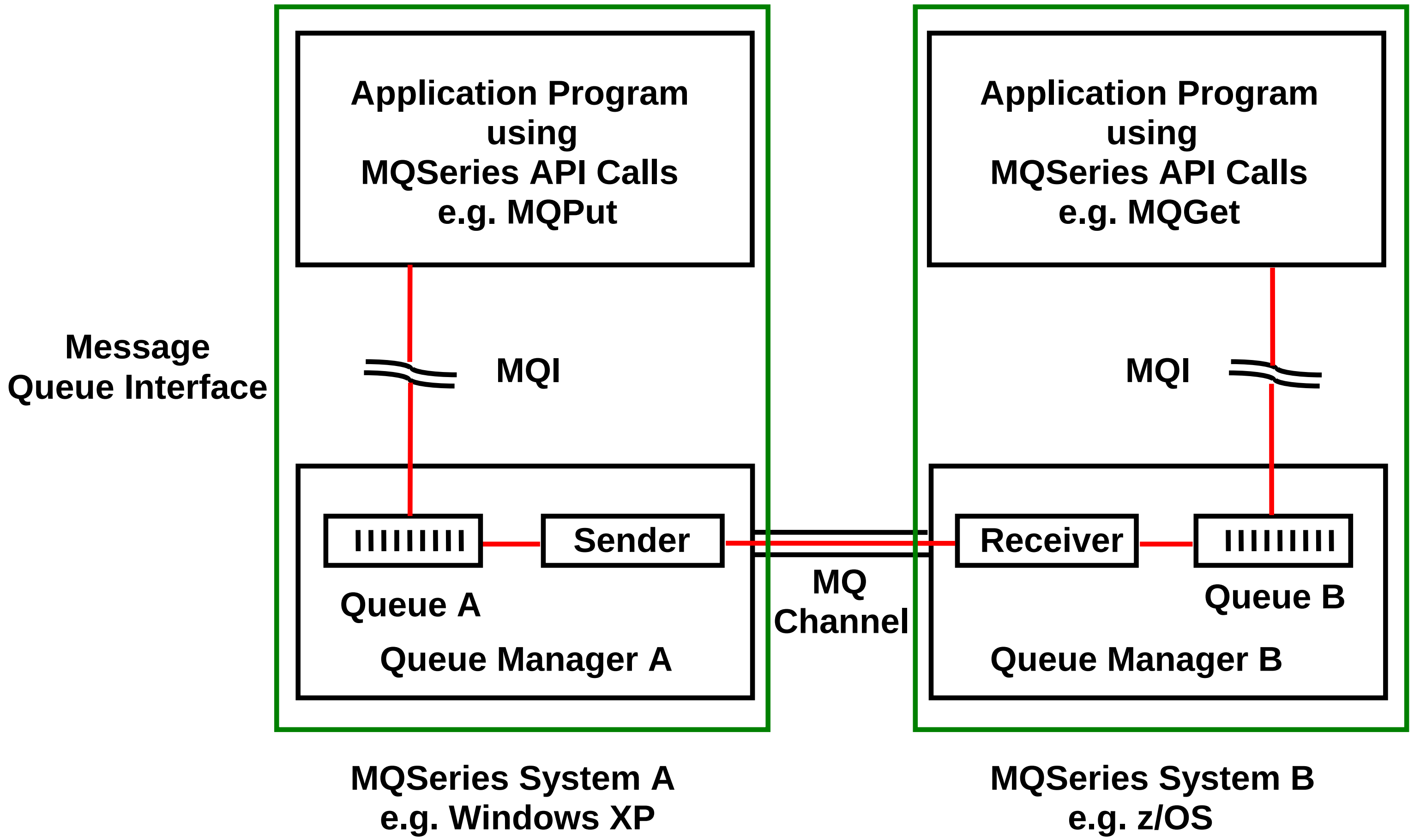
A message is a string of bytes that is meaningful to the applications that use it. Messages are used to transfer information from one application program to another (or to different parts of the same application). The applications can be running on the same platform, or on different platforms.



WebSphere MQ messages consist 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.


Application Program using MQSeries API Calls e.g. MQPut
Application Program using MQSeries API Calls e.g. MQGet
Message Queue Interface
MQI
MQI
IIIIIIIII
Sender
Receiver
IIIIIIIII
Queue A
Queue B
MQ Channel
Queue Manager A
Queue Manager B
MQSeries System A e.g. Windows XP
MQSeries System B e.g. z/OS

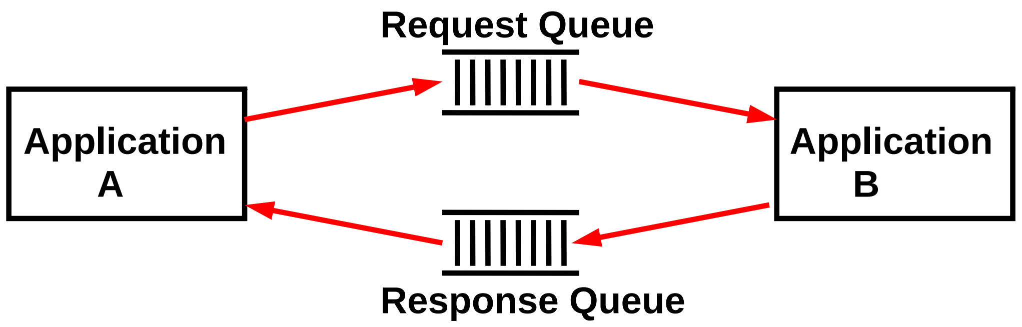
MQSeries Operation

The application program puts a message into queue A (transmission queue), using the MQPut call , which is part of the Message Queue Interface (MQI). Queue A is owned by queue manager A, who uses its sender to transmit the message over the MQ channel to the receiver of queue manager B. Queue manger B owns queue B (target queue, also called request queue or destination queue), where it stores the message. The receiving application program can now retrieve the message from queue B at ist leisure.

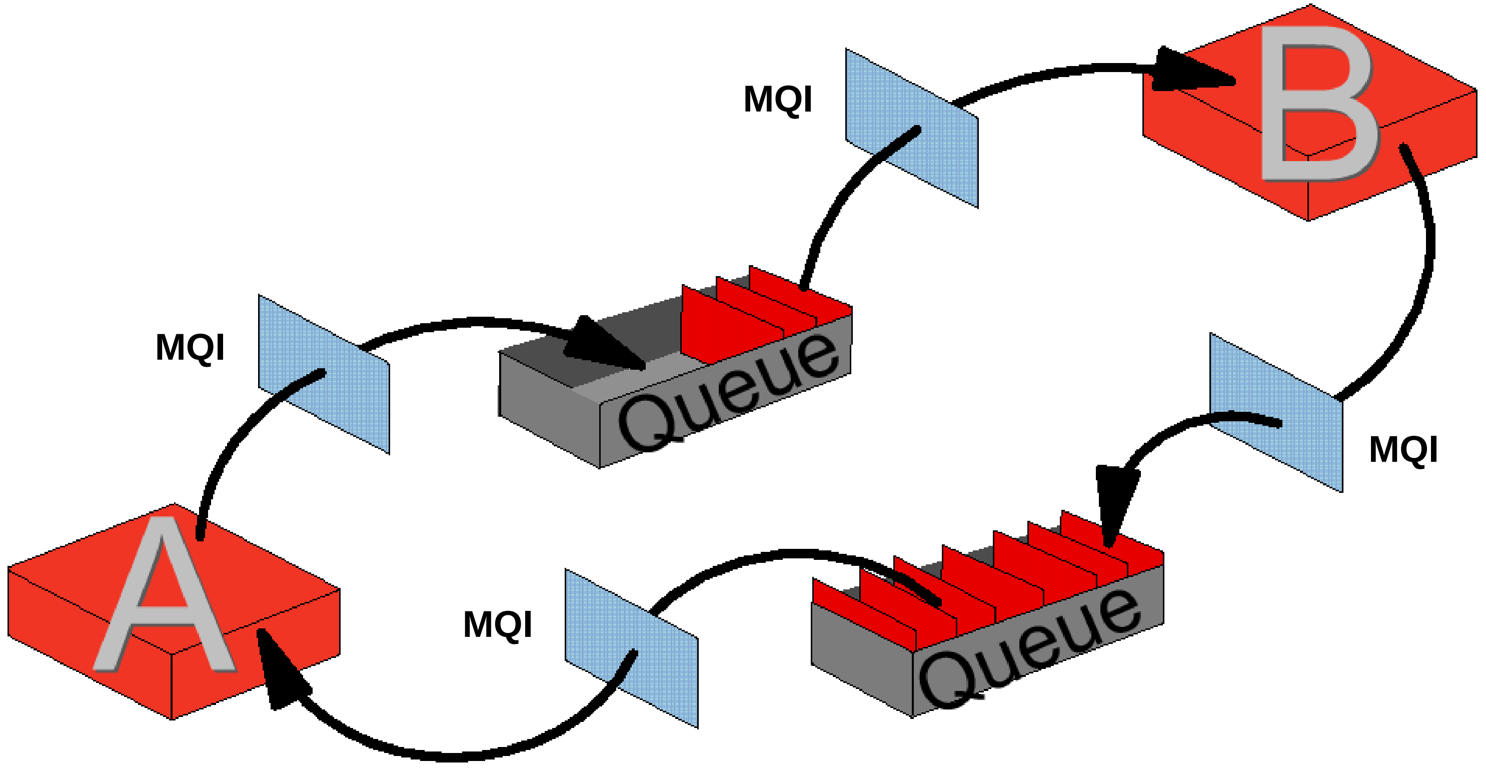
MQSeries pseudosynchronous operation

MQSeries operates asynchronously and unidirectional. Bidirectional operation requires 2 duplicate queues.

Although MQSeries uses asynchroner operation, a Request/reply Modus can simulate synchronous operation, and is frequently used that way.



For this, the message header containst a „Reply to ....“ indicator.



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.

The figure above 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.

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

Some Attributes of the Message Descriptor

- **The version of the message descriptor depends on the MQSeries version and platform you use. For the functions used with z/OS,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 (or higher). 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 will discuss *persistent* and *non-persiste*nt messages further down. 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 dat*e. 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 que*ue 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.**

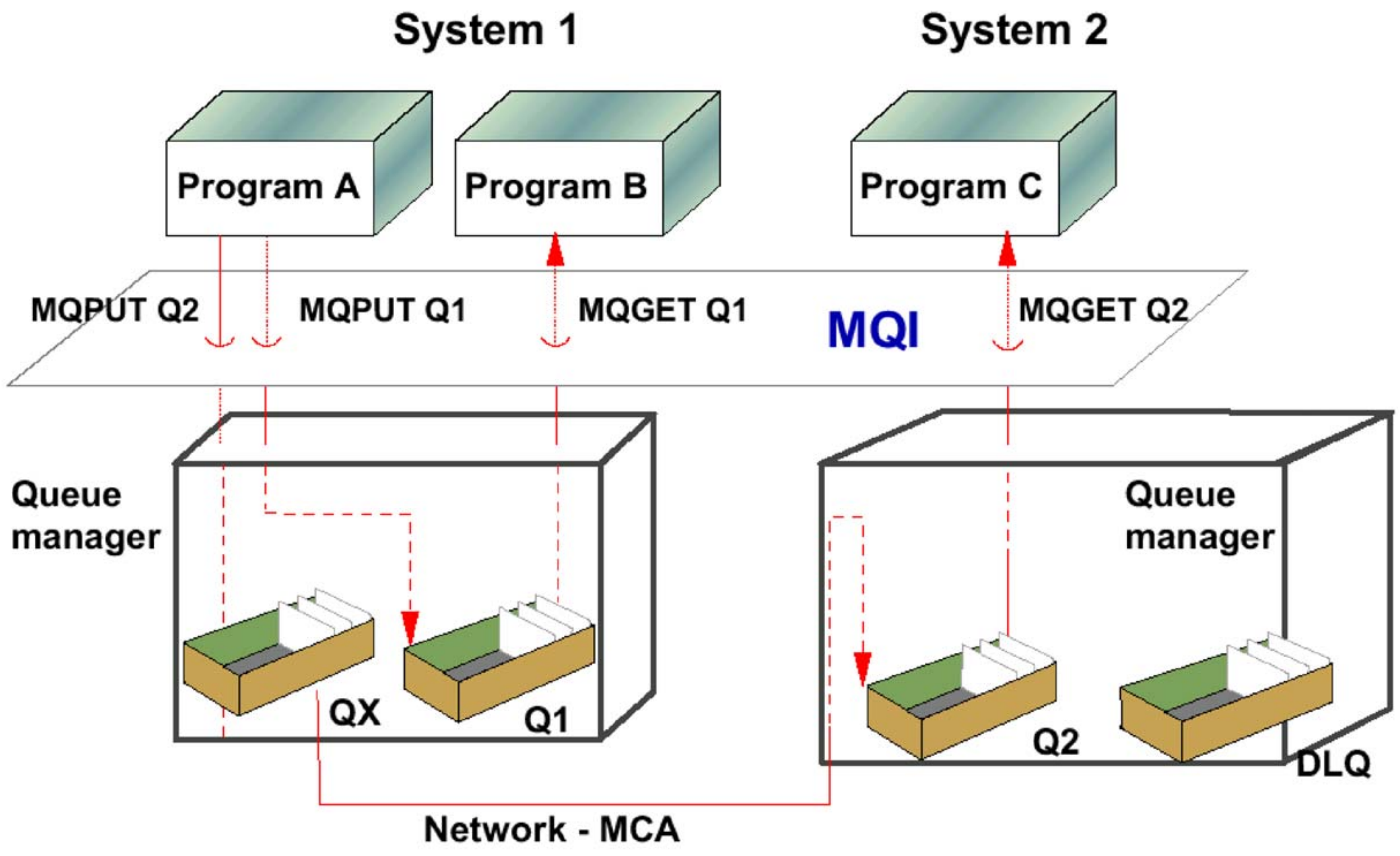
- **Each time a message is backed out, the *backout counter* is increased. An application can check this counter and act on it, for example, send the message to a different queue where the reason for the backout is analyzed by an administrator.**

- **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.**

Message Types

MQSeries knows four types of messages that can be spezified in the message descriptor:

- **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.**



The MQSeriers Queue Manager on a system can serve multiple applications with multiple queues simultaneously.

Queue Manager

The heart of MQSeries is the message queue manager (MQM), which is MQSeries' run-time program.

Before you can let your application programs use WebSphere MQ on your z/OS system, you must install the WebSphere MQ for z/OS product and start a queue manager. The queue manager owns and manages the set of resources that are used by WebSphere MQ. These resources include:

- **Page sets that hold the WebSphere MQ object definitions and message data**
- **Logs that are used to recover messages and objects in the event of queue manager failure**
- **Processor storage**
- **Connections through which different application environments (CICS ® , IMS ™ , and Batch) can access the WebSphere MQ API**
- **The WebSphere MQ channel initiator, which allows communication between WebSphere MQ on your z/OS system and other systems**

The queue manager has a name, and applications can connect to a rempte queue manager using its name.

The job of the queue manager is to manage queues and messages for applications. It provides an API, 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 below.

The queue manager can retain messages for future processing in the event of application or system outages. Messages are retained in a queue until a successful completion response is received through the MQI.

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 functions of a queue manager can be summarized as follows:

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

Note: The Networking Blueprint for the MQI identifies three communication styles (two in addition to MQI):

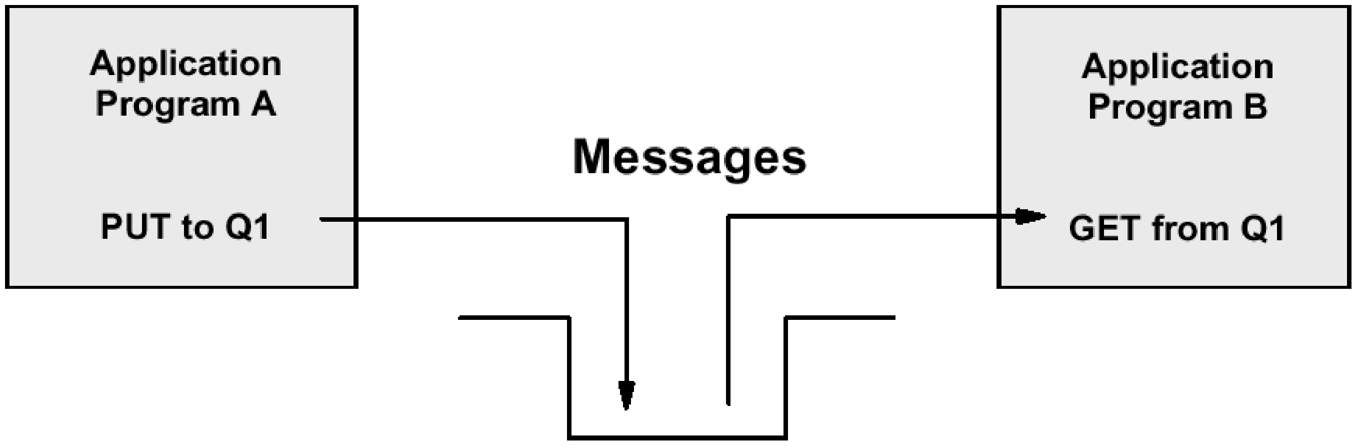
1. **Common Programming Interface - Communications (CPI-C, an RPC like method, originally developed for SNA)**
2. **Remote Procedure Call (RPC)**
3. **Message Queue Interface (MQI)**

- **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. Distribution lists are not supported in WebSphere MQ for z/OS.**
- **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 provides graphical user interfaces; other platforms use the command line interface or panels.**

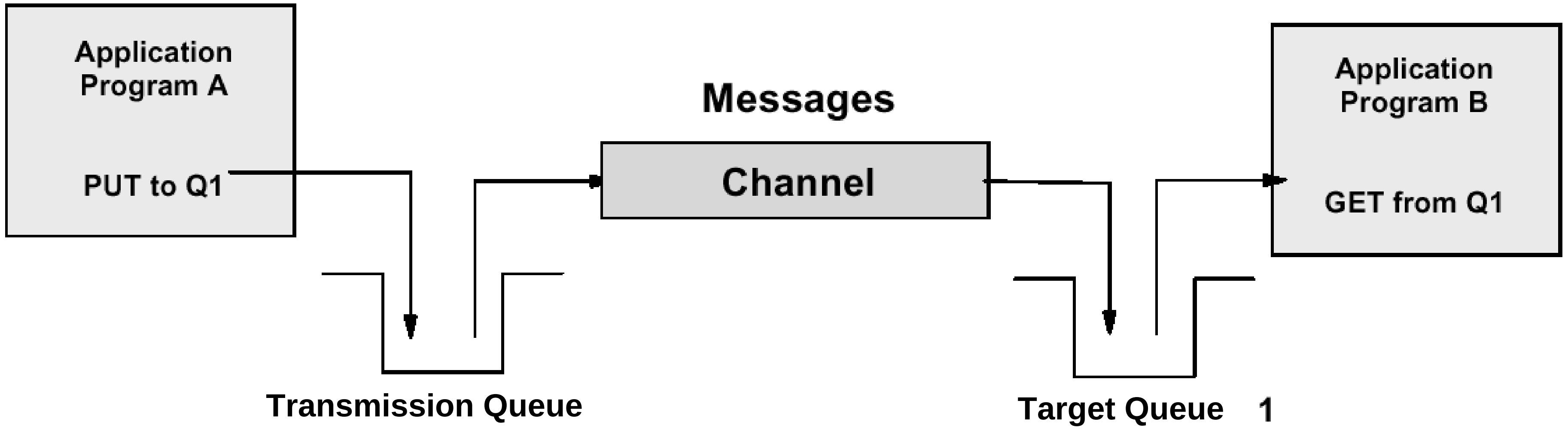
MQSeries clients do not have a queue manager in their machines. Client machines connect to a queue manager in a server. The queue manager manages the queues for all clients attached to it.

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.

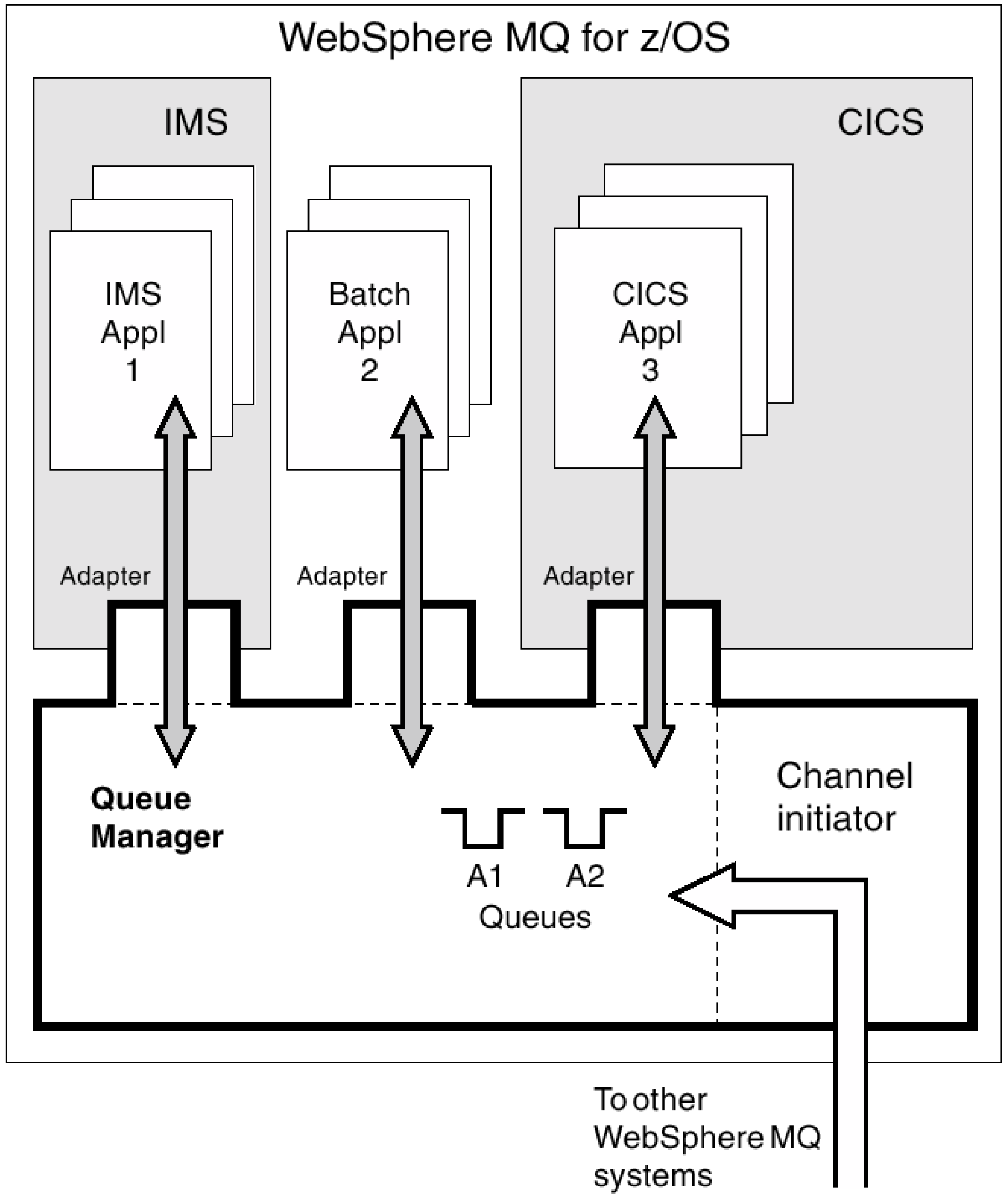


Program-to-Program Communication - One System



Program-to-Program Communication - Two Systems

A program may send messages to another program that runs in the same machine as the queue manager, 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.



Overview of WebSphere MQ on a single z/OS system

On z/OS, WebSphere MQ runs as a z/OS subsystem that is started at IPL time. Within the subsystem, the queue manager is started by executing a JCL procedure that specifies the z/OS data sets that contain information about the logs, and that hold object definitions and message data (the page sets). The subsystem and the queue manager have the same name, of up to four characters. All queue managers in your network must have unique names, even if they are on different systems, sysplexes, or platforms.

The figure on the left is an example of program-to-program Communication on a single system.

Überblick

Queues and Channels

Trigger

MQ – CICS Bridge

WebSphere MQ Objects

WebSphere MQ uses the term “object”.

Many of the tasks you carry out when using WebSphere MQ involve manipulating WebSphere MQ objects. In WebSphere MQ for z/OS the object types are queues, processes, authentication information objects, namelists, storage classes, Coupling Facility structures, channels, clusters, system objects, and queue manager security objects. The queue manager itself is an object, too.

Channel Objects are an example. You may look at channel objects as definitions. The attributes of a channel object define how communication is performed. For example, these attributes can specify whether Secure Sockets Layer (SSL) authentication is required when establishing a channel

Usually, an administrator creates one or more queue managers and their objects. The manipulation or administration of objects includes the following:

- **Starting and stopping queue managers.**
- **Creating objects, particularly queues, for applications.**
- **Working with channels to create communication paths to queue managers on other (remote) systems.**

The properties of an object are defined by its attributes. Some you can specify, others you can only view.

The objects are common across different MQSeries platforms. There are other objects that apply to z/OS systems only, such as the buffer pool, PSID (table space page set identifier), and storage class.

WebSphere MQ Objects

In WebSphere MQ for z/OS the object types are

- **queues,**
- **processes,**
- **authentication information objects,**
- **namelists,**
- **storage classes,**
- **Coupling Facility structures,**
- **channels,**
- **clusters,**
- **system objects, and**
- **queue manager security objects.**

The queue manager itself is an object, too.

What is a Queue (Message Queue)

A *queue (message queue)* is a data structure used to store messages until they are retrieved by an application.

Queues are managed by a *queue manage*r. The queue manager is responsible for maintaining the queues it owns, storing all the messages it receives onto the appropriate queues, and retrieving the messages in response to application requests.

The messages might be put on the queues by application programs, or by a queue manager as part of its operation.

There are local queues that are owned by the local queue manager, and remote queues that belong to a different queue manager.

Local queues are real queues; the queue manager can sore a message in them. Non-local queues are queue objects with attributes, which provide definitions which can be used by a local queue manager.

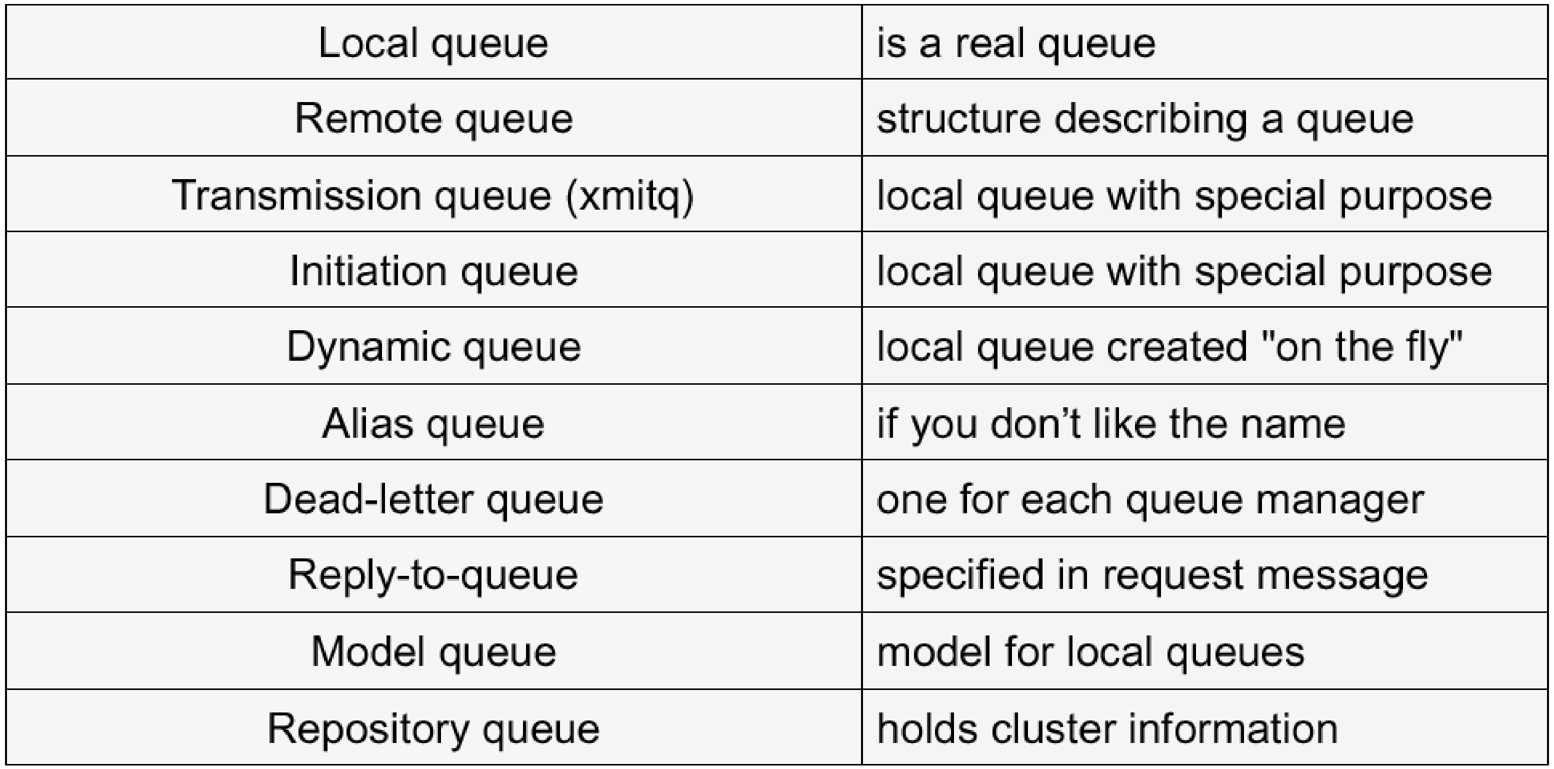
Queues

The term queue is used frequently in WebSphere MQ terminology. However, in different contexts, the term has different meanings:

- Queue objects are definitions, and are used to define a queue and ist properties. Queue objects can represent either an actual queue that can contain messages, or a method for finding the destination of another queue within the WebSphere MQ infrastructure.
- Queues that can contain messages are called local queues. Local queue objects are the only type of queue object that represent a real queue that holds messages.Transmission queues, which are the intermediate queues between queue managers, are special cases of local 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.
- A cluster instance of a queue, known to the local queue manager, is generally referred to as *cluster queue*. A cluster queue is not an actual queue object on the local queue manager. It is a local representation to the queue manager of an instance of a queue object that exists somewhere within a queue manager cluster.

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), or on a remote server. Listed below are different queue types and their purposes.



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

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, application program A and application 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. Local queues are also called real queues, because the queue manager can store a message in them. Non-local queues are queue objects, whose attributes can be used by the queue manager.

Cluster Queue

A cluster queue is a local queue that is known throughout a cluster of queue managers. 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 is an object, but not a real queue. A remote queue definition is used by a local queue manager when he wants to send a message to a remote 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 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.

What is a channel?

All network communication in WebSphere MQ is performed across a channel.

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.

Messages are transported from a transmission queue over a *channel to a target* queue*.* A channel is a logical communication link. The conversational style of program-to-program communication requires the existence of a communications connection between each pair of communicating applications. Channels shield applications from the underlying communications protocols.

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.

Types of Channels

In MQSeries, when talking about network communication links, there are two different kinds of channels:

Message channel

A message channel connects two queue managers through message channel agents (MCAs). A message channel is unidirectional, comprised of two message channel agents (a sender and a receiver) and a communication protocol. An MCA transfers messages from a transmission queue to a communication link, and from a communication link to a target queue. For bidirectional communication, it is necessary to define a *pair* of channels, consisting of a sender and a receiver.

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.

MQI channel

A Message Queue Interface (MQI channel) connects a WebSphere MQ client (also called a slim client) to a queue manager. WebSphere MQ clients do not have a queue manager of their own.

An MQI channel is bidirectional.

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 WebSphere MQ client
- Fat client

The difference between a slim client and a fat client is the way messages are sent. The queues reside either in the end user’s workstation or on a server. A slim client requires a WebSphere MQ Server that provides queue manager services. A WebSphere MQ Server usually attaches many slim clients.

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

When a slim WebSphere MQ 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.

However, the client in the tutorial mentioned above (and the associated figure) is a fat 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.

The following summarizes the three workstation types:

MQSeries Client

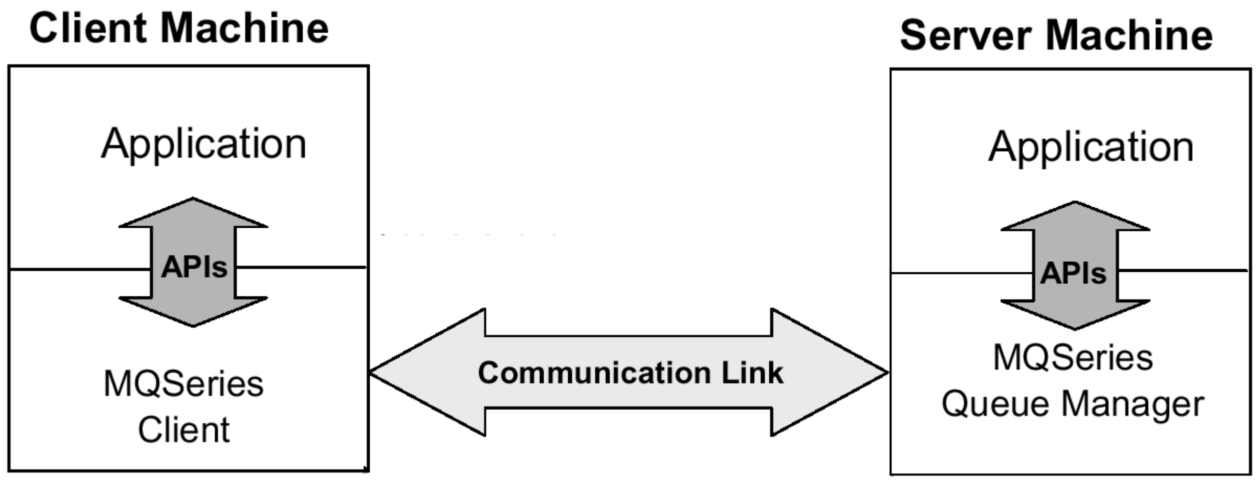
A client workstation does not have a queue manager of its own. It shares a queue manager in a server with other clients. All MQSeries objects, such as queues, are in the server. Since the connection between client and server is synchronous, the application cannot work when the communication is broken.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

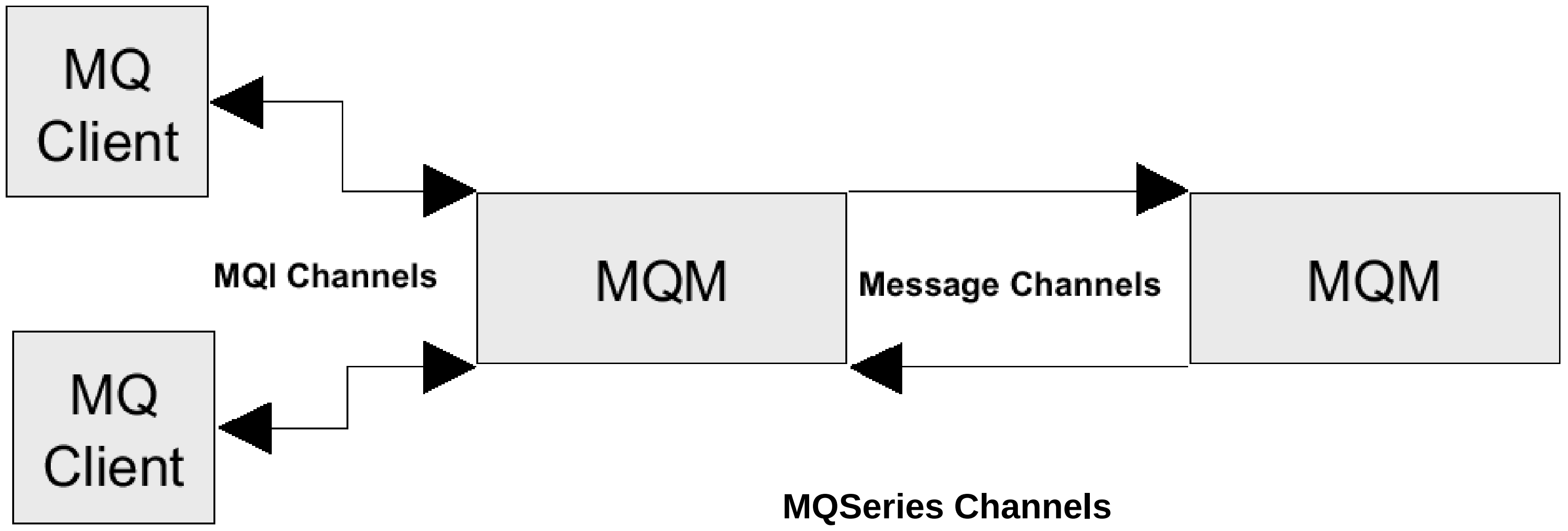
MQSeries for Windows was designed for use by a single user. It has its own "small footprint" queue manager with its own objects. However, it is not an intermediate node between other nodes. It is called a leaf node. You could also refer to it as a "fat" client. This product is able to queue outbound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



MQI Channel

Clients and servers are connected with MQI channels. An MQI channel consists of a sender/receiver channel-pair, called Client Connection channel and Server Connection channel.

The application on the client machine uses the MQI API to access the MQSeries Client, but the client machine has no queue manager.



MQSeries Channels

The figure above shows 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.

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.

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.

Überblick

Queues and Channels

Trigger

MQ – CICS Bridge

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.pmc.express.doc/tasks/tjfp0031_.html

Queue name resolution

When an application sends a message, the destination can be a queue on the same queue manager to which the application is connected. More frequently, it will be a destination on another queue manager within the WebSphere MQ infrastructure.

There may be a point-to-point connection between the sending and the receiving queue manager. Alternatively, there can be multiple intermediate queue managers between the queue manager to which an application is connected and the final destination.

The queue manager to which the application is connected must have knowledge of how to route messages to that remote queue manager. It must have a local queue designated to temporarily store messages travelling to that destination, called a transmission queue.

Each queue manager on route makes an independent decision regarding the next destination for a message on route to its final destination. This is based on the queue manager's own knowledge of the infrastructure. This knowledge is contained in the WebSphere MQ objects defined on that queue manager and can be supplemented with knowledge received from other queue managers in a queue manager cluster.

The process of resolving destination information, as provided by an application, into the next destination for messages on route to their final destination is called *queue name resolutio*n. Queue name resolution occurs within a queue manager each time a message is received from an application or another queue manager. Every time an MQOPEN call is performed to open a queue before putting messages on it, queue name resolution is performed by the queue manager. This takes the information provided in a parameter (the MQOD structure) on the MQOPEN call, the *object nam*e, and *object queue manager nam*e, and attempts to resolve this information into a valid destination queue name and queue manager name for the message.

After queue name resolution has been performed, an application can put messages to the resolved destination using MQPUT calls. This results in one of the following actions being performed by the queue manager:

- If queue name resolution has identified that the destination queue is local to the queue manager, the message are put directly into that queue.

- If queue name resolution has identified that the destination is on another queue manager, known to the current queue manager, the message is put on a *transmission queue* to be sent to that queue manager. This step requires that information is added to the message so that queue name resolution can be performed again when the message reaches the remote queue manager. This information is called a *transmission queue heade*r.

Transmission queues

The USAGE attribute of a local queue object can be used to designate a local queue as a transmission queue. The transmission of messages from a transmission queue to a remote queue manager is performed by a message channel.

Defining a transmission queue provides a queue manager with knowledge of how to route messages to a single destination queue manager. Any messages sent with an *object queue manager name* the same as the name of the transmission queue are placed on that transmission queue. For this reason, the name of the transmission queue and the name of the remote queue manager should generally match.

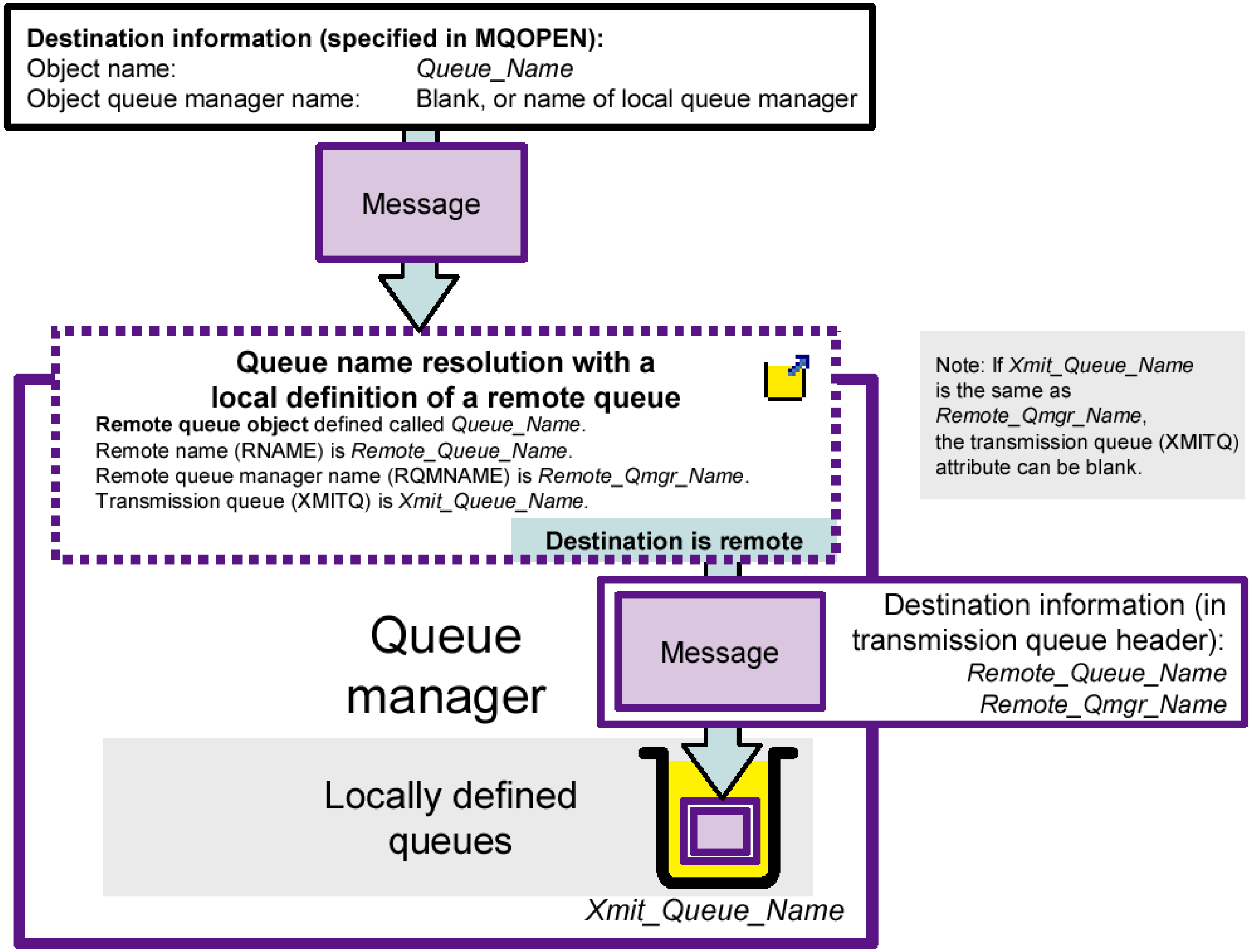
Remote queue objects

Remote queue objects are used to define routes to other queue managers within the WebSphere MQ message queuing infrastructure. This involves mapping queue manager names to transmission queues, and mapping queue names to different queue names on remote queue managers.

A remote queue object can be used as a local definition of a remote queue. This enables an application to open a remote queue object to put messages without specifying an explicit remote queue manager name in the same way as though it was a local queue. It can be used to place a queue on a remote queue manager without making any changes to an application that was originally written to access a local queue. To create a local definition of a remote queue, create a remote queue object with attributes as follows:

- **The name of the remote queue object: The name of the queue to which applications connected to the queue manager put messages.**
- **Remote name (): The name of the queue on the destination queue manager. It can be the same as the name of the remote queue object. Queue name resolution is performed on the remote machine using this queue name.**
- **Remote queue manager name (): The name of the destination queue manager. This is not necessarily the next queue manager on route, if the message has to transvers multiple queue manages to reach its destination. Queue name resolution is performed on the remote machine using this queue manager name.**
- **Transmission queue (): This attribute specifies the name of the transmission queue that takes the message to the next queue manager on route to the destination. If the remote queue manager name is the same as the transmission queue name, this attribute can be left blank.**

The figure below shows an application sending a message through a local definition of a remote queue. The *object name* specified by the application is the name of the remote queue object. Queue name resolution places this message on the transmission queue specified in the remote queue object definition. A transmission queue header is added to the message, which contains the *remote name* and *remote queue manager name* attributes from the remote queue object definition. When the message reaches its destination, the information in the transmission queue header is used to perform queue name resolution on that remote queue manager.



Destination information (specified in MQOPEN):
Object name: *Queue_Name*
Object queue manager name: Blank, or name of local queue manager
Message
Queue name resolution with a local definition of a remote queue
Remote queue object defined called *Queue_Name*.
Remote name (RNAME) is *Remote_Queue_Name*.
Remote queue manager name (RQMNAME) is *Remote_Qmgr_Name*.
Transmission queue (XMITQ) is *Xmit_Queue_Name*.
Destination is remote
Note: If *Xmit_Queue_Name* is the same as *Remote_Qmgr_Name*, the transmission queue (XMITQ) attribute can be blank.
Queue manager
Message
Destination information (in transmission queue header):
Remote_Queue_Name
Remote_Qmgr_Name
Locally defined queues
Xmit_Queue_Name

Distributed message channels

A distributed message channel is a channel that takes messages from a particular transmission queue on one queue manager and delivers those messages to a queue (or queues) on a specific remote queue manager.

A distributed message channel must be manually configured for all transmission queues defined on a queue manager. A queue manager places a message on a transmission queue as the result of queue name resolution. During this process, it adds enough information to that message in a transmission queue header for queue name resolution to occur when that message reaches the destination queue manager.

How does message transmission occur ?

Each distributed message channel consists of two MCAs, thus two channel objects, one on each queue manager. Each MCA assumes one of the following roles based on the type of message channel object defined on the queue manager:

- The sending MCA opens a particular transmission queue, specified in the attributes of the channel object, for exclusive input. This means that no two channels can be configured to flow messages from the same transmission queue. It gets messages from that transmission queue and sends them to the partner MCA.

- The receiving MCA receives messages from the sending MCA. For each message, it removes the transmission queue header from the message and reads its contents. It opens the queue specified in the transmission queue header for that message and then puts the message to the queue. These open and put actions are performed using standard MQI calls. This means that queue name resolution happens on the queue manager in the same way as though an application had connected to that queue manager and put the message using the details in the transmission queue header. If a valid destination cannot be determined for the message during the queue name resolution on that queue manager, the message cannot be delivered. We discuss this in 7.4.11, “Message delivery failures” on page 175.

The transmission queue header is generated during queue name resolution on the queue manager where the sending MCA is running and used to put the message to a queue on the queue manager where the receiving MCA is running. The header contains the following information:

- **Remote queue name:**

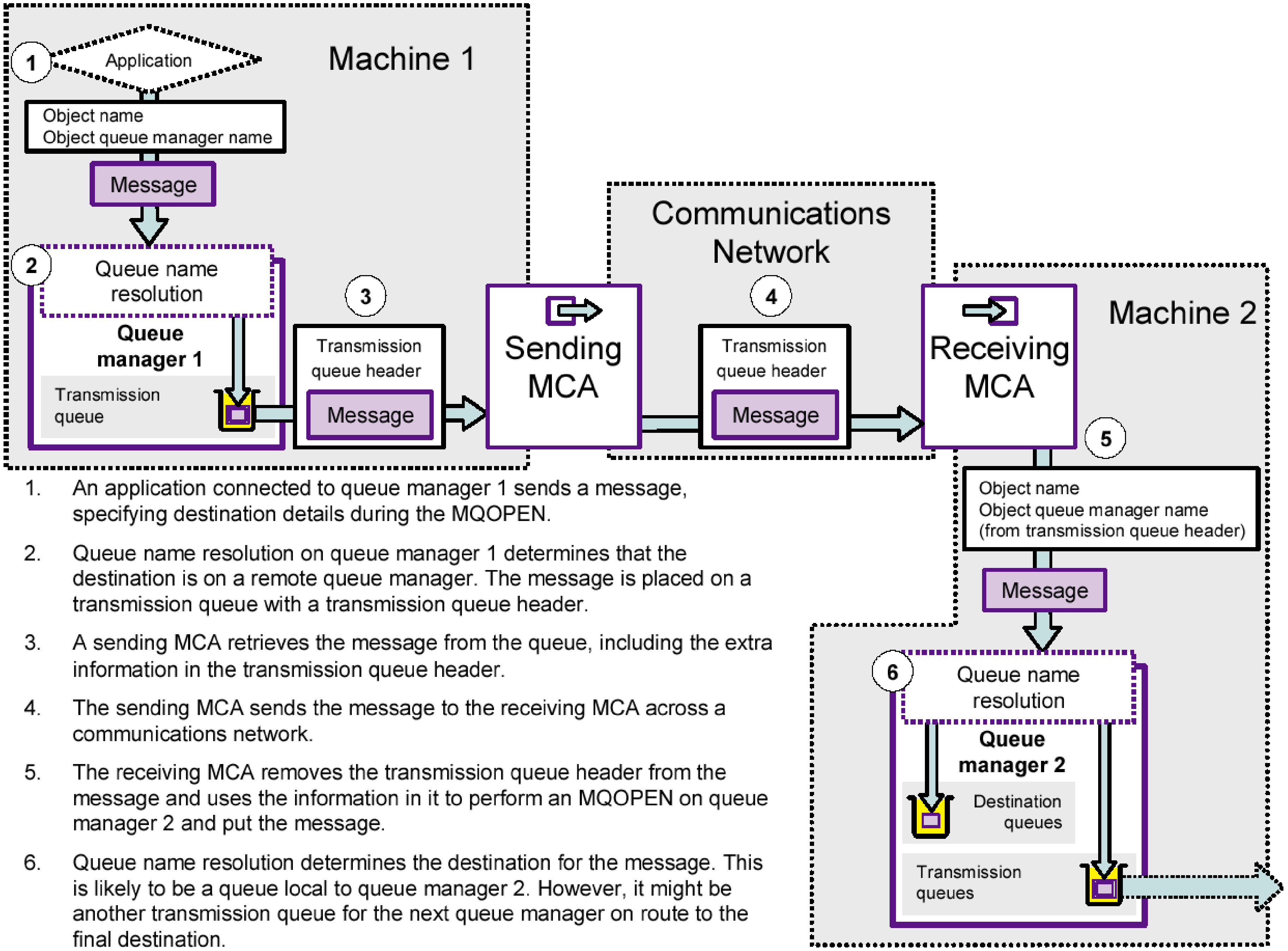
  The name of the destination queue for the message, which was resolved by the queue manager during queue name resolution. When the MCA at the remote queue manager attempts to put the message to a queue on the remote queue manager, it specifies this queue name in MQOD when opening the queue.

- **Remote queue manager name:**

  The name of the destination queue manager that hosts the destination queue. This is also resolved during queue name resolution and might not match the name of the queue manager to which the message is delivered, for example, if the next queue manager to receive the message is not the final destination for the message.

- **The original message descriptor of the message:**

  In order to add the transmission queue header to a message, the message becomes altered. The message queue header is added to the beginning of the message body, and the message descriptor is altered to describe the transmission queue header, rather than the data contained in the message. However, when the message arrives on the destination queue, the message must reflect the original message correctly, including the message descriptor. The original message descriptor of the message is stored in the transmission queue header and used when the MCA puts the message to the remote queue manager with the transmission queue header removed.



1. An application connected to queue manager 1 sends a message, specifying destination details during the MQOPEN.
2. Queue name resolution on queue manager 1 determines that the destination is on a remote queue manager. The message is placed on a transmission queue with a transmission queue header.
3. A sending MCA retrieves the message from the queue, including the extra information in the transmission queue header.
4. The sending MCA sends the message to the receiving MCA across a communications network.
5. The receiving MCA removes the transmission queue header from the message and uses the information in it to perform an MQOPEN on queue manager 2 and put the message.
6. Queue name resolution determines the destination for the message. This is likely to be a queue local to queue manager 2. However, it might be another transmission queue for the next queue manager on route to the final destination.

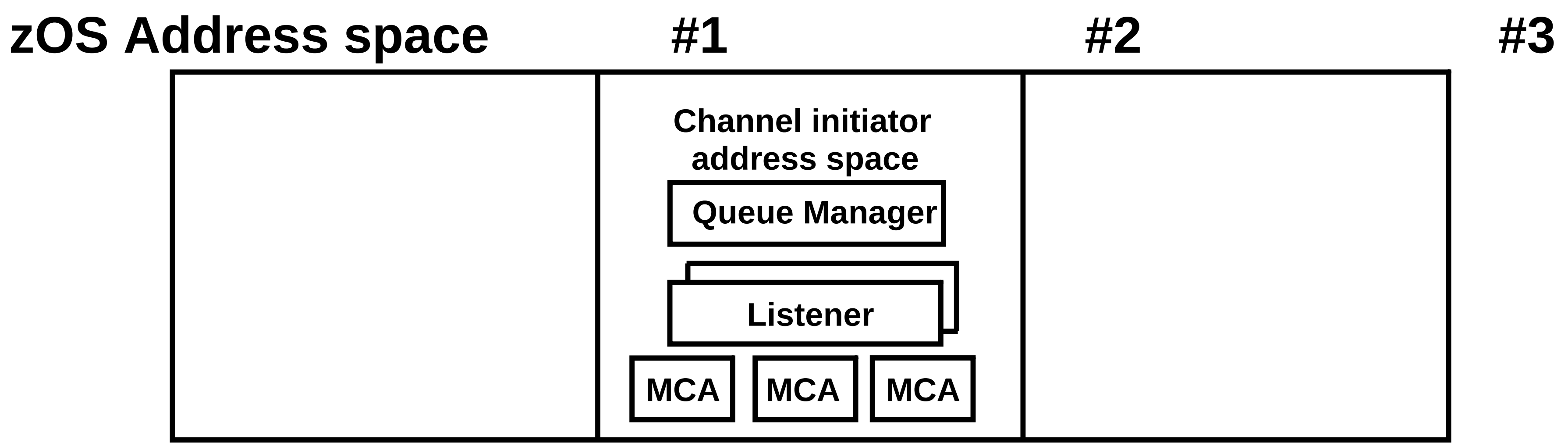
Channel Initiator

The channel initiator provides and manages resources that enable WebSphere MQ distributed queuing. WebSphere MQ uses Message Channel Agents (MCAs) to send messages from one queue manager to another.

To send messages from queue manager A to queue manager B, a sending MCA on queue manager A must set up a communications link to queue manager B. A receiving MCA must be started on queue manager B to receive messages from the communications link. This one-way path consisting of the sending MCA, the communications link, and the receiving MCA is known as a channel. The sending MCA takes messages from a transmission queue and sends them down a channel to the receiving MCA. The receiving MCA receives the messages and puts them on to the destination queue (target queue).

In WebSphere MQ for z/OS, the sending and receiving MCAs all run inside the channel initiator (the channel initiator is also known as the mover). The channel initiator runs as a z/OS address space under the control of the queue manager. There can be only a single channel initiator connected to a queue manager and it is run inside the same z/OS address space as the queue manager. It hosts all message channel agents (MCAs) for a queue manager, There can be thousands of MCA processes running inside the channel initiator concurrently.

You can use the channel initiator to start channels.



Channel Initiator

The channel initiator provides and manages resources that enable WebSphere MQ distributed queuing. WebSphere MQ uses Message Channel Agents (MCAs) to send messages from one queue manager to another.

To send messages from queue manager A to queue manager B, a sending MCA on queue manager A must set up a communications link to queue manager B. A receiving MCA must be started on queue manager B to receive messages from the communications link. This one-way path consisting of the sending MCA, the communications link, and the receiving MCA is known as a channel. The sending MCA takes messages from a transmission queue and sends them down a channel to the receiving MCA. The receiving MCA receives the messages and puts them on to the destination queue (target queue).

In WebSphere MQ for z/OS, the sending and receiving MCAs all run inside the channel initiator (the channel initiator is also known as the mover). The channel initiator runs as a z/OS address space under the control of the queue manager. There can be only a single channel initiator connected to a queue manager and it is run inside the same z/OS address space as the queue manager. It hosts all message channel agents (MCAs) for a queue manager, There can be thousands of MCA processes running inside the channel initiator concurrently.

You can use the channel initiator to start channels.

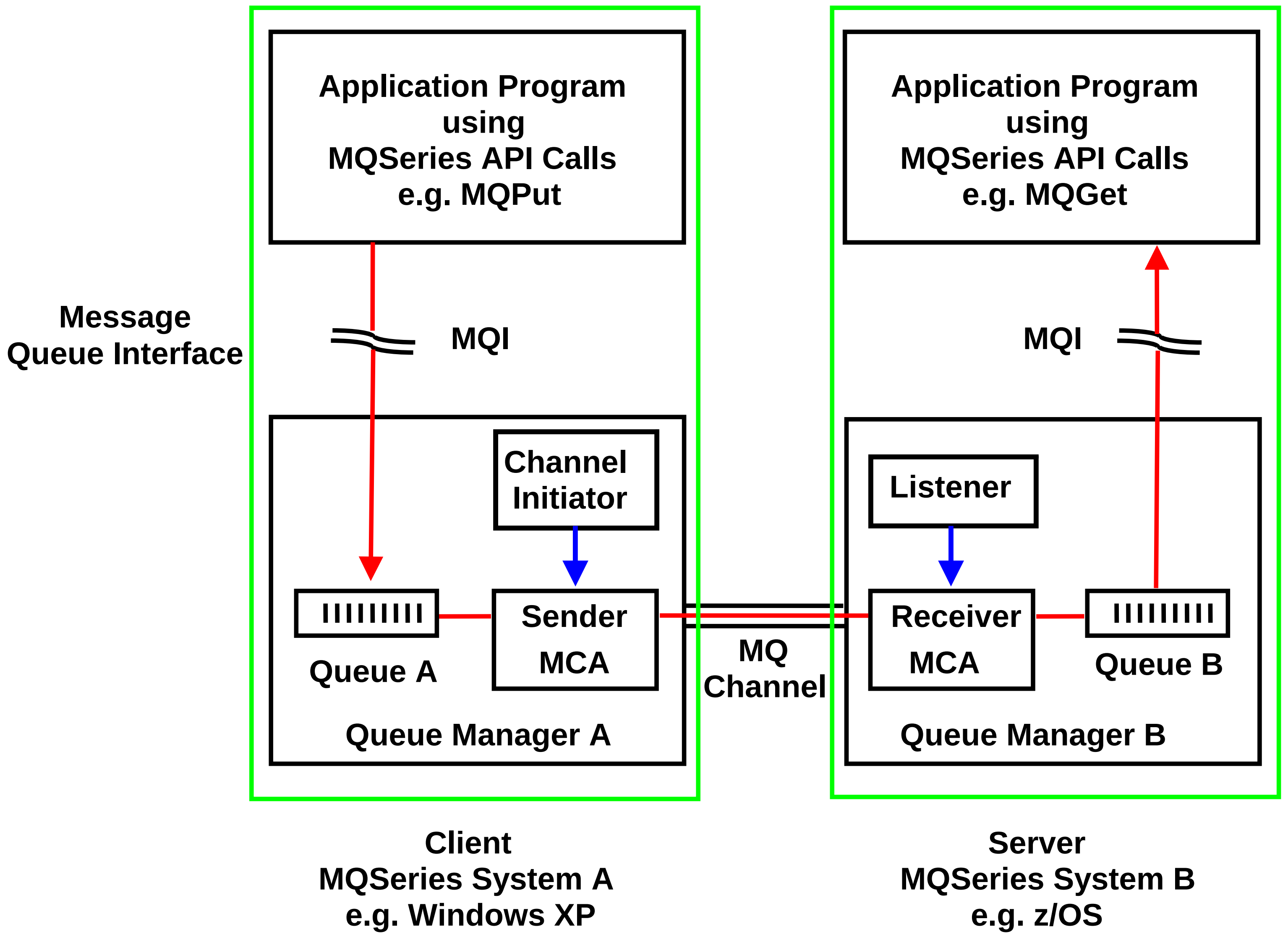
Listener

The listener process listens for connections to arrive on a port and then creates a message channel agent (MCA) to process that connection, whether it is a distributed message channel, cluster message channel, or client connection.

With WebSphere MQ for z/OS, listening on TCP/IP is performed within the WebSphere MQ for z/OS channel initiator.

Multiple TCP/IP listeners can be started within the channel initiator. Each listener listens on a particular TCP/IP port. A listener is started using the START LISTENER command against the subsystem of the queue manager.

WebSphere MQ for z/OS also supports LU 6.2 listeners.



MQSeries Operation

Queue A is called the transmission queue, and queue B is called the target queue or request queue.

Triggering

Messages arriving on queues represent events within the system. In most cases, processing of that message is required.

WebSphere MQ provides *triggering* to allow this processing to automatically be *initiated* by a queue manager. This processing can represent starting a channel to move messages from a transmission queue to a remote queue manager, or starting an application instance to perform an action on a message or a batch of messages.

Depending on the use of a queue, a message arriving on that queue might, or might not, represent an event for which processing must be performed. The queue can be configured to generate *trigger events* that match the usage of that queue.

Generation of trigger events

A queue can be configured in the following ways to generate trigger events:

- Every: A trigger message is generated for every message that arrives on the queue.
- First: After a queue becomes empty, the first message that arrives on that queue generates a trigger event. A trigger event is usually generated if any application has the queue open for input to retrieve messages. However, if a message arrives after a specified trigger interval, and no application has the queue open for input, another trigger event occurs.
- Trigger interval, which is a queue manager-wide attribute on the queue manager object, to the required number of milliseconds
- Depth: A message arriving on a queue, which brings the total number of messages on a queue above a certain threshold, generates a trigger event. A trigger event is not generated if any application has the queue open for input to retrieve messages.

Initiation queues and trigger messages

When a trigger event occurs, a message is placed on an *initiation queu*e. This message is called a *trigger messag*e. The name of this queue onto which the trigger message is place is specified in the initiation queue attribute of a local queue object.

Any local queue can be designated as an initiation queue. No special configuration needs to be performed on a local queue in order for it to be designated as an initiation queue, but it should not be configured as a transmission queue.

Each trigger message contains information about the action that needs to be performed in response to the trigger event. The message contains the following information:

- **A queue name: This is the name of the queue from which the event was generated.**
- **Details of an application to execute: Details of how to execute an application are platform specific. WebSphere MQ provides the PROCESS object type. This has attributes that provide enough information to execute a particular application in the operating system. If the PROCESS attribute of the queue that generated the event contains the name of a valid PROCESS object, the trigger message contains all attributes from that object.**
- **Trigger data: This is custom data from the trigger data () attribute of the queue that generated the event.**

Trigger monitor

An application that waits for messages to arrive on a local queue designated as an initiation queue is referred to as a trigger monitor.

The queue manager defines certain conditions as constituting *trigger events*. If triggering is enabled for a queue and a trigger event occurs, the queue manager sends a *trigger message* to a queue called an *initiation queue*. The presence of the trigger message on the initiation queue indicates that a trigger event has occurred.

The program which processes the initiation queue is called a *trigger-monitor application*, and its function is to read the trigger message and take appropriate action, based on the information contained in the trigger message.

Normally this action would be to start some other application, to process the queue which caused the trigger message to be generated. From the point of view of the queue manager, there is nothing special about the trigger-monitor application--it is simply another application that reads messages from a queue (the initiation queue).

If triggering is enabled for a queue, you have the option to create a *process-definition object* associated with it. This object contains information about the application that processes the message which caused the trigger event. If the process definition object is created, the queue manager extracts this information and places it in the trigger message, for use by the trigger-monitor application.

If you want a trigger to start a channel, you do not need to define a process definition object.

Application queue
An application queue is a local queue, which, when it has triggering set on and when the conditions are met, requires that trigger messages are written.

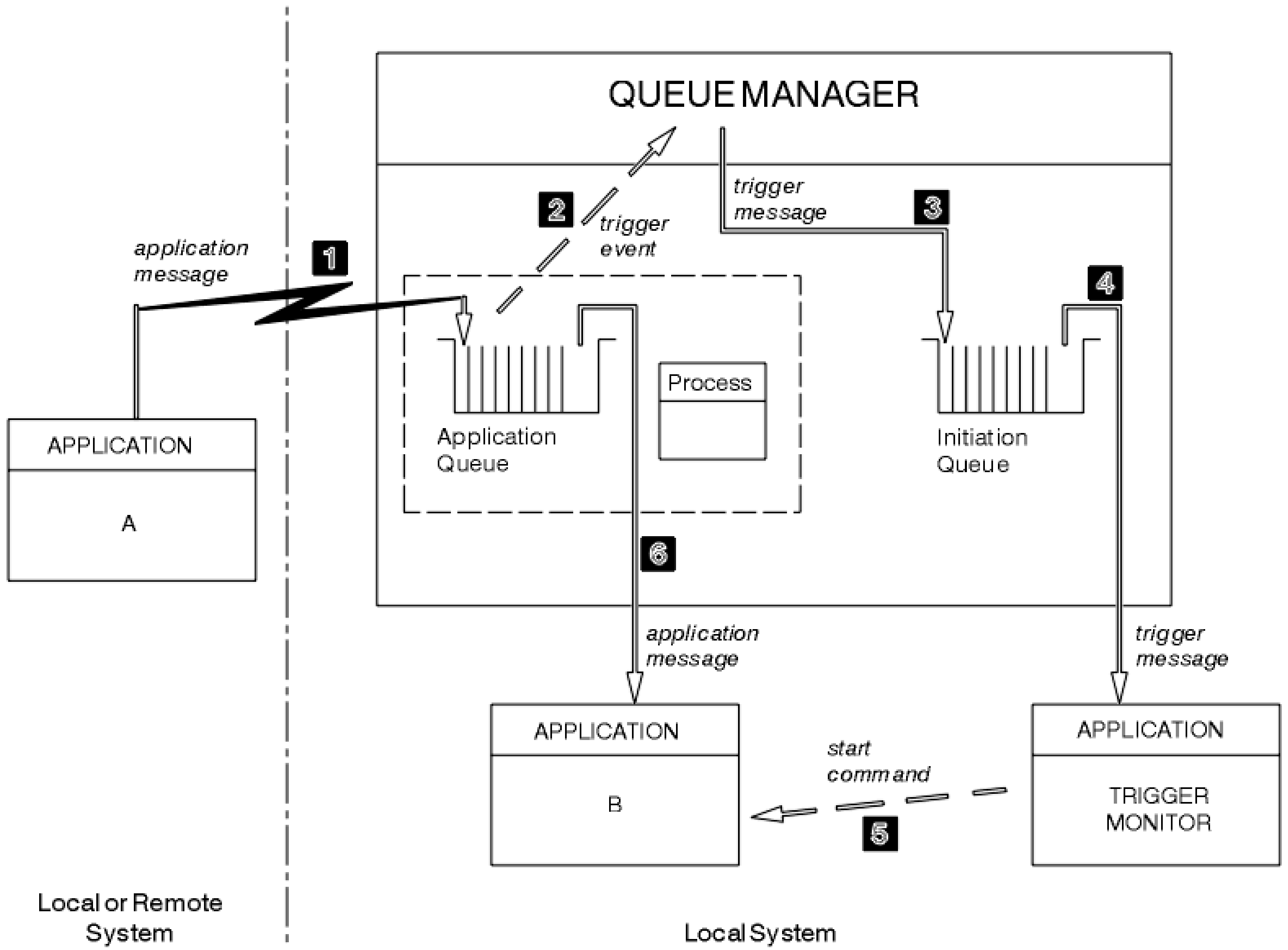
Trigger event
A trigger event is an event that causes a trigger message to be generated by the queue manager. This is usually a message arriving on an application queue.

Trigger message
The queue manager creates a trigger message when it recognizes a trigger event. It copies into the trigger message information about the application to be started. This information comes from the application queue and the process definition object associated with the application queue.

Initiation queue
An initiation queue is a local queue on which the queue manager puts trigger messages. A queue manager can own more than one initiation queue, and each one is associated with one or more application queues. A shared queue, a local queue accessible by queue managers in a queue-sharing group, can be an initiation queue on WebSphere MQ for z/OS.

Trigger monitor
A trigger monitor is a continuously-running program that serves one or more initiation queues. When a trigger message arrives on an initiation queue, the trigger monitor automatically retrieves the message. The trigger monitor uses the information in the trigger message. It issues a command to start the application that is to retrieve the messages arriving on the application queue, passing it information contained in the trigger message header, which includes the name of the application queue.

On all platforms, a special trigger monitor known as the channel initiator is responsible for starting channels. On z/OS, the channel initiator is usually started manually.



http://middleware.its.state.nc.us/middleware/Documentation/en_US/htm/csqzal06/csqzal0637.htm

The sequence of events is:

1. Application A, which can be either local or remote to the queue manager, puts a message on the application queue. Note that no application has this queue open for input. However, this fact is relevant only to trigger type FIRST and DEPTH.

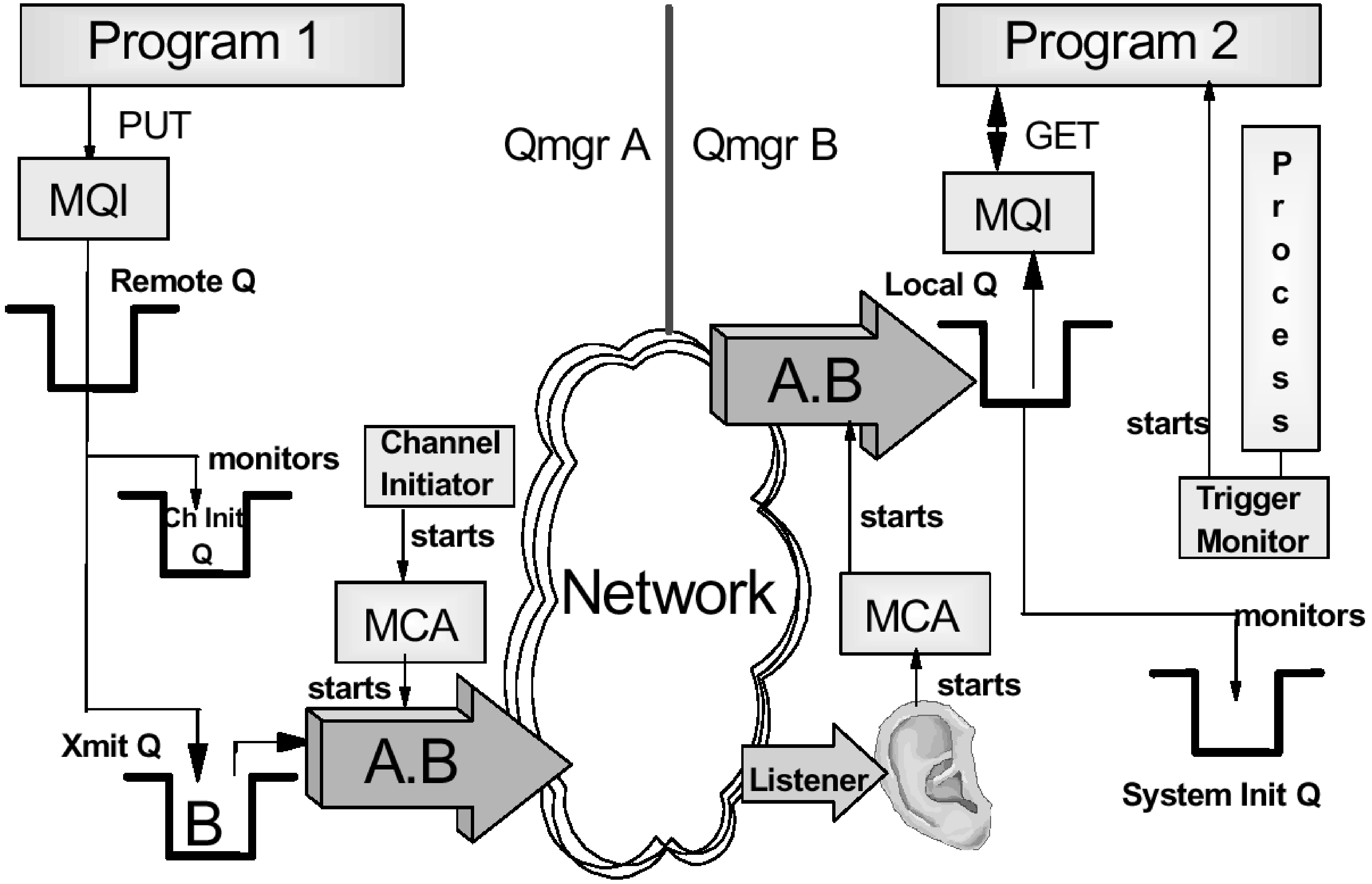
2. The queue manager checks to see if the conditions are met under which it has to generate a trigger event. They are, and a trigger event is generated. Information held within the associated process definition object is used when creating the trigger message.

3. The queue manager creates a trigger message and puts it on the initiation queue associated with this application queue, but only if an application (trigger monitor) has the initiation queue open for input.

4. The trigger monitor retrieves the trigger message from the initiation queue.

5. The trigger monitor issues a command to start application B (the server application).

6. Application B opens the application queue and retrieves the message.


Program 1
PUT
MQI
Remote Q
monitors
Ch Init Q
Channel Initiator
starts
MCA
starts
Xmit Q
B
A.B
Qmgr A
Qmgr B
Network
A.B
starts
MCA
Listener
starts
Program 2
GET
MQI
Local Q
Process
starts
Trigger Monitor
monitors
System Init Q

How MQSeries Works

The application program uses the Message Queue Interface (MQI) to communicate with the queue manager. 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. The figure above 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.

Überblick

Queues and Channels

Trigger

MQ – CICS Bridge

Message Segmenting and Grouping

In MQSeries messages can be *segmented* or *groupe*d. 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.

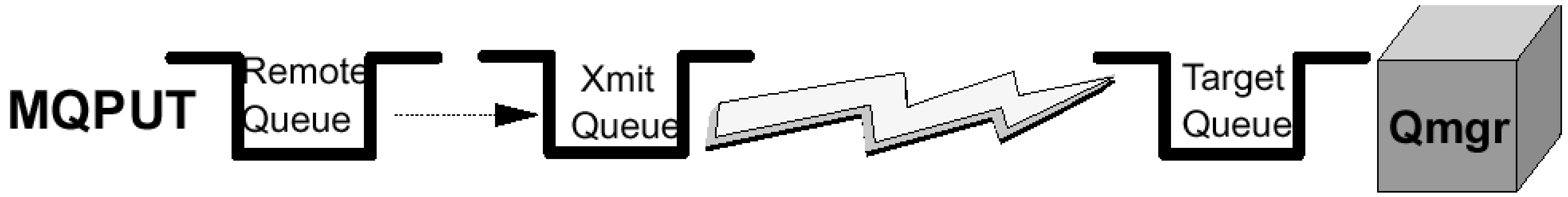
A second method of segmenting leaves the programmer in control so that he or she can split a message according to logical boundaries or buffer size available for the program. The programmer puts each segment as a separate physical message; thus several physical messages build one logical message. The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

Normal distributed processing



MQPUT to a Remote Queue

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 remote queue on the sender's side is a queue object. Its attributes contain definitions of the target queue.

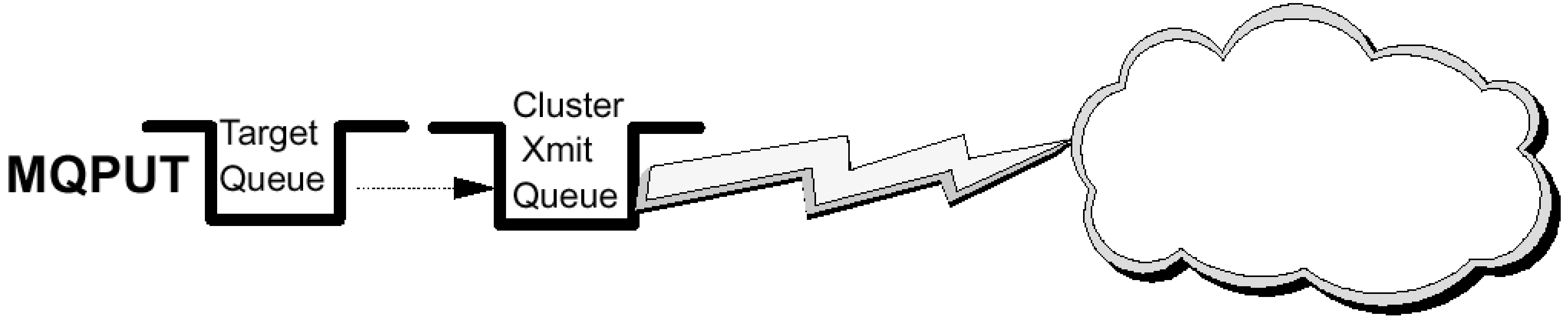
The message channel agents move the messages across the network and place them into the destination queues. The figure above shows the relationship between a transmission (Xmit) queue and the target queue manager.

Queue Manager Cluster

Queue manager clusters allow multiple instances of the *same* service (multiple instances of a queue with the same name) to be hosted through *multiple* queue managers.This allows for workload distribution. That is, the queue manager can send messages to different instances of an application.

Applications requesting a particular service can connect to any queue manager within the queue manager cluster. When applications make requests for the service, the queue manager to which they are connected automatically *workload balances* these requests across all available queue managers that host an instance of that service. This allows a *pool* of machines to exist within the queue manager cluster, each hosting a queue manager and the applications required to provide the service. Queue managers can dynamically join or leave the queue manager cluster to cope with varying loads placed on a particular service provided by a system.

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.

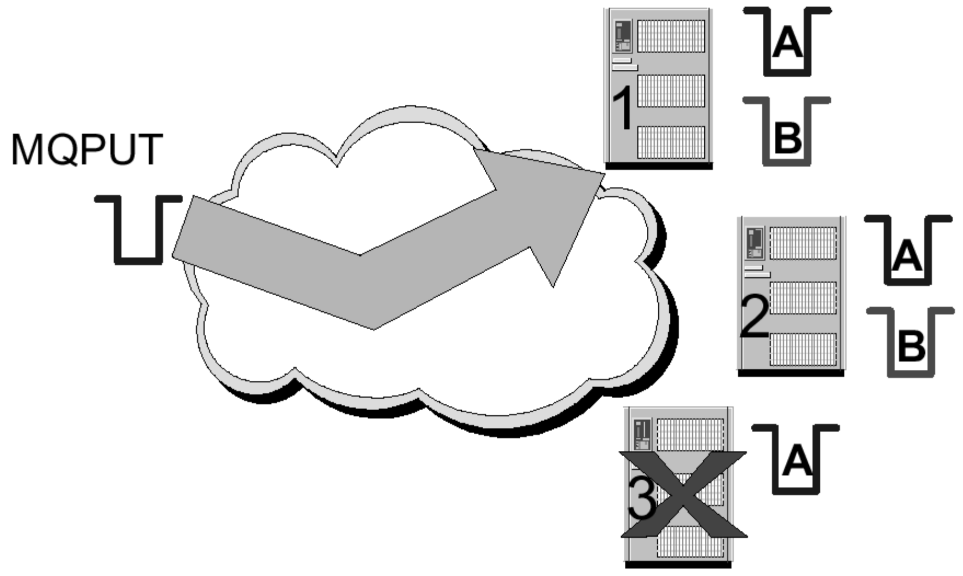


MQPUT to a Cluster Queue

With clustering, you send a message to a queue with a specific name somewhere in the cluster. In the above figure this is represented by a cloud. You specify the name of a target queue. 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.

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.



Accessing Cluster Queues

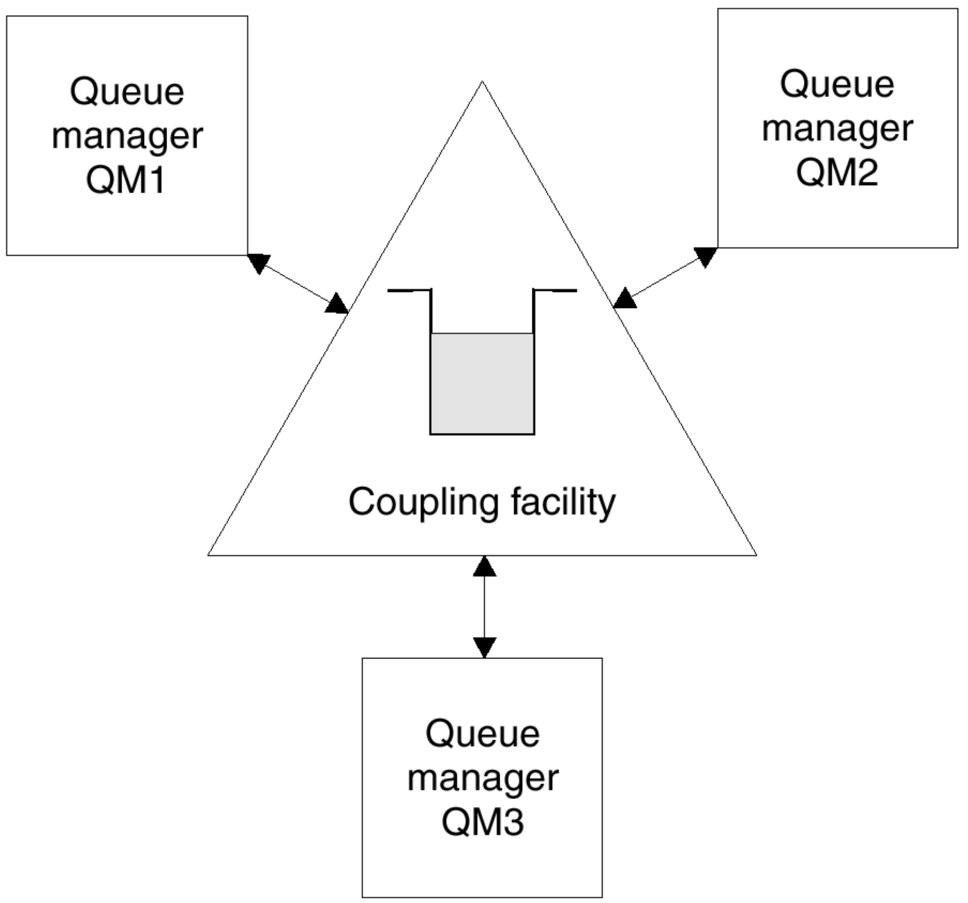
You create multiple instances of a queue by defining a queue with the same name on multiple queue managers that belong to the cluster. You must also name the cluster when you define the queue. When the application specifies only the queue name, where is the message sent?

This is shown above. MQSeries distributes the messages round-robin. You can change this default action by writing your own workload balancing routine.

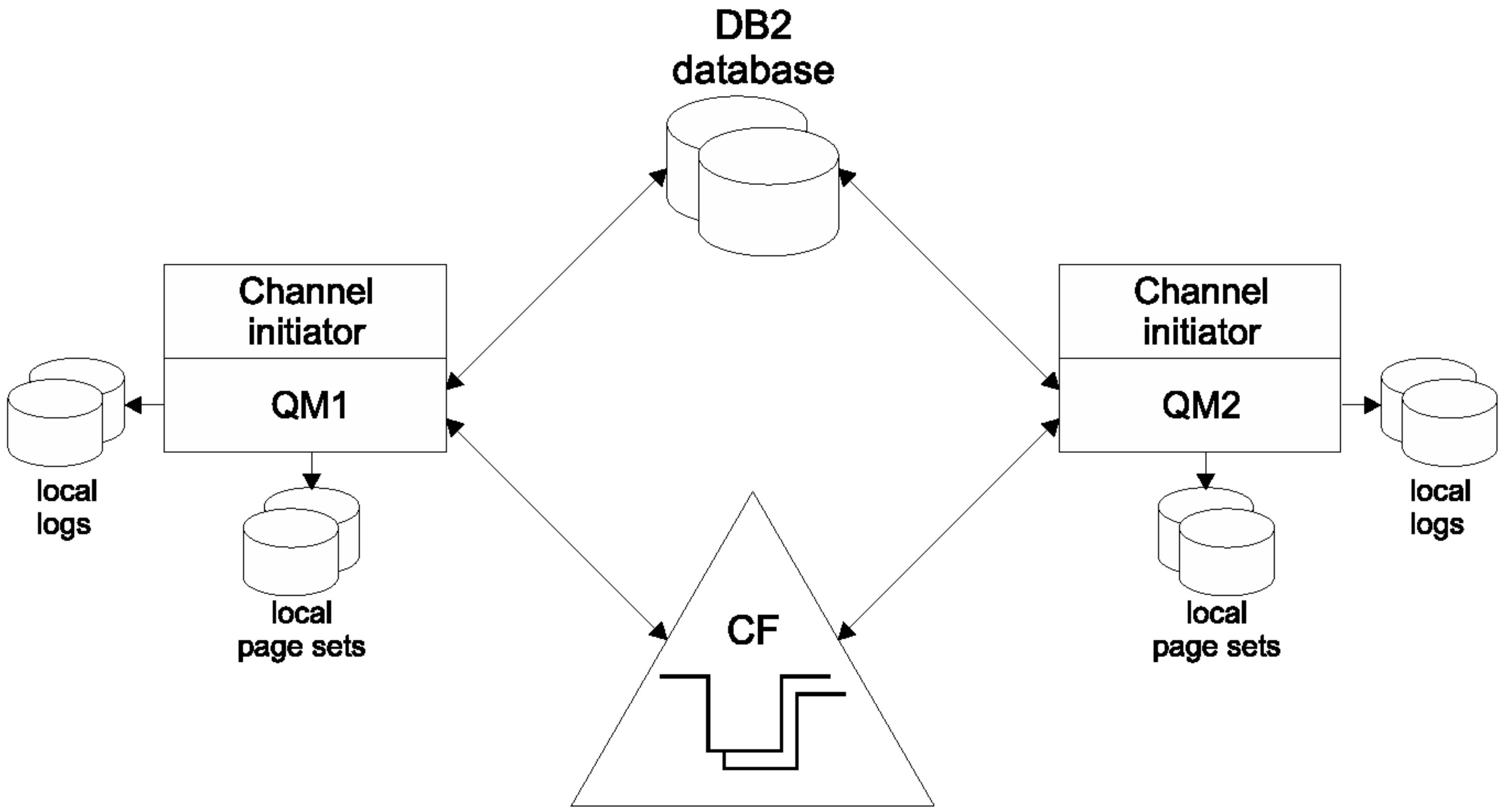
The figure shows messages put in one of the three cluster queues named A. Each of the three queue managers on the right owns a queue with this name. By default, the first message is placed in queue A on queue manager 1, the next in queue A on queue manager 2, the third goes to queue manager 3 and the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

If in the figure above queue A is available in system 3, but queue B is unavailable. System 3 is automatically excluded from receiving messages for its queue B.

Queue-sharing group



Multiple queue managers on multiple MVS images within the same queue-sharing group can MQPUT messages to and MQGET messages from the same shared queue. This is achieved by storing all the messages in a shared queue in the same coupling facility list structure.



Queue-sharing group

The group of queue managers that can access the same shared queues is called a queue-sharing group. Each member of the queue-sharing group has access to the same set of shared queues.

Queue-sharing groups have a name of up to four characters. The name must be unique in your network, and must be different from any queue manager names.

The figure above illustrates a queue-sharing group that contains two queue managers. Each queue manager has a channel initiator and its own local page sets and log data sets.

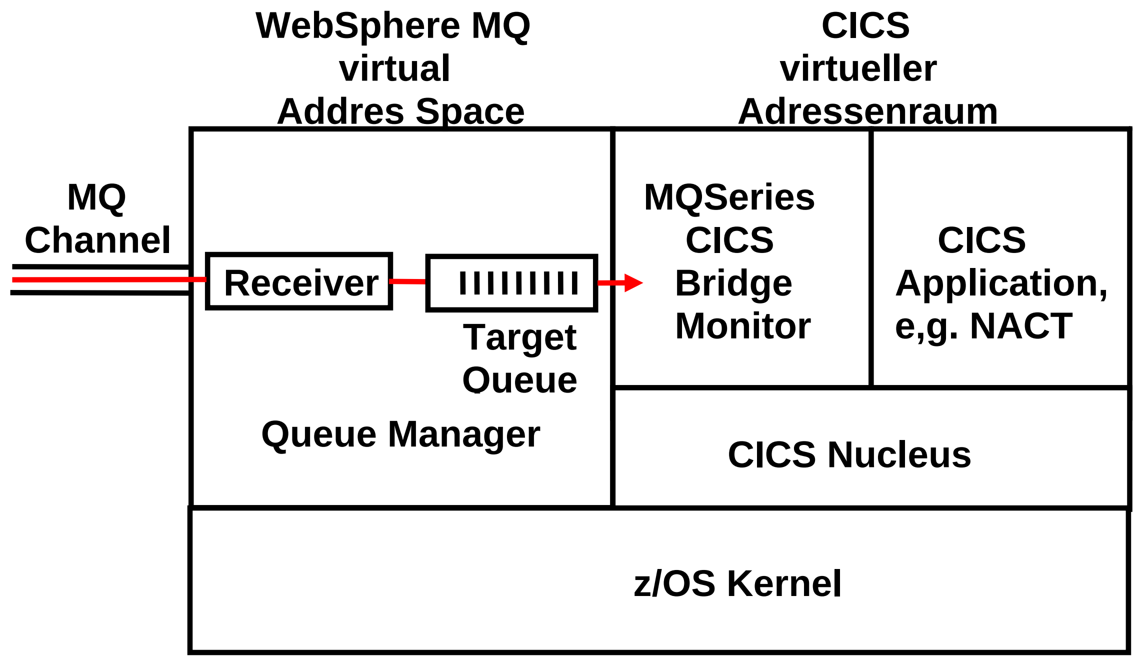
The CICS bridge

The WebSphere MQ-CICS bridge is the component of WebSphere MQ for z/OS that allows direct access from WebSphere MQ applications to applications on your CICS system. In bridge applications there are no WebSphere MQ calls within the CICS application (the bridge enables *implicit MQI suppor*t). This means that you can re-engineer legacy applications that were controlled by 3270-connected terminals to be controlled by WebSphere MQ messages, without having to rewrite, recompile, or re-link them.

The bridge enables an application that is not running in a CICS environment to run a *program* or *transaction* on CICS and get a response back. This non-CICS application can be run from any environment that has access to a WebSphere MQ network that encompasses WebSphere MQ for z/OS.

A *program* is a CICS program that can be invoked using the EXEC CICS LINK command. It must conform to the DPL subset of the CICS API, that is, it must not use CICS terminal or syncpoint facilities.

A *transaction* is a CICS transaction designed to run on a 3270 terminal. This transaction can use BMS commands. It can be conversational or part of a pseudoconversation. It is permitted to issue syncpoints.


WebSphere MQ virtual Addres Space
CICS virtueller Adressenraum
MQ Channel
Receiver
IIIIIIIII
Target Queue
Queue Manager
MQSeries CICS Bridge Monitor
CICS Application, e,g. NACT
CICS Nucleus
z/OS Kernel

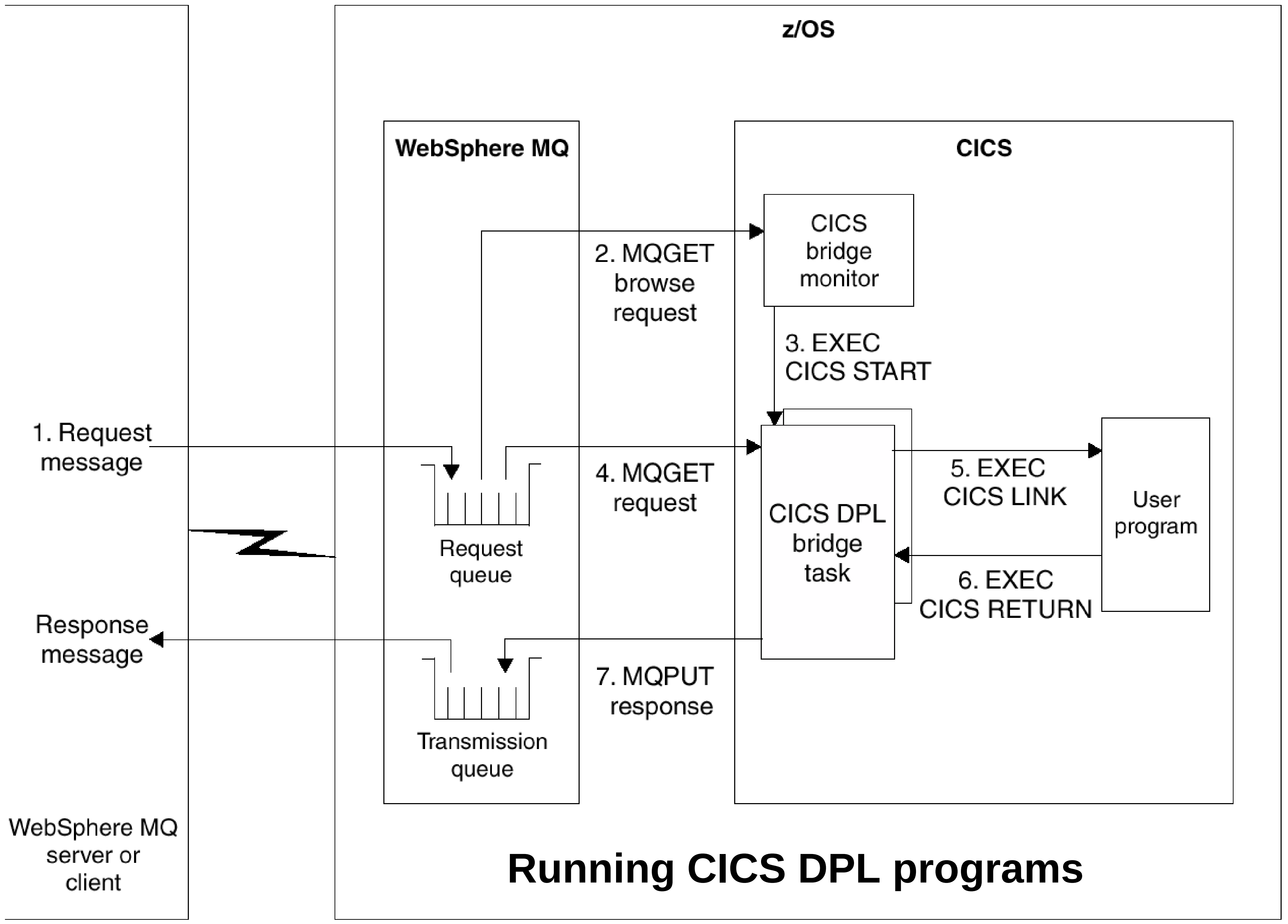
MQSeries Server Accessing CICS

System configuration for the CICS bridge:

When you are setting your system up, you should ensure that:

- **Both WebSphere MQ and CICS are running in the same z/OS image.**
- **The WebSphere MQ request queue is local to the CICS bridge, however the response queue can be local or remote.**
- **The CICS bridge tasks normally run in the same CICS as the bridge monitor. The user programs can be in the same or a different CICS system. CICS transaction routing can be used.**
- **The WebSphere MQ-CICS adapter is enabled.**
- **A specific CICS system is referenced in the Message header.**

When using shared queues, it is possible to use multiple bridge monitors on the CICS bridge in multiple CICS regions. However all bridge monitors must be associated with WebSphere MQ queue managers.



Running CICS DPL programs

Running CICS DPL programs

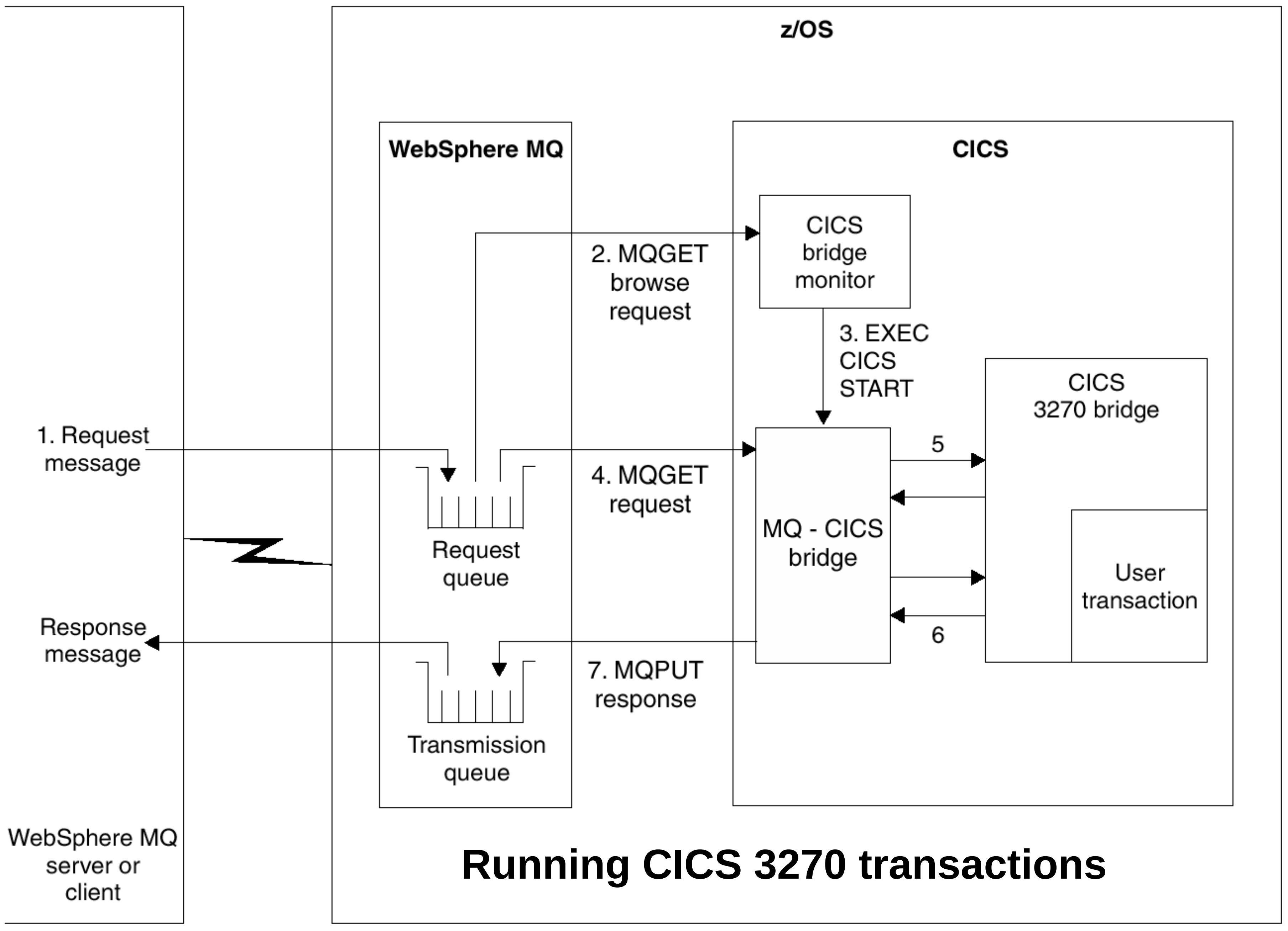
Data necessary to run the program is provided in the WebSphere MQ message. The bridge builds a COMMAREA from this data, and runs the program using EXEC CICS LINK. The figure below shows the sequence of actions taken to process a single message to run a CICS DPL program:

The following takes each step in turn, and explains what takes place:

1. A message, with a request to run a CICS program, is put on the request queue.
2. The CICS bridge monitor task, which is constantly browsing the queue, recognizes that a ‘start unit of work’ message is waiting.
3. Relevant authentication checks are made, and a CICS DPL bridge task is started with the appropriate authority, with a particular userid (depending on the options used to start the bridge monitor).
4. The CICS DPL bridge task removes the message from the request queue.
5. The CICS DPL bridge task builds a COMMAREA from the data in the message and issues an EXEC CICS LINK for the program requested in the message.
6. The program returns the response in the COMMAREA used by the request.
7. The CICS DPL bridge task reads the COMMAREA, creates a message, and puts it on the reply-to queue (transmission queue) specified in the request message. All response messages (normal and error, requests and replies) are put to the reply-to queue with default context.
8. The CICS DPL bridge task ends. If this is the last flow in the transaction then the transaction ends, if it is not the last message, the transaction waits until the next message is received or the specified timeout interval expires.

A unit of work can be just a single user program, or it can be multiple user programs. There is no limit to the number of messages you can send to make up a unit of work.

In this scenario, a unit of work made up of many messages works in the same way, with the exception that the CICS bridge task waits for the next request message in the final step unless it is the last message in the unit of work.



Running CICS 3270 transactions

Running CICS 3270 transactions

Data necessary to run the transaction is provided in the WebSphere MQ message. The CICS transaction runs as if it has a real 3270 terminal, but instead uses one or more MQ messages to communicate between the CICS transaction and the WebSphere MQ application

Unlike traditional 3270 emulators, the bridge does not work by replacing the VTAM flows with WebSphere MQ messages. Instead, the message consists of a number of parts called vectors, each of which corresponds to an EXEC CICS request. Therefore the application is talking directly to the CICS transaction, rather than through an emulator, using the actual data used by the transaction (known as application data structures or ADSs).

The following explains what takes place in each step :

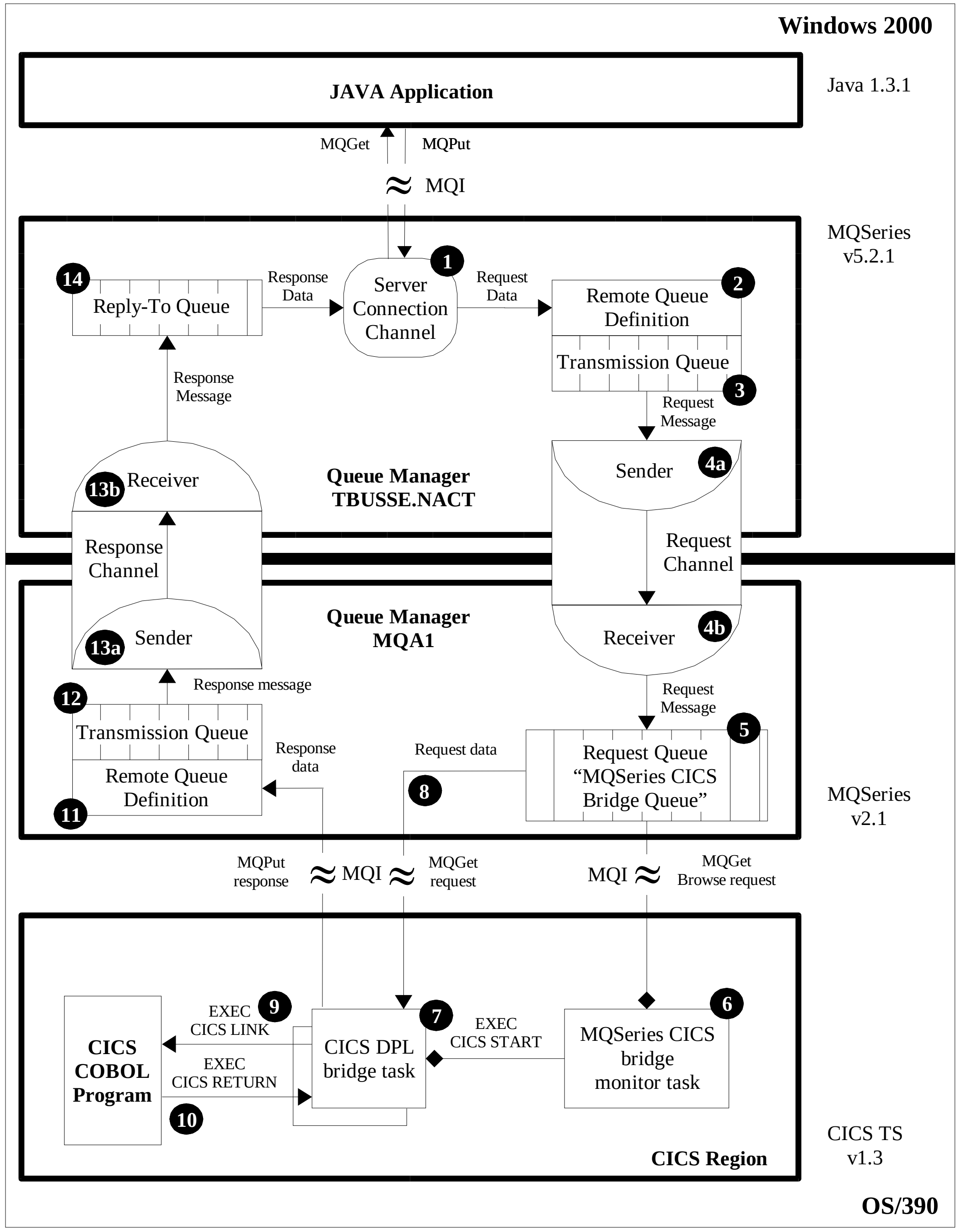
1. **A message, with a request to run a CICS transaction, is put on the request queue.**
2. **The CICS bridge monitor task, which is constantly browsing the queue, recognizes that a 'start unit of work' message is waiting .**
3. **Relevant authentication checks are made, and a CICS 3270 bridge task is started with the appropriate authority, with a particular userid (depending on the options used to start the bridge monitor).**
4. **The WebSphere MQ-CICS bridge removes the message from the queue and changes task to run a user transaction**
5. **Vectors in the message provide data to answer all terminal-related input EXEC CICS requests in the transaction.**
6. **Terminal-related output EXEC CICS requests result in output vectors being built.**
7. **The WebSphere MQ-CICS bridge builds all the output vectors into a single message and puts this on the reply-to queue.**
8. **The CICS 3270 bridge task ends. If this is the last flow in the transaction then the transaction ends, if it is not the last message, the transaction waits until the next message is received or the specified timeout interval expires.**

Note: The WebSphere MQ CICS bridge is a WebSphere MQ supplied CICS exit associated with the bridge transaction. It uses the CICS Link3270 mechanism.

A traditional CICS application usually consists of one or more transactions linked together as a pseudoconversation. In general, each transaction is started by the 3270 terminal user entering data onto the screen and pressing an AID key. This model of application can be emulated by a WebSphere MQ application. A message is built for the first transaction, containing information about the transaction, and input vectors. This is put on the queue. The reply message consists of the output vectors, the name of the next transaction to be run, and a token that is used to represent the pseudoconversation. The WebSphere MQ application builds a new input message, with the transaction name set to the next transaction and the facility token and remote system id set to the value returned on the previous message. Vectors for this second transaction are added to the message, and the message put on the queue. This process is continued until the application ends.

It is possible to include all of the WebSphere MQ messages for multiple transactions within the same bridge session which reduces monitoring overheads and improves performance.

An alternative approach to writing CICS applications is the conversational model. In this model, the original message might not contain all the data to run the transaction. If the transaction issues a request that cannot be answered by any of the vectors in the message, a message is put onto the reply-to queue requesting more data. The WebSphere MQ application gets this message and puts a new message back to the queue with a vector to satisfy the request.



MQSeries accessing CICS

The figure on the left is an example of program-to-program communication involving two systems.

Java Presentation logic runs on a workstation and accesses CICS COMMAREA on a mainframe.

We will do a tutorial with a Java presentation logic running on a client using the CICS MQSeries Bridge to access COMMAREA. The figure on the left is taken from this tutorial.