MESSAGE-ORIENTED COMMUNICATION

Remote procedure calls and remote object invocations contribute to hiding communication in distributed systems, that is, they enhance access transparency. Unfortunately, neither mechanism is always appropriate. In particular, when it cannot be assumed that the receiving side is executing at the time a request is issued, alternative communication services are needed. Likewise, the inherent synchronous nature of RPCs, by which a client is blocked until its request has been processed, sometimes needs to be replaced by something else.
That something else is messaging. In this section we concentrate on message-oriented communication in distributed systems by first taking a closer look at what exactly synchronous behavior is and what its implications are.

Message-Oriented Transient Communication

Many distributed systems and applications are built directly on top of the simple message-oriented model offered by the transport layer. To better understand and appreciate the message-oriented systems as part of middleware solutions.

Berkeley Sockets

Special attention has been paid to standardizing the interface of the transport layer to allow programmers to make use of its entire suite of (messaging) protocols through a simple set of primitives. Also, standard interfaces make it easier to port an application to a different machine.
Another important interface is XTI, which stands for the X/Open Transport Interface, formerly called the Transport Layer Interface (TLI), and developed by AT&T. Sockets and XTI are very similar in their model of network programming, but differ in their set of primitives.
Conceptually, a socket is a communication end point to which an application can write data that are to be sent out over the underlying network, and from which incoming data can be read. A socket forms an abstraction over the actual communication end point that is used by the local operating system for a specific transport protocol.

Figure 4-14. The socket primitives for TCP/IP.

Primitive	Meaning
Socket	Create a new communication end point
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Servers generally execute the first four primitives, normally in the order given. When calling the socket primitive, the caller creates a new communication end point for a specific transport protocol. Internally, creating a communication end point means that the local operating system reserves resources to accommodate sending and receiving messages for the specified protocol.

The bind primitive associates a local address with the newly-created socket. For example, a server should bind the IP address of its machine together with a (possibly well-known) port number to a socket. Binding tells the operating system that the server wants to receive messages only on the specified address and port

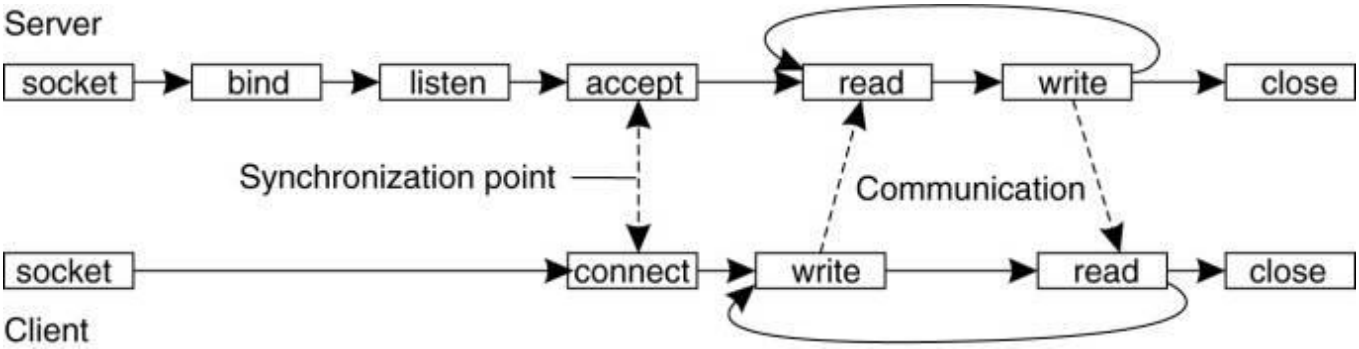
The listen primitive is called only in the case of connection-oriented communication. It is a non-blocking call that allows the local operating system to reserve enough buffers for a specified maximum number of connections that the caller is willing to accept.

A call to accept blocks the caller until a connection request arrives. When a request arrives, the local operating system creates a new socket with the same properties as the original one, and returns it to the caller. This approach will allow the server to, for example, fork off a process that will subsequently handle the actual communication through the new connection. The server, in the meantime, can go back and wait for another connection request on the original socket.

Let us now take a look at the client side. Here, too, a socket must first be created using the socket primitive, but explicitly binding the socket to a local address is not necessary, since the operating system can dynamically allocate a port when the connection is set up. The connect primitive requires that the caller specifies the transport-level address to which a connection request is to be sent. The client is blocked until a connection has been set up successfully, after which both sides can start exchanging information through the send and receive primitives. Finally, closing a connection is symmetric when using sockets, and is established by having both the

client and server call the close primitive. The general pattern followed by a client and server for connection-oriented communication using sockets is shown in Fig. 4-15.

Figure 4-15. Connection-oriented communication pattern using sockets.



The Message-Passing Interface (MPI)

With the advent of high-performance multi-computers, developers have been looking for message-oriented primitives that would allow them to easily write highly efficient applications. This means that the primitives should be at a convenient level of abstraction (to ease application development), and that their implementation incurs only minimal overhead. Sockets were deemed insufficient for two reasons. First, they were at the wrong level of abstraction by supporting only simple send and receive primitives. Second, sockets had been designed to communicate across networks using general-purpose protocol stacks such as TCP/IP. They were not considered suitable for the proprietary protocols developed for high-speed interconnection networks, such as those used in high-performance server clusters. Those protocols required an interface that could handle more advanced features, such as different forms of buffering and synchronization.

The result was that most interconnection networks and high-performance multi-computers were shipped with proprietary communication libraries. These libraries offered a wealth of high-level and generally efficient communication primitives. Of course, all libraries were mutually incompatible, so that application developers now had a portability problem.

The need to be hardware and platform independent eventually led to the definition of a standard for message passing, simply called the Message-Passing Interface or MPI. MPI is designed for parallel applications and as such is tailored to transient communication. It makes direct use of the underlying network. Also, it assumes that serious failures such as process crashes or network partitions are fatal and do not require automatic recovery.

MPI assumes communication takes place within a known group of processes. Each group is assigned an identifier. Each process within a group is also assigned a (local) identifier. A (groupID, processID) pair therefore uniquely identifies the source or destination of a message, and is used instead of a transport-level

address. There may be several, possibly overlapping groups of processes involved in a computation and that are all executing at the same time.

At the core of MPI are messaging primitives to support transient communication, of which the most intuitive ones are summarized in Fig. 4-16.

Figure 4-16. Some of the most intuitive message-passing primitives of MPI.

Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there is none
MPI_irecv	Check if there is an incoming message, but do not block

Transient asynchronous communication is supported by means of the MPI_bsend primitive. The sender submits a message for transmission, which is generally first copied to a local buffer in the MPI runtime system. When the message has been copied, the sender continues. The local MPI runtime system will remove the message from its local buffer and take care of transmission as soon as a receiver has called a receive primitive.

There is also a blocking send operation, called MPI_send, of which the semantics are implementation dependent. The primitive MPI_send may either block the caller until the specified message has been copied to the MPI runtime system at the sender's side, or until the receiver has initiated a receive operation. Synchronous communication by which the sender blocks until its request is accepted for further processing is available through the MPI_ssend primitive. Finally, the strongest form of synchronous communication is also supported: when a sender calls MPI_sendrecv, it sends a request to the receiver and blocks until the latter returns a reply. Basically, this primitive corresponds to a normal RPC.

Both MPI_send and MPI_ssend have variants that avoid copying messages from user buffers to buffers internal to the local MPI runtime system. These variants correspond to a form of asynchronous communication. With MPI_isend, a sender passes a pointer to the message after which the MPI runtime system takes care of communication. The sender immediately continues. To prevent overwriting the message before communication completes, MPI offers primitives to check for completion, or even to block if required. As with MPI_send,

whether the message has actually been transferred to the receiver or that it has merely been copied by the local MPI runtime system to an internal buffer is left unspecified.

Likewise, with MPI_issend, a sender also passes only a pointer to the MPI runtime system. When the runtime system indicates it has processed the message, the sender is then guaranteed that the receiver has accepted the message and is now working on it.

The operation MPI_recv is called to receive a message; it blocks the caller until a message arrives. There is also an asynchronous variant, called MPI_irecv, by which a receiver indicates that is prepared to accept a message. The receiver can check whether or not a message has indeed arrived, or block until one does.

The semantics of MPI communication primitives are not always straightforward, and different primitives can sometimes be interchanged without affecting the correctness of a program. The official reason why so many different forms of communication are supported is that it gives implementers of MPI systems enough possibilities for optimizing performance. Cynics might say the committee could not make up its collective mind, so it threw in everything. MPI has been designed for high-performance parallel applications, which makes it easier to understand its diversity in different communication primitives.

Message-Oriented Persistent Communication

We now come to an important class of message-oriented middleware services, generally known as message-queuing systems, or just Message-Oriented Middleware (MOM). Message-queuing systems provide extensive support for persistent asynchronous communication. The essence of these systems is that they offer intermediate-term storage capacity for messages, without requiring either the sender or receiver to be active during message transmission. An important difference with Berkeley sockets and MPI is that message-queuing systems are typically targeted to support message transfers that are allowed to take minutes instead of seconds or milliseconds. We first explain a general approach to message-queuing systems, and conclude this section by comparing them to more traditional systems, notably the Internet e-mail systems.

Message-Queuing Model

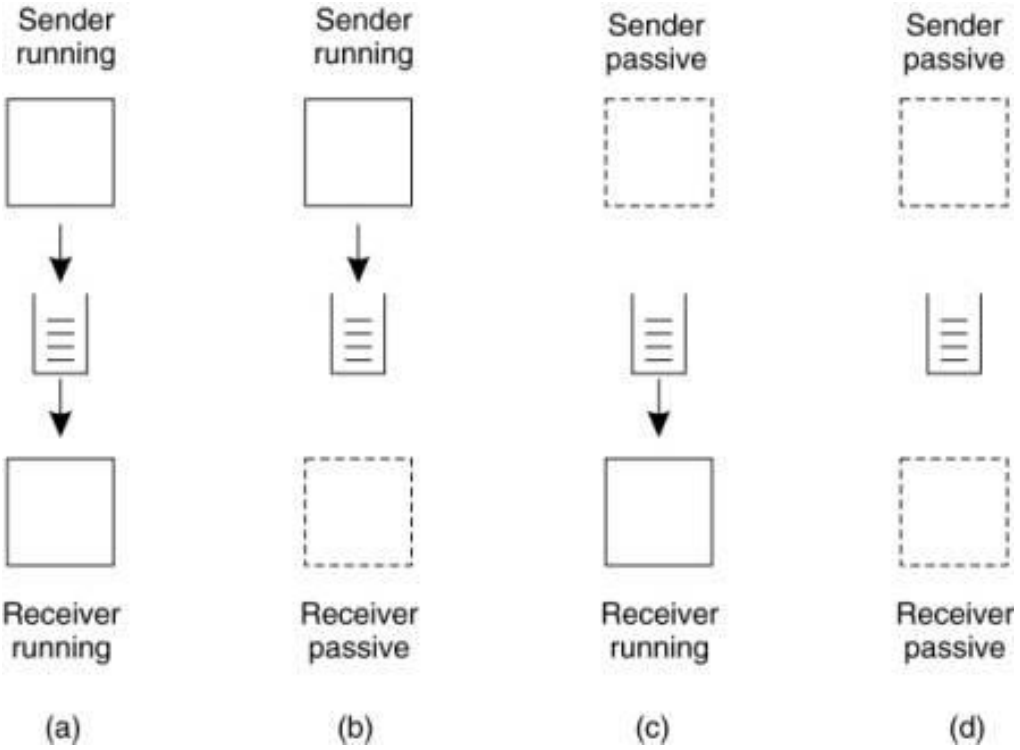
The basic idea behind a message-queuing system is that applications communicate by inserting messages in specific queues. These messages are forwarded over a series of communication servers and are eventually delivered to the destination, even if it was down when the message was sent. In practice, most communication servers are directly connected to each other. In other words, a message is generally transferred directly to a destination server. In principle, each application has its own private queue to which other applications can send

messages. A queue can be read only by its associated application, but it is also possible for multiple applications to share a single queue.

An important aspect of message-queuing systems is that a sender is generally given only the guarantees that its message will eventually be inserted in the recipient's queue. No guarantees are given about when, or even if the message will actually be read, which is completely determined by the behavior of the recipient.

These semantics permit communication loosely-coupled in time. There is thus no need for the receiver to be executing when a message is being sent to its queue. Likewise, there is no need for the sender to be executing at the moment its message is picked up by the receiver. The sender and receiver can execute completely independently of each other. In fact, once a message has been deposited in a queue, it will remain there until it is removed, irrespective of whether its sender or receiver is executing. This gives us four combinations with respect to the execution mode of the sender and receiver, as shown in Fig. 4-17.

Figure 4-17. Four combinations for loosely-coupled communications using queues.



In Fig. 4-17(a), both the sender and receiver execute during the entire transmission of a message. In Fig. 4-17(b), only the sender is executing, while the receiver is passive, that is, in a state in which message delivery is not possible. Nevertheless, the sender can still send messages. The combination of a passive sender and an executing receiver is shown in Fig. 4-17(c). In this case, the receiver can read messages that were sent to it, but it is not necessary that their respective senders are executing as well. Finally, in Fig. 4-17(d), we see the situation that the system is storing (and possibly transmitting) messages even while sender and receiver are passive.

Messages can, in principle, contain any data. The only important aspect from the perspective of middleware is that messages are properly addressed. In practice, addressing is done by providing a system-wide unique name of the destination queue. In some cases, message size may be limited, although it is also possible that the underlying system takes care of fragmenting and assembling large messages in a way that is completely transparent to applications. An effect of this approach is that the basic interface offered to applications can be extremely simple, as shown in Fig. 4-18.

Figure 4-18. Basic interface to a queue in a message-queuing system.

Primitive	Meaning
Put	Append a message to a specified queue
Get	Block until the specified queue is nonempty, and remove the first message
Poll	Check a specified queue for messages, and remove the first. Never block
Notify	Install a handler to be called when a message is put into the specified queue

The put primitive is called by a sender to pass a message to the underlying system that is to be appended to the specified queue. As we explained, this is a nonblocking call. The get primitive is a blocking call by which an authorized process can remove the longest pending message in the specified queue. The process is blocked only if the queue is empty. Variations on this call allow searching for a specific message in the queue, for example, using a priority, or a matching pattern. The non-blocking variant is given by the poll primitive. If the queue is empty, or if a specific message could not be found, the calling process simply continues.

Finally, most queuing systems also allow a process to install a handler as a callback function, which is automatically invoked whenever a message is put into the queue. Callbacks can also be used to automatically start a process that will fetch messages from the queue if no process is currently executing. This approach is often implemented by means of a daemon on the receiver's side that continuously monitors the queue for incoming messages and handles accordingly.

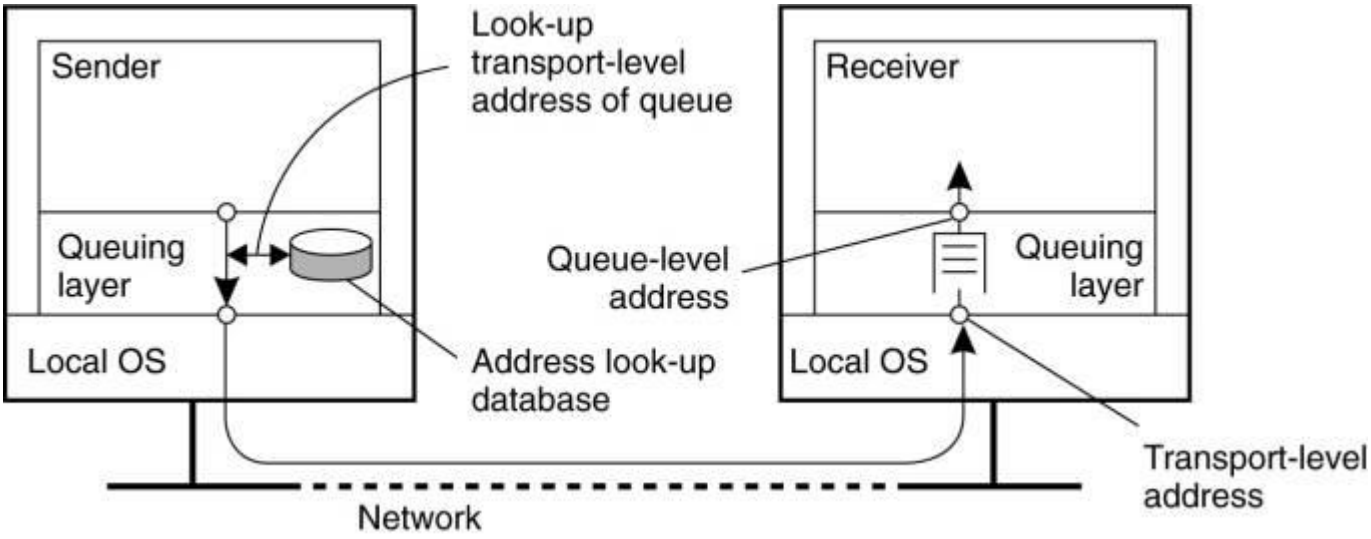
General Architecture of a Message-Queuing System

Let us now take a closer look at what a general message-queuing system looks like. One of the first restrictions that we make is that messages can be put only into queues that are local to the sender, that is, queues on the same machine, or no worse than on a machine nearby such as on the same LAN that can be efficiently reached through an RPC. Such a queue is called the source queue. Likewise, messages can be read only from local queues. However, a message put into a queue will contain the specification of a destination queue to which it

should be transferred. It is the responsibility of a message-queuing system to provide queues to senders and receivers and take care that messages are transferred from their source to their destination queue.

It is important to realize that the collection of queues is distributed across multiple machines. Consequently, for a message-queuing system to transfer messages, it should maintain a mapping of queues to network locations. In practice, this means that it should maintain a (possibly distributed) database of queue names to network locations, as shown in Fig. 4-19. Note that such a mapping is completely analogous to the use of the Domain Name System (DNS) for e-mail in the Internet. For example, when sending a message to the logical mail address steen@cs.vu.nl, the mailing system will query DNS to find the network (i.e., IP) address of the recipient's mail server to use for the actual message transfer.

Figure 4-19. The relationship between queue-level addressing and network-level addressing.

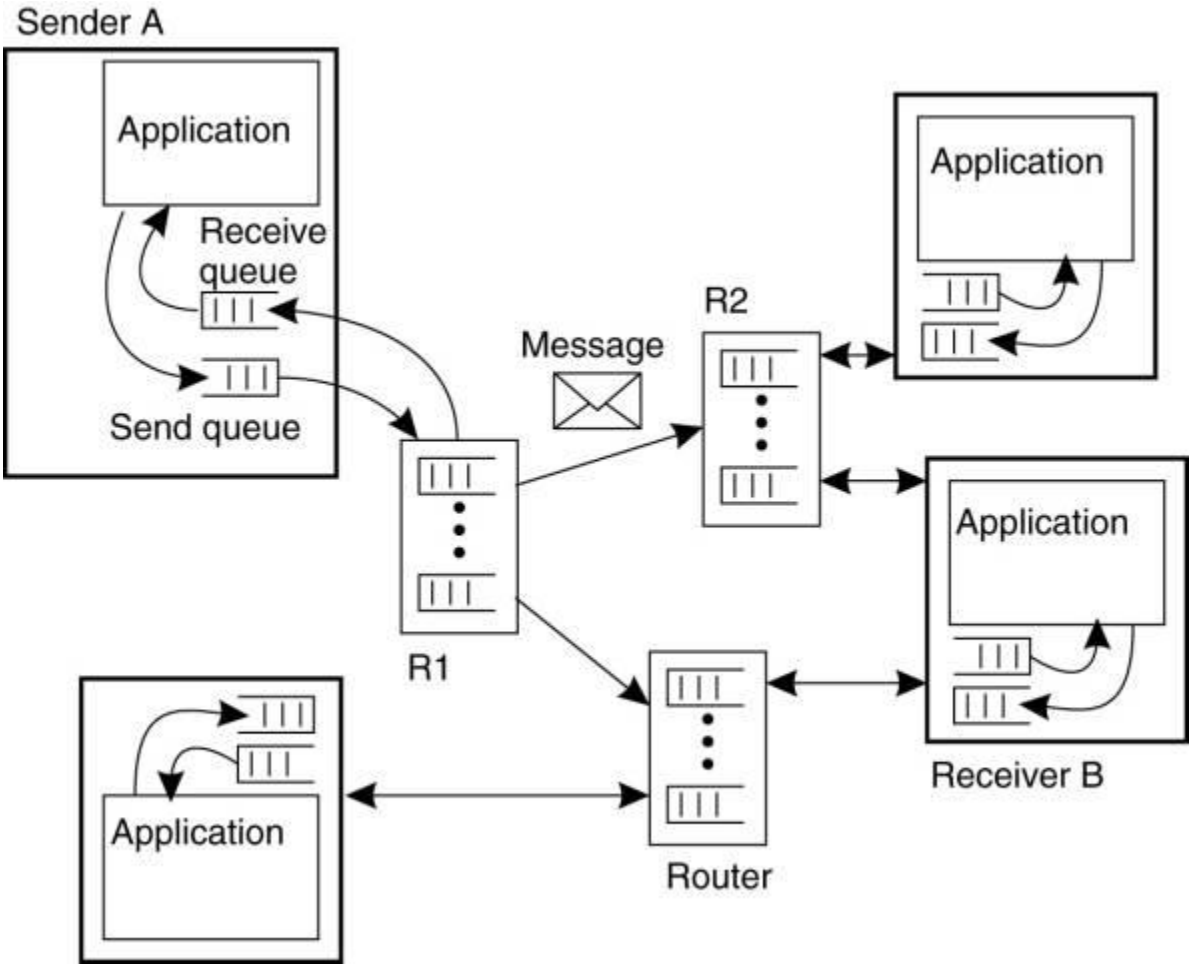


Queues are managed by queue managers. Normally, a queue manager interacts directly with the application that is sending or receiving a message. However, there are also special queue managers that operate as routers, or relays: they forward incoming messages to other queue managers. In this way, a message-queuing system may gradually grow into a complete, application-level, overlay network, on top of an existing computer network. This approach is similar to the construction of the early MBone over the Internet, in which ordinary user processes were configured as multicast routers.

Relays can be convenient for a number of reasons. For example, in many message-queuing systems, there is no general naming service available that can dynamically maintain queue-to-location mappings. Instead, the topology of the queuing network is static, and each queue manager needs a copy of the queue-to location mapping. It is needless to say that in large-scale queuing systems, this approach can easily lead to network-management problems.

One solution is to use a few routers that know about the network topology. When a sender A puts a message for destination B in its local queue, that message is first transferred to the nearest router, say R1, as shown in Fig. 4-20. At that point, the router knows what to do with the message and forwards it in the direction of B. For example, R1 may derive from B's name that the message should be forwarded to router R2. In this way, only the routers need to be updated when queues are added or removed, while every other queue manager has to know only where the nearest router is.

Figure 4-20. The general organization of a message-queuing system with routers.



Relays can thus generally help build scalable message-queuing systems. However, as queuing networks grow, it is clear that the manual configuration of networks will rapidly become completely unmanageable. The only solution is to adopt dynamic routing schemes as is done for computer networks. In that respect, it is somewhat surprising that such solutions are not yet integrated into some of the popular message-queuing systems.

Another reason why relays are used is that they allow for secondary processing of messages. For example, messages may need to be logged for reasons of security or fault tolerance.

Finally, relays can be used for multicasting purposes. In that case, an incoming message is simply put into each send queue.

Message Brokers

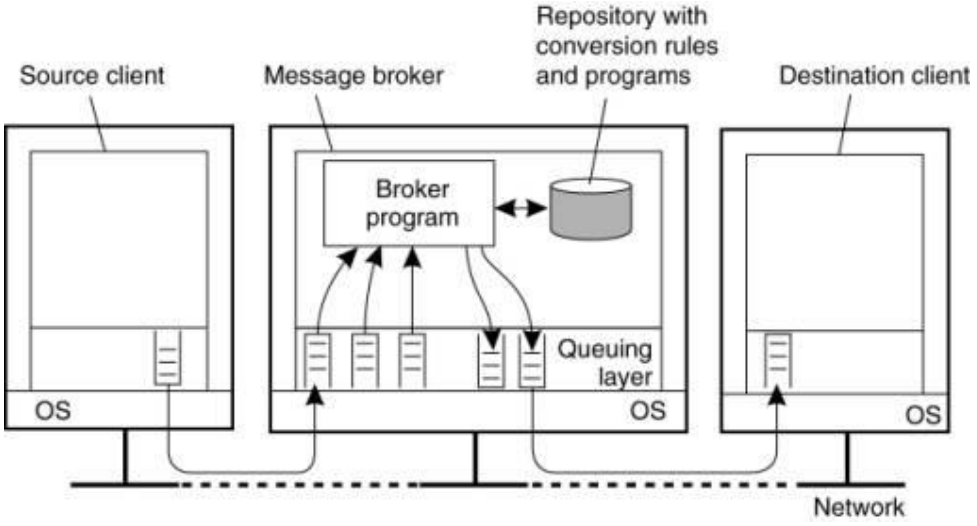
An important application area of message-queuing systems is integrating existing and new applications into a single, coherent distributed information system. Integration requires that applications can understand the messages they receive. In practice, this requires the sender to have its outgoing messages in the same format as that of the receiver.

The problem with this approach is that each time an application is added to the system that requires a separate message format, each potential receiver will have to be adjusted in order to produce that format.

An alternative is to agree on a common message format, as is done with traditional network protocols. Unfortunately, this approach will generally not work for message-queuing systems. The problem is the level of abstraction at which these systems operate. A common message format makes sense only if the collection of processes that make use of that format indeed have enough in common. If the collection of applications that make up a distributed information system is highly diverse (which it often is), then the best common format may well be no more than a sequence of bytes.

Although a few common message formats for specific application domains have been defined, the general approach is to learn to live with different formats, and try to provide the means to make conversions as simple as possible. In message-queuing systems, conversions are handled by special nodes in a queuing network, known as message brokers. A message broker acts as an application-level gateway in a message-queuing system. Its main purpose is to convert incoming messages so that they can be understood by the destination application. Note that to a message-queuing system, a message broker is just another application, as shown in Fig. 4-21. In other words, a message broker is generally not considered to be an integral part of the queuing system.

Figure 4-21. The general organization of a message broker in a message-queuing system.



A message broker can be as simple as a reformatter for messages. For example, assume an incoming message contains a table from a database, in which records are separated by a special end-of-record delimiter and fields within a record have a known, fixed length. If the destination application expects a different delimiter between records, and also expects that fields have variable lengths, a message broker can be used to convert messages to the format expected by the destination.

In a more advanced setting, a message broker may act as an application-level gateway, such as one that handles the conversion between two different database applications. In such cases, frequently it cannot be guaranteed that all information contained in the incoming message can actually be transformed into something appropriate for the outgoing message.

In this case, rather than (only) converting messages, a broker is responsible for matching applications based on the messages that are being exchanged. In such a model, called publish/subscribe, applications send messages in the form of publishing. In particular, they may publish a message on topic X, which is then sent to the broker. Applications that have stated their interest in messages on topic X, that is, who have subscribed to those messages, will then receive these messages from the broker.

At the heart of a message broker lies a repository of rules and programs that can transform a message of type T1 to one of type T2. The problem is defining the rules and developing the programs. Most message broker products come with sophisticated development tools, but the bottom line is still that the repository needs to be filled by experts. Here we see a perfect example where commercial products are often misleadingly said to provide "intelligence," where, in fact, the only intelligence is to be found in the heads of those experts.

A Note on Message-Queuing Systems

Considering what we have said about message-queuing systems, it would appear that they have long existed in the form of implementations for e-mail services. E-mail systems are generally implemented through a collection of mail servers that store and forward messages on behalf of the users on hosts directly connected to the server. Routing is generally left out, as e-mail systems can make direct use of the underlying transport services. For example, in the mail protocol for the Internet, SMTP, a message is transferred by setting up a direct TCP connection to the destination mail server.

What makes e-mail systems special compared to message-queuing systems is that they are primarily aimed at providing direct support for end users. This explains, for example, why a number of groupware applications are based directly on an e-mail system. In addition, e-mail systems may have very specific requirements such as

automatic message filtering, support for advanced messaging databases (e.g., to easily retrieve previously stored messages), and so on.

General message-queuing systems are not aimed at supporting only end users. An important issue is that they are set up to enable persistent communication between processes, regardless of whether a process is running a user application, handling access to a database, performing computations, and so on. This approach leads to a different set of requirements for message-queuing systems than pure email systems. For example, e-mail systems generally need not provide guaranteed message delivery, message priorities, logging facilities, efficient multicasting, load balancing, fault tolerance, and so on for general usage.

General-purpose message-queuing systems, therefore, have a wide range of applications, including e-mail, workflow, groupware, and batch processing. However, as we have stated before, the most important application area is the integration of a (possibly widely-dispersed) collection of databases and applications into a federated information system. For example, a query expanding several databases may need to be split into sub-queries that are forwarded to individual databases. Message-queuing systems assist by providing the basic means to package each sub-query into a message and routing it to the appropriate database.

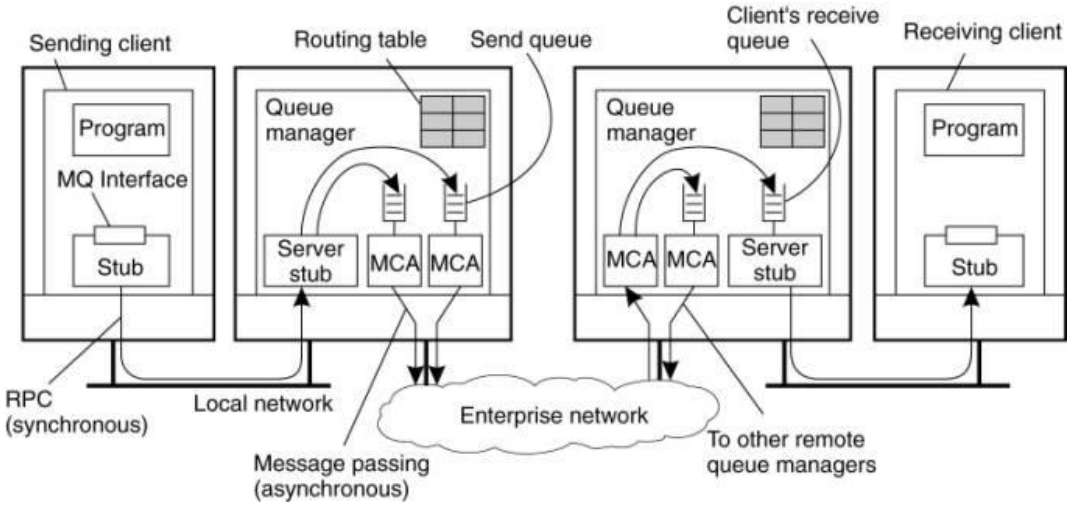
Example: IBM's Web-Sphere Message-Queuing System

To help understand how message-queuing systems work in practice, let us take a look at one specific system, namely the message-queuing system that is part of IBM's WebSphere product. Formerly known as MQSeries, it is now referred to as WebSphere MQ. There is a wealth of documentation on Web-Sphere MQ, and in the following we can only resort to the basic principles. Many architectural details concerning message-queuing networks can be found in IBM. Programming message-queuing networks is not something that can be learned on a Sunday afternoon, and MQ's programming guide is a good example showing that going from principles to practice may require substantial effort.

Overview

The basic architecture of an MQ queuing network is quite straightforward, and is shown in Fig. 4-22. All queues are managed by queue managers. A queue manager is responsible for removing messages from its send queues, and forwarding those to other queue managers. Likewise, a queue manager is responsible for handling incoming messages by picking them up from the underlying network and subsequently storing each message in the appropriate input queue. To give an impression of what messaging can mean: a message has a maximum default size of 4 MB, but this can be increased up to 100 MB. A queue is normally restricted to 2 GB of data, but depending on the underlying operating system, this maximum can be easily set higher.

Figure 4-22. General organization of IBM's message-queuing system.



Queue managers are pairwise connected through message channels, which are an abstraction of transport-level connections. A message channel is a unidirectional, reliable connection between a sending and a receiving queue manager, through which queued messages are transported. For example, an Internet-based message channel is implemented as a TCP connection. Each of the two ends of a message channel is managed by a message channel agent (MCA). A sending MCA is basically doing nothing else than checking send queues for a message, wrapping it into a transport-level packet, and sending it along the connection to its associated receiving MCA. Likewise, the basic task of a receiving MCA is listening for an incoming packet, unwrapping it, and subsequently storing the unwrapped message into the appropriate queue.

Queue managers can be linked into the same process as the application for which it manages the queues. In that case, the queues are hidden from the application behind a standard interface, but effectively can be directly manipulated by the application. An alternative organization is one in which queue managers and applications run on separate machines. In that case, the application is offered the same interface as when the queue manager is colocated on the same machine. However, the interface is implemented as a proxy that communicates with the queue manager using traditional RPC-based synchronous communication. In this way, MQ basically retains the model that only queues local to an application can be accessed.

Channels

An important component of MQ is formed by the message channels. Each message channel has exactly one associated send queue from which it fetches the messages it should transfer to the other end. Transfer along the channel can take place only if both its sending and receiving MCA are up and running.

One alternative is to have an application directly start its end of a channel by activating the sending or receiving MCA. However, from a transparency point of view, this is not a very attractive alternative. A better approach to start a sending MCA is to configure the channel's send queue to set off a trigger when a message is first put into

the queue. That trigger is associated with a handler to start the sending MCA so that it can remove messages from the send queue.

Another alternative is to start an MCA over the network. In particular, if one side of a channel is already active, it can send a control message requesting that the other MCA to be started. Such a control message is sent to a daemon listening to a well-known address on the same machine as where the other MCA is to be started.

Channels are stopped automatically after a specified time has expired during which no more messages were dropped into the send queue.

Each MCA has a set of associated attributes that determine the overall behavior of a channel. Some of the attributes are listed in Fig. 4-23. Attribute values of the sending and receiving MCA should be compatible and perhaps negotiated first before a channel can be set up. For example, both MCAs should obviously support the same transport protocol. An example of a nonnegotiable attribute is whether or not messages are to be delivered in the same order as they are put into the send queue. If one MCA wants FIFO delivery, the other must comply. An example of a negotiable attribute value is the maximum message length, which will simply be chosen as the minimum value specified by either MCA.

Figure 4-23. Some attributes associated with message channel agents.

Attribute	Description
Transport type	Determines the transport protocol to be used
FIFO delivery	Indicates that messages are to be delivered in the order they are sent
Message length	Maximum length of a single message
Setup retry count	Specifies maximum number of retries to start up the remote MCA
Delivery retries	Maximum times MCA will try to put received message into queue

Message Transfer

To transfer a message from one queue manager to another (possibly remote) queue manager, it is necessary that each message carries its destination address, for which a transmission header is used. An address in MQ consists of two parts. The first part consists of the name of the queue manager to which the message is to be delivered. The second part is the name of the destination queue resorting under that manager to which the message is to be appended.

Besides the destination address, it is also necessary to specify the route that a message should follow. Route specification is done by providing the name of the local send queue to which a message is to be appended. Thus it is not necessary to provide the full route in a message. Recall that each message channel has exactly one send

queue. By telling to which send queue a message is to be appended, we efectively specify to which queue manager a message is to be forwarded.

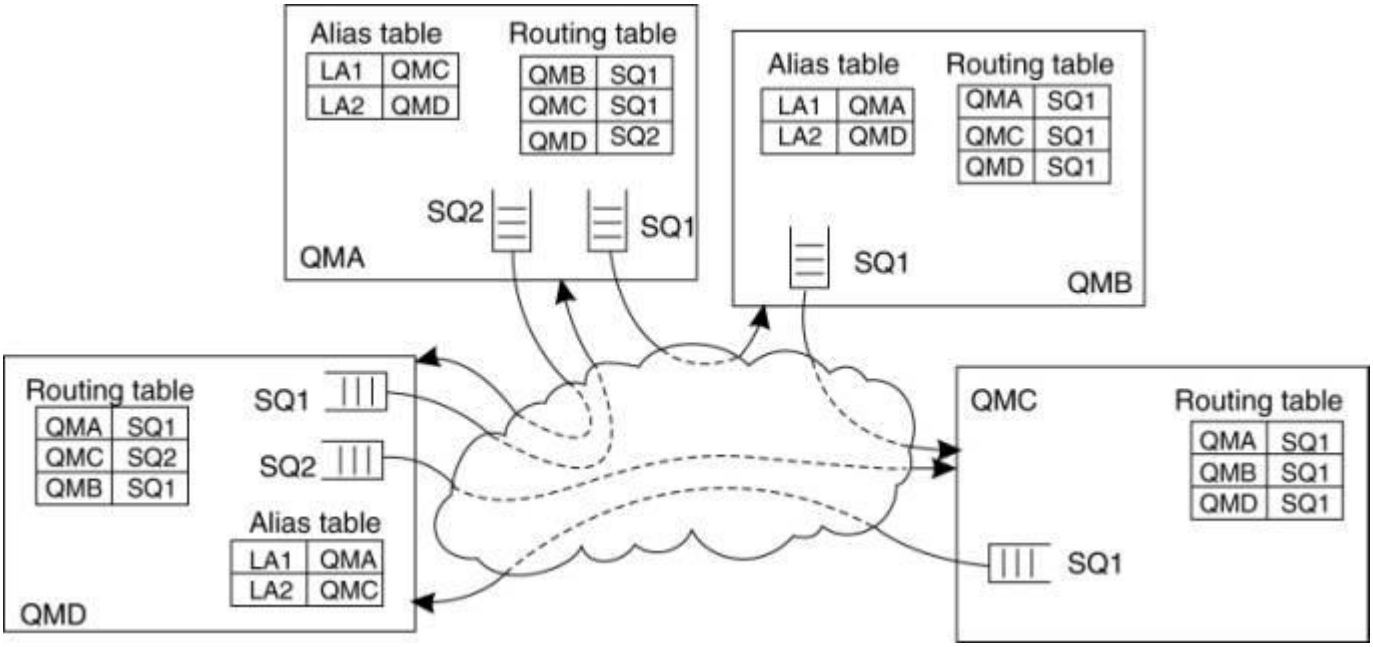
In most cases, routes are explicitly stored inside a queue manager in a routing table. An entry in a routing table is a pair (destQM, sendQ), where destQM is the name of the destination queue manager, and sendQ is the name of the local send queue to which a message for that queue manager should be appended. (A routing table entry is called an alias in MQ.)

It is possible that a message needs to be transferred across multiple queue managers before reaching its destination. Whenever such an intermediate queue manager receives the message, it simply extracts the name of the destination queue manager from the message header, and does a routing-table look-up to find the local send queue to which the message should be appended.

It is important to realize that each queue manager has a system wide unique name that is effectively used as an identifier for that queue manager. The problem with using these names is that replacing a queue manager, or changing its name, will affect all applications that send messages to it. Problems can be alleviated by using a local alias for queue manager names. An alias defined within a queue manager M1 is another name for a queue manager M2, but which is available only to applications interfacing to M1. An alias allows the use of the same (logical) name for a queue, even if the queue manager of that queue changes. Changing the name of a queue manager requires that we change its alias in all queue managers. However, applications can be left unaffected.

The principle of using routing tables and aliases is shown in Fig. 4-24. For example, an application linked to queue manager QMA can refer to a remote queue manager using the local alias LA1. The queue manager will first look up the actual destination in the alias table to find it is queue manager QMC. The route to QMC is found in the routing table, which states that messages for QMC should be appended to the outgoing queue SQ1, which is used to transfer messages to queue manager QMB. The latter will use its routing table to forward the message to QMC.

Figure 4-24. The general organization of an MQ queuing network using routing tables and aliases.



Following this approach of routing and aliasing leads to a programming interface that, fundamentally, is relatively simple, called the Message Queue Interface (MQI). The most important primitives of MQI are summarized in Fig. 4-25.

Figure 4-25. Primitives available in the message-queuing interface.

Primitive	Description
MQopen	Open a (possibly remote) queue
MQclose	Close a queue
MQput	Put a message into an opened queue
MQget	Get a message from a (local) queue

To put messages into a queue, an application calls the MQopen primitive, specifying a destination queue in a specific queue manager. The queue manager can be named using the locally-available alias. Whether the destination queue is actually remote or not is completely transparent to the application. MQopen should also be called if the application wants to get messages from its local queue. Only local queues can be opened for reading incoming messages. When an application is finished with accessing a queue, it should close it by calling MQclose.

Messages can be written to, or read from, a queue using MQput and MQget, respectively. In principle, messages are removed from a queue on a priority basis. Messages with the same priority are removed on a first-in, first-out basis, that is, the longest pending message is removed first. It is also possible to request for specific

messages. Finally, MQ provides facilities to signal applications when messages have arrived, thus avoiding that an application will continuously have to poll a message queue for incoming messages.

Managing Overlay Networks

From the description so far, it should be clear that an important part of managing MQ systems is connecting the various queue managers into a consistent overlay network. Moreover, this network needs to be maintained over time. For small networks, this maintenance will not require much more than average administrative work, but matters become complicated when message queuing is used to integrate and disintegrate large existing systems. A major issue with MQ is that overlay networks need to be manually administrated. This administration not only involves creating channels between queue managers, but also filling in the routing tables. Obviously, this can grow into a nightmare. Unfortunately, management support for MQ systems is advanced only in the sense that an administrator can set virtually every possible attribute, and tweak any thinkable configuration. However, the bottom line is that channels and routing tables need to be manually maintained.

At the heart of overlay management is the channel control function component, which logically sits between message channel agents. This component allows an operator to monitor exactly what is going on at two end points of a channel. In addition, it is used to create channels and routing tables, but also to manage the queue managers that host the message channel agents. In a way, this approach to overlay management strongly resembles the management of cluster servers where a single administration server is used. In the latter case, the server essentially offers only a remote shell to each machine in the cluster, along with a few collective operations to handle groups of machines. The good news about distributed-systems management is that it offers lots of opportunities if you are looking for an area to explore new solutions to serious problems.

Stream-Oriented Communication

Communication has concentrated on exchanging more-or-less independent and complete units of information. Examples include a request for invoking a procedure, the reply to such a request, and messages exchanged between applications as in message-queuing systems. The characteristic feature of this type of communication is that it does not matter at what particular point in time communication takes place. Although a system may perform too slow or too fast, timing has no effect on correctness.

There are also forms of communication in which timing plays a crucial role. Consider, for example, an audio stream built up as a sequence of 16-bit samples, each representing the amplitude of the sound wave as is done through Pulse Code Modulation (PCM). Also assume that the audio stream represents CD quality, meaning that the original sound wave has been sampled at a frequency of 44,100 Hz. To reproduce the original sound, it is

essential that the samples in the audio stream are played out in the order they appear in the stream, but also at intervals of exactly 1/44,100 sec. Playing out at a different rate will produce an incorrect version of the original sound.

Support for Continuous Media

Support for the exchange of time-dependent information is often formulated as support for continuous media. A medium refers to the means by which information is conveyed. These means include storage and transmission media, presentation media such as a monitor, and so on. An important type of medium is the way that information is represented. In other words, how is information encoded in a computer system? Different representations are used for different types of information. For example, text is generally encoded as ASCII or Unicode. Images can be represented in different formats such as GIF or JPEG. Audio streams can be encoded in a computer system by, for example, taking 16-bit samples using PCM.

In continuous (representation) media, the temporal relationships between different data items are fundamental to correctly interpreting what the data actually means. We already gave an example of reproducing a sound wave by playing out an audio stream. As another example, consider motion. Motion can be represented by a series of images in which successive images must be displayed at a uniform spacing T in time, typically 30–40 msec per image. Correct reproduction requires not only showing the stills in the correct order, but also at a constant frequency of 1/T images per second.

In contrast to continuous media, discrete (representation) media, is characterized by the fact that temporal relationships between data items are not fundamental to correctly interpreting the data. Typical examples of discrete media include representations of text and still images, but also object code or executable files.

Data Stream

To capture the exchange of time-dependent information, distributed systems generally provide support for data streams. A data stream is nothing but a sequence of data units. Data streams can be applied to discrete as well as continuous media. For example, UNIX pipes or TCP/IP connections are typical examples of (byte-oriented) discrete data streams. Playing an audio file typically requires setting up a continuous data stream between the file and the audio device.

Timing is crucial to continuous data streams. To capture timing aspects, a distinction is often made between different transmission modes. In asynchronous transmission mode the data items in a stream are transmitted one after the other, but there are no further timing constraints on when transmission of items should take place. This

is typically the case for discrete data streams. For example, a file can be transferred as a data stream, but it is mostly irrelevant exactly when the transfer of each item completes.

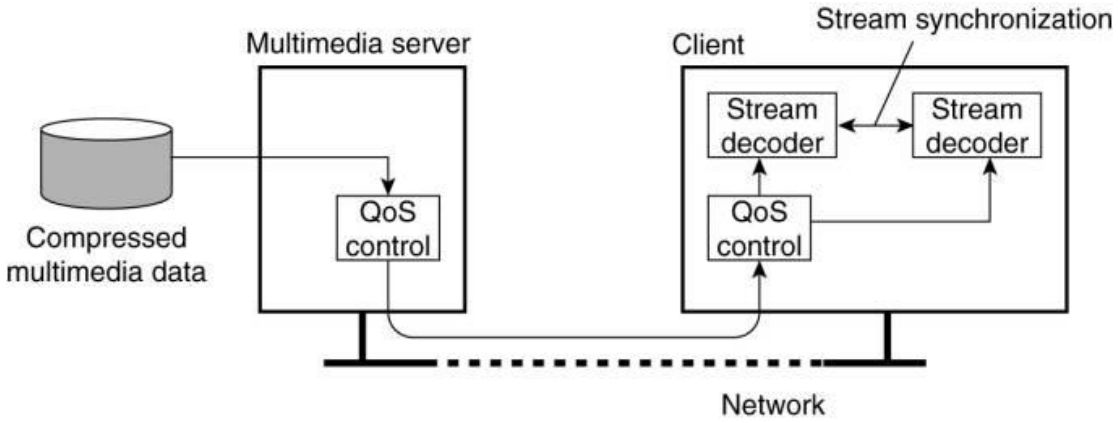
In synchronous transmission mode, there is a maximum end-to-end delay defined for each unit in a data stream. Whether a data unit is transferred much faster than the maximum tolerated delay is not important. For example, a sensor may sample temperature at a certain rate and pass it through a network to an operator. In that case, it may be important that the end-to-end propagation time through the network is guaranteed to be lower than the time interval between taking samples, but it cannot do any harm if samples are propagated much faster than necessary.

Finally, in isochronous transmission mode, it is necessary that data units are transferred on time. This means that data transfer is subject to a maximum and minimum end-to-end delay, also referred to as bounded (delay) jitter. Isochronous transmission mode is particularly interesting for distributed multimedia systems, as it plays a crucial role in representing audio and video.

Streams can be simple or complex. A simple stream consists of only a single sequence of data, whereas a complex stream consists of several related simple streams, called sub streams. The relation between the sub streams in a complex stream is often also time dependent. For example, stereo audio can be transmitted by means of a complex stream consisting of two sub streams, each used for a single audio channel. It is important, however, that those two sub streams are continuously synchronized. In other words, data units from each stream are to be communicated pairwise to ensure the effect of stereo. Another example of a complex stream is one for transmitting a movie. Such a stream could consist of a single video stream, along with two streams for transmitting the sound of the movie in stereo. A fourth stream might contain subtitles for the deaf, or a translation into a different language than the audio. Again, synchronization of the sub-streams is important. If synchronization fails, reproduction of the movie fails. We return to stream synchronization below.

From a distributed systems perspective, we can distinguish several elements that are needed for supporting streams. For simplicity, we concentrate on streaming stored data, as opposed to streaming live data. In the latter case, data is captured in real time and sent over the network to recipients. The main difference between the two is that streaming live data leaves less opportunities for tuning a stream. We can then sketch a general client-server architecture for supporting continuous multimedia streams as shown in Fig. 4-26.

Figure 4-26. A general architecture for streaming stored multimedia data over a network.



This general architecture reveals a number of important issues that need to be dealt with. In the first place, the multimedia data, notably video and to a lesser extent audio, will need to be compressed substantially in order to reduce the required storage and especially the network capacity. More important from the perspective of communication are controlling the quality of the transmission and synchronization issues.

STREAMS AND QUALITY OF SERVICE

Timing (and other nonfunctional) requirements are generally expressed as Quality of Service (QoS) requirements. These requirements describe what is needed from the underlying distributed system and network to ensure that, for example, the temporal relationships in a stream can be preserved. QoS for continuous data streams mainly concerns timeliness, volume, and reliability. In this section we take a closer look at QoS and its relation to setting up a stream.

Properties of QoS

1. The required bit rate at which data should be transported.
2. The maximum delay until a session has been set up (i.e., when an application can start sending data).
3. The maximum end-to-end delay (i.e., how long it will take until a data unit makes it to a recipient).
4. The maximum delay variance, or jitter.
5. The maximum round-trip delay.

It should be noted that many refinements can be made to these specificationsHowever, when dealing with stream-oriented communication that is based on the Internet protocol stack, we simply have to live with the fact that the basis of communication is formed by an extremely simple, best-effort datagram service: IP. When the going gets tough, as may easily be the case in the Internet, the specification of IP allows a protocol

implementation to drop packets whenever it sees fit. Many, if not all distributed systems that support stream-oriented communication, are currently built on top of the Internet protocol stack. So much for QoS specifications. (Actually, IP does provide some QoS support, but it is rarely implemented.)

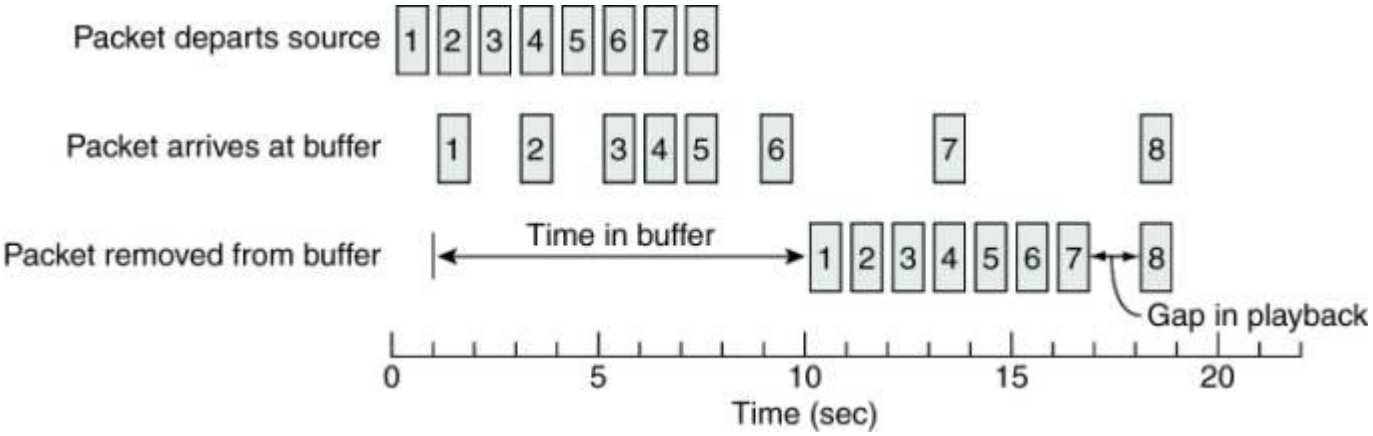
Enforcing QoS

Given that the underlying system offers only a best-effort delivery service, a distributed system can try to conceal as much as possible of the lack of quality of service. Fortunately, there are several mechanisms that it can deploy.

First, the situation is not really so bad as sketched so far. For example, the Internet provides a means for differentiating classes of data by means of its differentiated services. A sending host can essentially mark outgoing packets as belonging to one of several classes, including an expedited forwarding class that essentially specifies that a packet should be forwarded by the current router with absolute priority. In addition, there is also an assured forwarding class, by which traffic is divided into four subclasses, along with three ways to drop packets if the network gets congested. Assured forwarding therefore effectively defines a range of priorities that can be assigned to packets, and as such allows applications to differentiate time-sensitive packets from noncritical ones.

Besides these network-level solutions, a distributed system can also help in getting data across to receivers. Although there are generally not many tools available, one that is particularly useful is to use buffers to reduce jitter. The principle is simple, as shown in Fig. 4-27. Assuming that packets are delayed with a certain variance when transmitted over the network, the receiver simply stores them in a buffer for a maximum amount of time. This will allow the receiver to pass packets to the application at a regular rate, knowing that there will always be enough packets entering the buffer to be played back at that rate.

Figure 4-27. Using a buffer to reduce jitter.

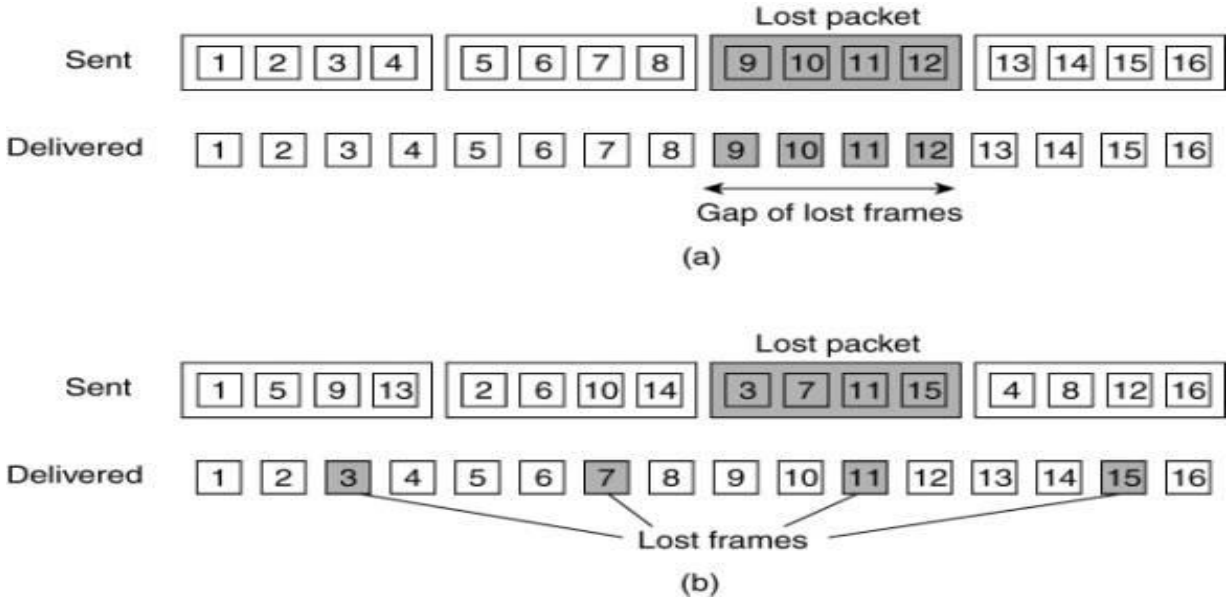


Of course, things may go wrong, as is illustrated by packet #8 in Fig. 4-27. The size of the receiver's buffer corresponds to 9 seconds of packets to pass to the application. Unfortunately, packet #8 took 11 seconds to reach the receiver, at which time the buffer will have been completely emptied. The result is a gap in the playback at the application. The only solution is to increase the buffer size. The obvious drawback is that the delay at which the receiving application can start playing back the data contained in the packets increases as well.

Other techniques can be used as well. Realizing that we are dealing with an underlying best-effort service also means that packets may be lost. To compensate for this loss in quality of service, we need to apply error correction techniques. Requesting the sender to retransmit a missing packet is generally out of the question, so that forward error correction (FEC) needs to be applied. A well-known technique is to encode the outgoing packets in such a way that any k out of n received packets is enough to reconstruct k correct packets.

One problem that may occur is that a single packet contains multiple audio and video frames. As a consequence, when a packet is lost, the receiver may actually perceive a large gap when playing out frames. This effect can be somewhat circumvented by interleaving frames, as shown in Fig. 4-28. In this way, when a packet is lost, the resulting gap in successive frames is distributed over time. Note, however, that this approach does require a larger receive buffer in comparison to non-interleaving, and thus imposes a higher start delay for the receiving application. For example, when considering Fig. 4-28(b), to play the first four frames, the receiver will need to have four packets delivered, instead of only one packet in comparison to non-interleaved transmission.

Figure 4-28. The effect of packet loss in (a) non-interleaved transmission and (b) interleaved transmission.



Stream Synchronization

An important issue in multimedia systems is that different streams, possibly in the form of a complex stream, are mutually synchronized. Synchronization of streams deals with maintaining temporal relations between streams. Two types of synchronization occur.

The simplest form of synchronization is that between a discrete data stream and a continuous data stream. Consider, for example, a slide show on the Web that has been enhanced with audio. Each slide is transferred from the server to the client in the form of a discrete data stream. At the same time, the client should play out a specific (part of an) audio stream that matches the current slide that is also fetched from the server. In this case, the audio stream is to be synchronized with the presentation of slides.

A more demanding type of synchronization is that between continuous data streams. A daily example is playing a movie in which the video stream needs to be synchronized with the audio, commonly referred to as lip synchronization. Another example of synchronization is playing a stereo audio stream consisting of two sub-streams, one for each channel. Proper play out requires that the two sub-streams are tightly synchronized: a difference of more than 20 μsec can distort the stereo effect.

Synchronization takes place at the level of the data units of which a stream is made up. In other words, we can synchronize two streams only between data units. The choice of what exactly a data unit is depends very much on the level of abstraction at which a data stream is viewed. To make things concrete, consider again a CD-quality (single-channel) audio stream. At the finest granularity, such a stream appears as a sequence of 16-bit samples. With a sampling frequency of 44,100 Hz, synchronization with other audio streams could, in theory, take place approximately every 23 μsec. For high-quality stereo effects, it turns out that synchronization at this level is indeed necessary.

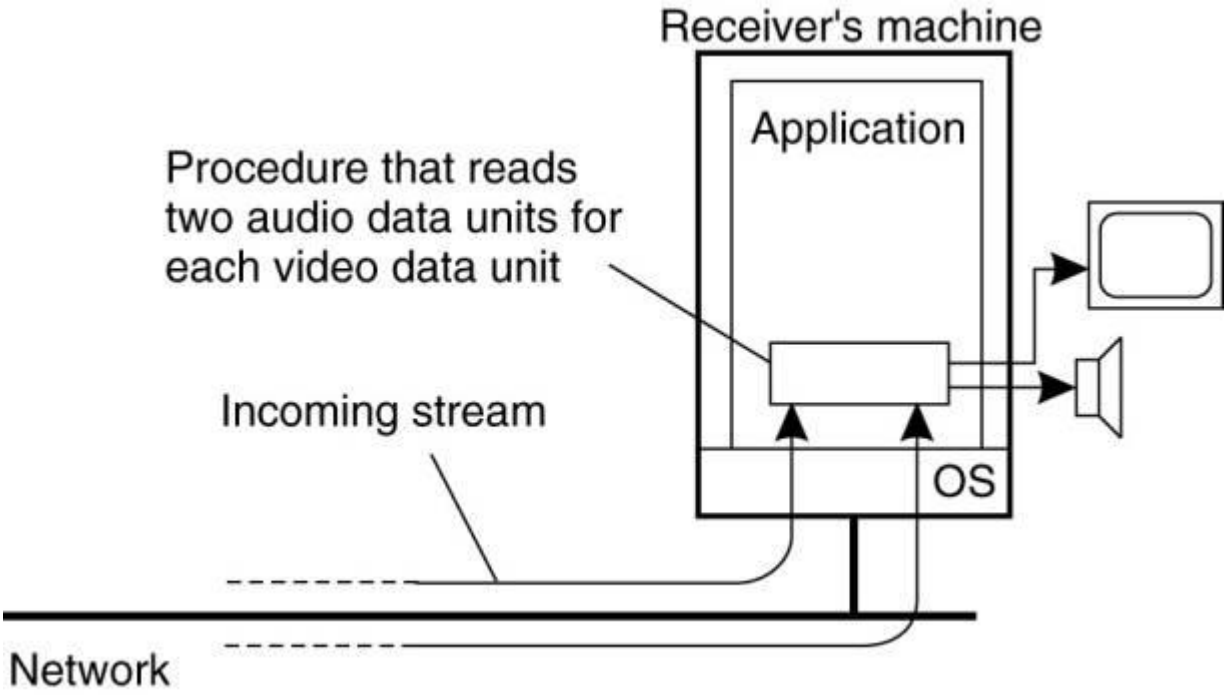
However, when we consider synchronization between an audio stream and a video stream for lip synchronization, a much coarser granularity can be taken. As we explained, video frames need to be displayed at a rate of 25 Hz or more. Taking the widely-used NTSC standard of 29.97 Hz, we could group audio samples into logical units that last as long as a video frame is displayed (33 msec). With an audio sampling frequency of 44,100 Hz, an audio data unit can thus be as large as 1470 samples, or 11,760 bytes (assuming each sample is 16 bits). In practice, larger units lasting 40 or even 80 msec can be tolerated (Steinmetz, 1996).

Synchronization Mechanisms

Let us now see how synchronization is actually done. Two issues need to be distinguished: (1) the basic mechanisms for synchronizing two streams, and (2) the distribution of those mechanisms in a networked environment.

Synchronization mechanisms can be viewed at several different levels of abstraction. At the lowest level, synchronization is done explicitly by operating on the data units of simple streams. This principle is shown in Fig. 4-29. In essence, there is a process that simply executes read and write operations on several simple streams, ensuring that those operations adhere to specific timing and synchronization constraints.

Figure 4-29. The principle of explicit synchronization on the level data units.

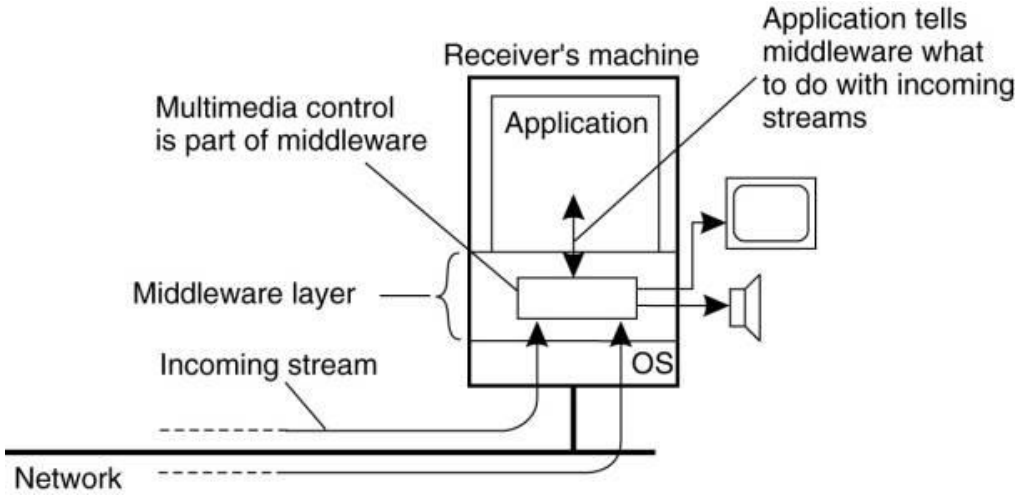


For example, consider a movie that is presented as two input streams. The video stream contains uncompressed low-quality images of 320x240 pixels, each encoded by a single byte, leading to video data units of 76,800 bytes each. Assume that images are to be displayed at 30 Hz, or one image every 33 msec. The audio stream is assumed to contain audio samples grouped into units of 11760 bytes, each corresponding to 33 ms of audio, as explained above. If the input process can handle 2.5 MB/sec, we can achieve lip synchronization by simply alternating between reading an image and reading a block of audio samples every 33 ms.

The drawback of this approach is that the application is made completely responsible for implementing synchronization while it has only low-level facilities available. A better approach is to offer an application an interface that allows it to more easily control streams and devices. Returning to our example, assume that the video display has a control interface that allows it to specify the rate at which images should be displayed. In addition, the interface offers the facility to register a user-defined handler that is called each time k new images have arrived. An analogous interface is offered by the audio device. With these control interfaces, an application developer can write a simple monitor program consisting of two handlers, one for each stream, that jointly check if the video and audio stream are sufficiently synchronized, and if necessary, adjust the rate at which video or audio units are presented.

This last example is illustrated in Fig. 4-30, and is typical for many multimedia middleware systems. In effect, multimedia middleware offers a collection of interfaces for controlling audio and video streams, including interfaces for controlling devices such as monitors, cameras, microphones, etc. Each device and stream has its own high-level interfaces, including interfaces for notifying an application when some event occurred. The latter are subsequently used to write handlers for synchronizing streams.

Figure 4-30. The principle of synchronization as supported by high-level interfaces.



The distribution of synchronization mechanisms is another issue that needs to be looked at. First, the receiving side of a complex stream consisting of sub streams that require synchronization, needs to know exactly what to do. In other words, it must have a complete synchronization specification locally available. Common practice is to provide this information implicitly by multiplexing the different streams into a single stream containing all data units, including those for synchronization.

This latter approach to synchronization is followed for MPEG streams. The MPEG (Motion Picture Experts Group) standards form a collection of algorithms for compressing video and audio. Several MPEG standards exist. MPEG-2, for example, was originally designed for compressing broadcast quality video into 4 to 6 Mbps. In MPEG-2, an unlimited number of continuous and discrete streams can be merged into a single stream. Each input stream is first turned into a stream of packets that carry a timestamp based on a 90-kHz system clock. These streams are subsequently multiplexed into a program stream then consisting of variable-length packets, but which have in common that they all have the same time base. The receiving side DE multiplexes the stream, again using the timestamps of each packet as the basic mechanism for inter-stream synchronization.

Another important issue is whether synchronization should take place at the sending or the receiving side. If the sender handles synchronization, it may be possible to merge streams into a single stream with a different type of data unit. Consider again a stereo audio stream consisting of two sub streams, one for each channel. One

possibility is to transfer each stream independently to the receiver and let the latter synchronize the samples pairwise. Obviously, as each sub stream may be subject to different delays, synchronization can be extremely difficult. A better approach is to merge the two sub streams at the sender. The resulting stream consists of data units consisting of pairs of samples, one for each channel. The receiver now merely has to read in a data unit, and split it into a left and right sample. Delays for both channels are now identical.