Unit 4: Communication

Contents

Inter-process communication is at the heart of all distributed systems. Communication in distributed systems has traditionally always been based on low-level message passing as offered by the underlying network. Modern distributed systems often consist of thousands or even millions of processes scattered across a network with unreliable communication such as the Internet. Unless the primitive communication facilities of computer networks are replaced by something else, development of large-scale distributed applications is extremely difficult.

4.1 Foundations

Layered Protocols

Due to the absence of shared memory, all communication in distributed systems is based on sending and receiving (low level) messages. When process P wants to communicate with process Q, it first builds a message in its own address space. Then it executes a system call that causes the operating system to send the message over the network to Q. Although this basic idea sounds simple enough, in order to prevent confusion, P and Q have to agree on the meaning of the bits being sent.

The OSI reference model

To make it easier to deal with the numerous levels and issues involved in communication, the International Standards Organization (ISO) developed a reference model that clearly identifies the various levels involved, gives them standard names, and points out which level should do which job. This model is called the Open Systems Interconnection Reference Model abbreviated as ISO OSI or sometimes just the OSI model.

The OSI model is designed to allow open systems to communicate. An open system is one that is prepared to communicate with any other open system by using standard rules that govern the format, contents, and meaning of the messages sent and received. These rules are formalized in what are called communication protocols. To allow a group of computers to communicate over a network, they must all agree on the protocols to be used. A protocol is said to provide a communication service. There are two types of such services.

- In the case of **a connection-oriented service**, before exchanging data the sender and receiver first explicitly establish a connection, and possibly negotiate specific parameters of the protocol they will use. When they are done, they release (terminate) the connection. The telephone is a typical connection-oriented communication service.
- With **connectionless services**, no setup in advance is needed. The sender just transmits the first message when it is ready. Dropping a letter in a mailbox is an example of making use of

connectionless communication service. With computers, both connection-oriented and connectionless communication are common.

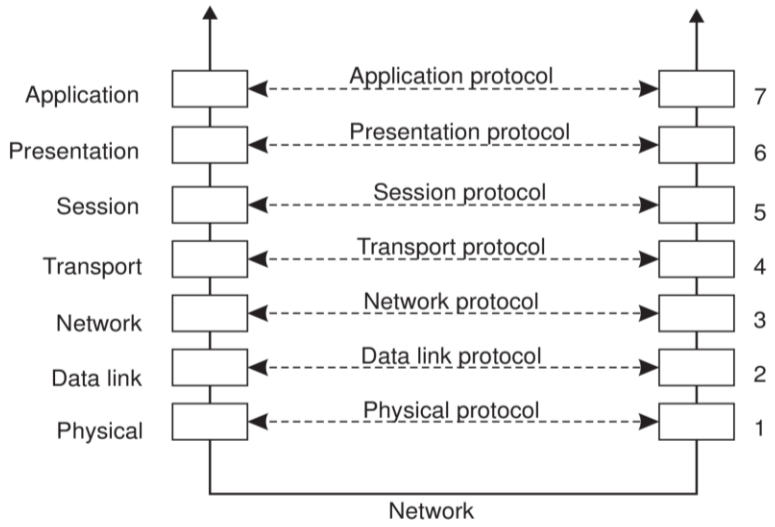


Figure: Layers, interfaces, and protocols in the OSI reference model

In the OSI model, communication is divided into seven levels or layers. Each layer offers one or more specific communication services to the layer above it. In this way, the problem of getting a message from A to B can be divided into manageable pieces, each of which can be solved independently of the others. Each layer provides an interface to the one above it. The interface consists of a set of operations that together define the service the layer is prepared to offer. The seven OSI layers are:

Physical layer deals with standardizing how two computers are connected and how 0s and 1s are represented.

Data link layer provides the means to detect and possibly correct transmission errors, as well as protocols to keep a sender and receiver in the same pace.

Network layer contains the protocols for routing a message through a computer network, as well as protocols for handling congestion.

Transport layer mainly contains protocols for directly supporting applications, such as those that establish reliable communication, or support real-time streaming of data.

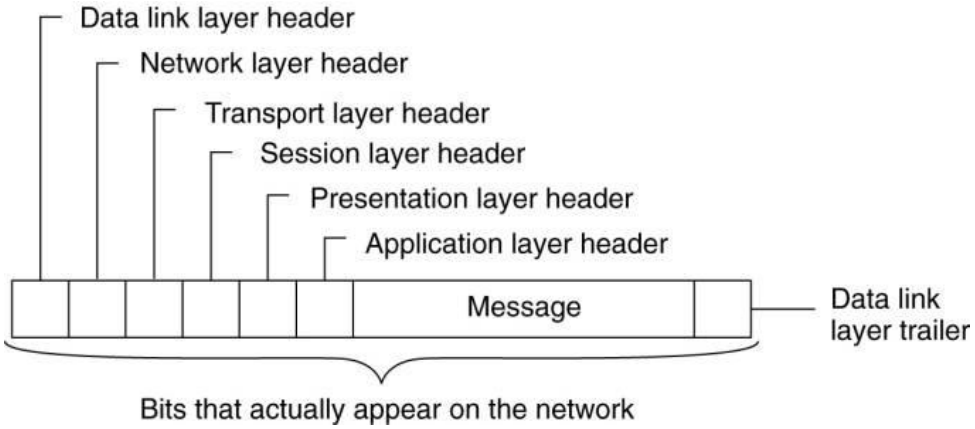
Session layer provides support for sessions between applications.

Presentation layer prescribes how data is represented in a way that is independent of the hosts on which communicating applications are running.

Application layer essentially, everything else: e-mail protocols, Web access protocols, file-transfer protocols, and so on.

When process P wants to communicate with some remote process Q, it builds a message and passes that message to the application layer as offered to it by means of an interface. This interface will typically appear in the form of a library procedure. The application layer software then adds a header to the front of the message and passes the resulting message across the layer 6/7 interface to the presentation layer. The presentation layer, in turn, adds its own header and passes the result down to the session layer, and

so on. Some layers add not only a header to the front, but also a trailer to the end. When it hits the bottom, the physical layer actually transmits the message by putting it onto the physical transmission medium.



When the message arrives at the remote machine hosting Q, it is passed upward, with each layer stripping off and examining its own header. Finally, the message arrives at the receiver, process Q, which may reply to it using the reverse path. The information in the layer-n header is used for the layer-n protocol.

Middleware protocols

Middleware is an application that logically lives (mostly) in the OSI application layer, but which contains many general-purpose protocols that warrant their own layers, independent of other, more specific applications. Let us briefly look at some examples.

The Domain Name System (DNS) is a distributed service that is used to look up a network address associated with a name, such as the address of a so-called domain name like www.distributed-systems.net. In terms of the OSI reference model, DNS is an application and therefore is logically placed in the application layer. However, it should be quite obvious that DNS is offering a general-purpose, application-independent service. Arguably, it forms part of the middleware.

As another example, there are various ways to establish authentication, that is, provide proof of a claimed identity. **Authentication protocols** are not closely tied to any specific application, but instead, can be integrated into a middleware system as a general service. Likewise, authorization protocols by which authenticated users and processes are granted access only to those resources for which they have authorization, tend to have a general, application-independent nature. Being labeled as applications in the OSI reference model, these are clear examples that belong in the middleware.

Distributed commit protocols establish that in a group of processes, possibly spread out across a number of machines, either all processes carry out a particular operation, or that the operation is not carried out at all. This phenomenon is also referred to as atomicity and is widely applied in transactions. As it turns out, commit protocols can present an interface independently of specific applications, thus providing a general-purpose transaction service. In such a form, they typically belong to the middleware and not to the OSI application layer.

As a last example, consider a **distributed locking protocol** by which a resource can be protected against simultaneous access by a collection of processes that are distributed across multiple machines. It is not hard to imagine that such protocols can be designed in an application-independent fashion, and accessible through a relatively simple, again application-independent interface. As such, they generally belong in the middleware.

These protocol examples are not directly tied to communication, yet there are also many middleware communication protocols. For example, with a so-called **remote procedure call**, a process is offered a facility to locally call a procedure that is effectively implemented on a remote machine. This communication service belongs to one of the oldest types of middleware services and is used for realizing access transparency. In a similar vein, there are **high-level communication services** for setting and synchronizing streams for transferring real-time data, such as needed for multimedia applications. As a last example, some middleware systems offer **reliable multicast services** that scale to thousands of receivers spread across a wide-area network.

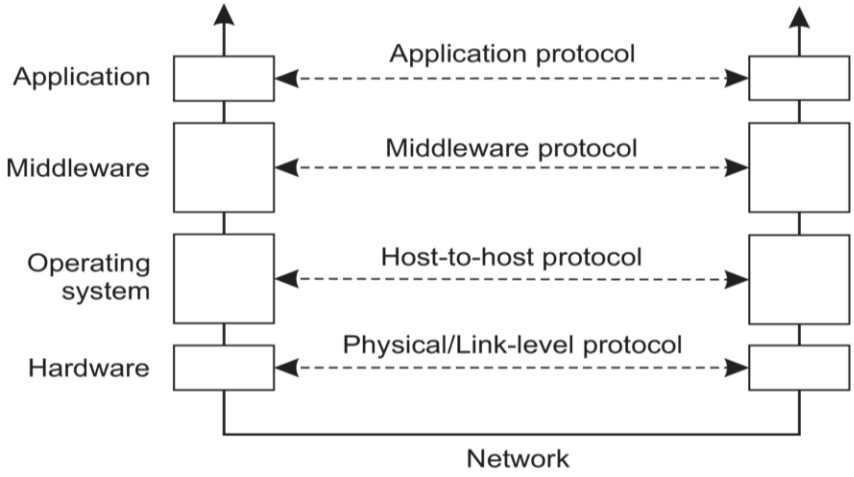


Figure: An adapted reference model for networked communication

Taking this approach to layering leads to the adapted and simplified reference model for communication, as shown in Figure above. Compared to the OSI model,

- The session and presentation layer have been replaced by a single middleware layer that contains application-independent protocols. These protocols do not belong in the lower layers we just discussed.
- Network and transport services have been grouped into communication services as normally offered by an operating system, which, in turn, manages the specific lowest-level hardware used to establish communication.

Types of Communication

To understand the various alternatives in communication that middleware can offer to applications, we view the middleware as an additional service in client-server computing, as shown in Figure below. Consider, for example an electronic mail system. In principle, the core of the mail delivery system can be seen as a middleware communication service. Each host runs a user agent allowing users to compose, send, and receive e-mail. A sending user agent passes such mail to the mail delivery system, expecting it, in turn, to eventually deliver the mail to the intended recipient. Likewise, the user agent at the receiver's side connects to the mail delivery system to see whether any mail has come in. If so, the messages are transferred to the user agent so that they can be displayed and read by the user.

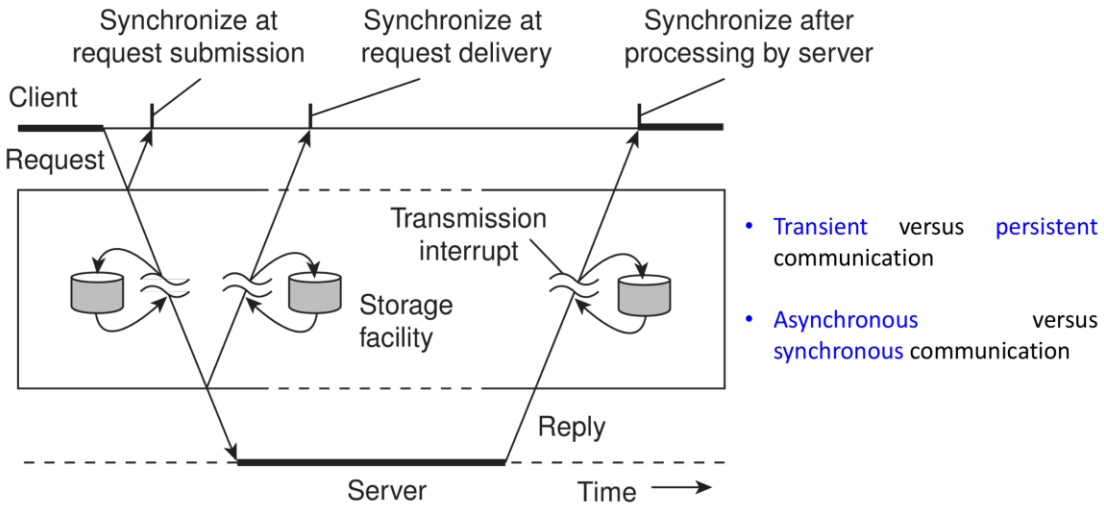


Figure: Viewing middleware as intermediate (distributed) service in application-level communication

An electronic mail system is a typical example in which communication is persistent. With **persistent communication**, a message that has been submitted for transmission is stored by the communication middleware as long as it takes to deliver it to the receiver. In this case, the middleware will store the message at one or several of the storage facilities shown in Figure above. As a consequence, it is not necessary for the sending application to continue execution after submitting the message. Likewise, the receiving application need not be executing when the message is submitted.

In contrast, with **transient communication**, a message is stored by the communication system only as long as the sending and receiving application are executing. More precisely, in terms of Figure, if the middleware cannot deliver a message due to a transmission interrupt, or because the recipient is currently not active, it will simply be discarded. Typically, all transport-level communication services offer only transient communication. In this case, the communication system consists of traditional store-and-forward routers. If a router cannot deliver a message to the next one or the destination host, it will simply drop the message.

Besides being persistent or transient, communication can also be asynchronous or synchronous.

The characteristic feature of **asynchronous communication** is that a sender continues immediately after it has submitted its message for transmission. This means that the message is (temporarily) stored immediately by the middleware upon submission.

With **synchronous communication**, the sender is blocked until its request is known to be accepted. There are essentially three points where synchronization can take place.

- First, the sender may be blocked until the middleware notifies that it will take over transmission of the request.
- Second, the sender may synchronize until its request has been delivered to the intended recipient.
- Third, synchronization may take place by letting the sender wait until its request has been fully processed, that is, up to the time that the recipient returns a response.

Various combinations of persistence and synchronization occur in practice. Popular ones are persistence in combination with synchronization at request submission, which is a common scheme for many message-queuing systems. Likewise, transient communication with synchronization after the request has been fully processed is also widely used. This scheme corresponds with remote procedure calls.

4.2 Remote Procedure Call

Many distributed systems have been based on explicit message exchange between processes. However, the operations send and receive do not conceal communication at all, which is important to achieve access transparency in distributed systems. This problem has long been known, but little was done about it until researchers in the 1980s introduced a completely different way of handling communication. Although the idea is refreshingly simple (once someone has thought of it), the implications are often subtle.

In a nutshell, the proposal was to allow programs to call procedures located on other machines. When a process on machine A calls a procedure on machine B, the calling process on A is suspended, and execution of the called procedure takes place on B. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message passing at all is visible to the programmer. This method is known as **Remote Procedure Call**, or often just **RPC**.

While the basic idea sounds simple and elegant, subtle problems exist. As the calling and called procedures run on different machines, they execute in different address spaces, which causes complications. Parameters and results also have to be passed, which can be complicated, especially if the machines are not identical. Finally, either or both machines can crash and each of the possible failures causes different problems. Still, most of these can be dealt with, and RPC is a widely-used technique that underlies many distributed systems.

Basic RPC operation

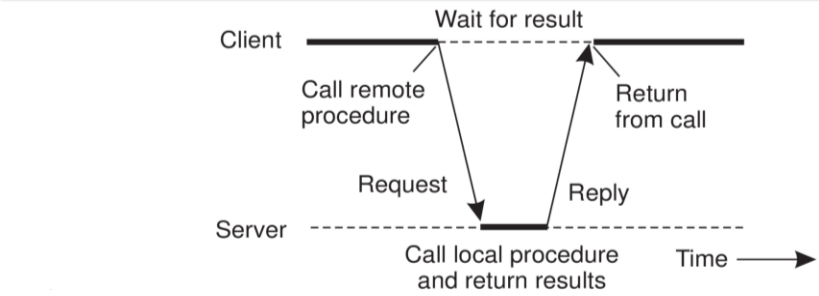
The idea behind RPC is to make a remote procedure call look as much as possible like a local one. In other words, we want RPC to be transparent—the calling procedure should not be aware that the called procedure is executing on a different machine or vice versa.

Suppose that a program has access to a database that allows it to append data to a stored list, after which it returns a reference to the modified list. The operation is made available to a program by means of a routine append: newlist = append(data, dbList)

In a traditional (single-processor) system, append is extracted from a library by the linker and inserted into the object program. In principle, it can be a short procedure, which could be implemented by a few file operations for accessing the database.

Even though append eventually does only a few basic file operations, it is called in the usual way, by pushing its parameters onto the stack. The programmer does not know the implementation details of append, and this is, of course, how it is supposed to be.

RPC achieves its transparency in an analogous way. When append is actually a remote procedure, a different version of append, called a **client stub**, is offered to the calling client. Like the original one, it, too, is called using a normal calling sequence. However, unlike the original one, it does not perform an append operation. Instead, it packs the parameters into a message and requests that message to be sent to the server as illustrated in Figure below. Following the call to send, the client stub calls receive, blocking itself until the reply comes back.



When the message arrives at the server, the server's operating system passes it to a **server stub**. A server stub is the server-side equivalent of a client stub: it is a piece of code that transforms requests coming in over the network into local procedure calls. Typically the server stub will have called receive and be blocked waiting for incoming messages. The server stub unpacks the parameters from the message and then calls the server procedure in the usual way. From the server's point of view, it is as though it is being called directly by the client—the parameters and return address are all on the stack where they belong and nothing seems unusual. The server performs its work and then returns the result to the caller (in this case the server stub) in the usual way.

When the server stub gets control back after the call has completed, it packs the result in a message and calls send to return it to the client. After that, the server stub usually does a call to receive again, to wait for the next incoming request.

When the result message arrives at the client's machine, the operating system passes it through the receive operation, which had been called previously, to the client stub, and the client process is subsequently unblocked. The client stub inspects the message, unpacks the result, copies it to its caller, and returns in the usual way. When the caller gets control following the call to append, all it knows is that it appended some data to a list. It has no idea that the work was done remotely at another machine.

Remote services are accessed by making ordinary (i.e., local) procedure calls, not by calling send and receive. All the details of the message passing are hidden away in the two library procedures, just as the details of actually making system calls are hidden away in traditional libraries.

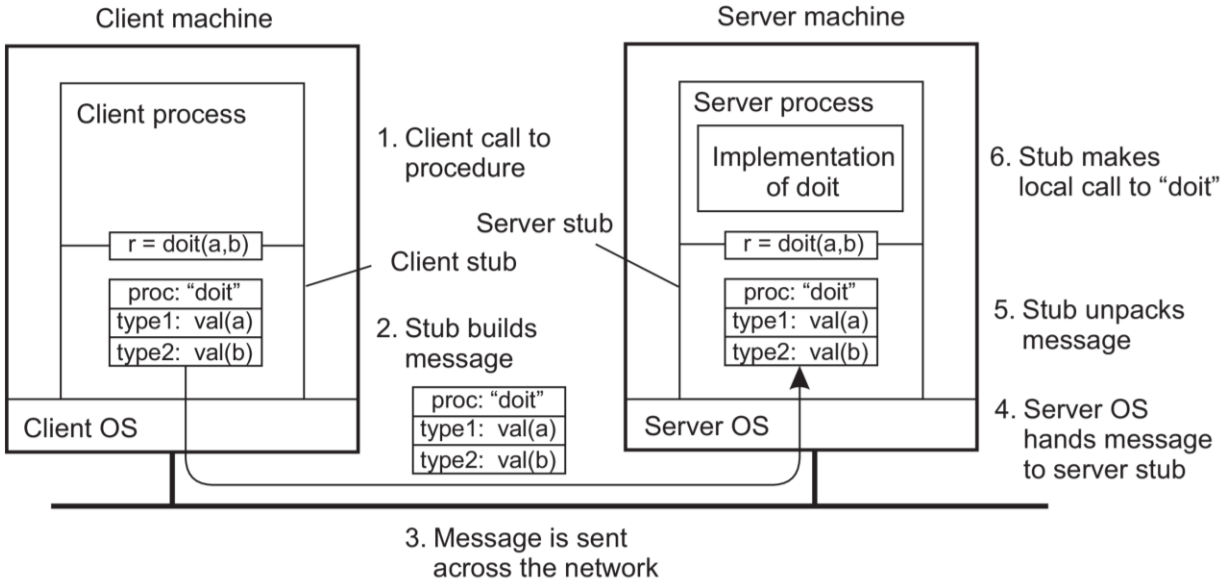


Figure: The steps involved in calling a remote procedure doit(a,b).

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client's OS sends the message to the remote OS.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameter(s) and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs the result in a message and calls its local OS.
8. The server's OS sends the message to the client's OS.
9. The client's OS gives the message to the client stub.
10. The stub unpacks the result and returns it to the client.

Parameter passing

The function of the client stub is to take its parameters, pack them into a message, and send them to the server stub. While this sounds straightforward, it is not quite as simple as it at first appears. Packing parameters into a message is called **parameter marshaling**.

RPC-based application support

Hiding a remote procedure call requires that the caller and the callee agree on the format of the messages they exchange and that they follow the same steps when it comes to, for example, passing complex data structures. In other words, both sides in an RPC should follow the same protocol or the RPC will not work correctly. There are at least two ways in which RPC-based application development can be supported. The first one is to let a developer specify exactly what needs to be called remotely, from which complete client-side and server-side stubs can be generated. A second approach is to embed remote procedure calling as part of a programming-language environment.

Language-based support

The approach described up until now is largely independent of a specific programming language. As an alternative, we can also embed remote procedure calling into a language itself. The main benefit is that application development often becomes much simpler. Also, reaching a high degree of access transparency is often simpler as many issues related to parameter passing can be circumvented altogether.

A well-known example in which remote procedure calling is fully embedded is Java, where an RPC is referred to as a remote method invocation (RMI). In essence, a client being executed by its own (Java) virtual machine can invoke a method of an object managed by another virtual machine. By simply reading an application's source code, it may be hard or even impossible to see whether a method invocation is to a local or to a remote object.

Advantages of RPC

1. It supports process and thread oriented models.
2. The internal message passing mechanism of RPC is hidden from the user.
3. The effort to re-write and re-develop the code is minimum is RPC.
4. It can be used in distributed environment as well as the local environment.
5. Multiple protocol layers are omitted by RPC to improve performance.

Disadvantages of RPC

1. It is a concept that can be implemented in different ways. It is not a standard.
2. There is no flexibility in RPC for hardware architecture. It is only interaction based.
3. It increased cost.

Variations on RPC

As in conventional procedure calls, when a client calls a remote procedure, the client will block until a reply is returned. This strict request-reply behavior is unnecessary when there is no result to return, or may hinder efficiency when multiple RPCs need to be performed. In the following we look at two variations on the RPC scheme:

1. Asynchronous RPC
2. Multicast RPC

Asynchronous RPC

With asynchronous RPCs, the server, in principle, immediately sends a reply back to the client the moment the RPC request is received, after which it locally calls the requested procedure. The reply acts as an acknowledgment to the client that the server is going to process the RPC. The client will continue without further blocking as soon as it has received the server's acknowledgment. Figure shows how client and server interact in the case of asynchronous RPCs.

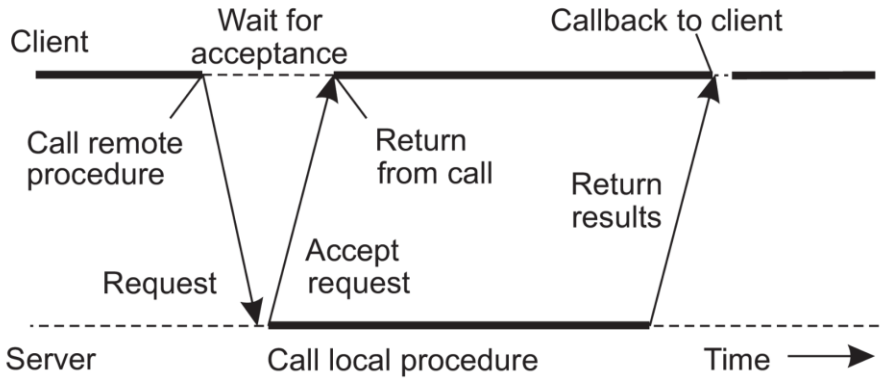


Figure: A client and server interacting through asynchronous RPC

Asynchronous RPCs can also be useful when a reply will be returned but the client is not prepared to wait for it and do nothing in the meantime. A typical case is when a client needs to contact several servers independently. In that case, it can send the call requests one after the other, effectively establishing that the servers operate more or less in parallel. After all call requests have been sent, the client can start waiting for the various results to be returned. In cases such as these, it makes sense to organize the communication between the client and server through an asynchronous RPC combined with a callback. In this scheme, also referred to as **deferred synchronous RPC**, the client first calls the server, waits for the acceptance, and continues. When the results become available, the server sends a response message that leads to a callback at the client's side.

Multicast RPC

Asynchronous and deferred synchronous RPCs facilitate another alternative to remote procedure calls, namely executing multiple RPCs at the same time. Adopting the one-way RPCs (i.e., when a server does not tell the client it has accepted its call request but immediately starts processing it), a multicast RPC

boils down to sending an RPC request to a group of servers. In this example, the client sends a request to two servers, who subsequently process that request independently and in parallel. When done, the result is returned to the client where a callback takes place.

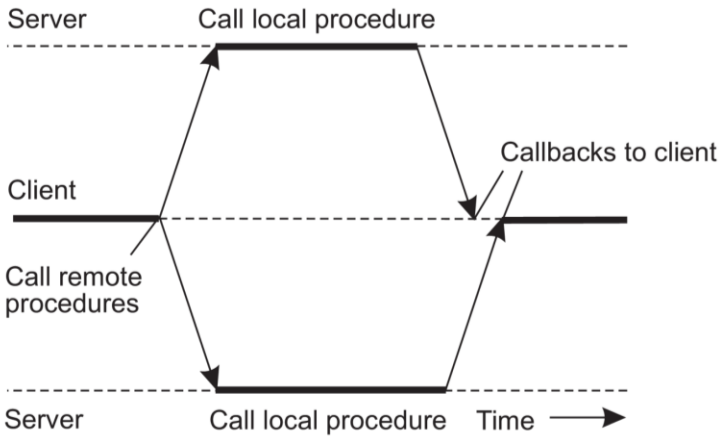


Figure: The principle of multicast RPC

There are several issues that we need to consider.

- First, as before, the client application may be unaware of the fact that an RPC is actually being forwarded to more than one server. For example, to increase fault tolerance, we may decide to have all operations executed by a backup server who can take over when the main server fails. That a server has been replicated can be completely hidden from a client application by an appropriate stub. Yet even the stub need not be aware that the server is replicated, for example, because we are using a transport-level multicast address.
- Second, we need to consider what to do with the responses. In particular, will the client proceed after all responses have been received, or wait just for one? It all depends. When the server has been replicated for fault tolerance, we may decide to wait for just the first response, or perhaps until a majority of the servers returns the same result. On the other hand, if the servers have been replicated to do the same work but on different parts of the input, their results may need to be merged before the client can continue. Again, such matters can be hidden in the client-side stub, yet the application developer will, at the very least, have to specify the purpose of the multicast RPC.

4.3 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, may need to be replaced by something else such as messaging.

Simple transient messaging with sockets

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, we first discuss messaging through transport-level sockets.

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 port that is used by the local operating system for a specific transport protocol. In the following text, we concentrate on the socket operations for TCP, which are shown in table.

Table: The socket operations for TCP/IP

Operation	**Description**
socket	Create a new communication end point
bind	Attach a local address to a socket
listen	Tell operating system what the maximum number of pending connection requests should be
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 operations, normally in the order given.

When calling the socket operation, 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 for sending and receiving messages for the specified protocol.

The bind operation 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. In the case of connection-oriented communication, the address is used to receive incoming connection requests.

The listen operation is called only in the case of connection-oriented communication. It is a nonblocking call that allows the local operating system to reserve enough buffers for a specified maximum number of pending connection requests 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 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 operation, 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 operation 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 operations. Finally, closing a connection is symmetric when using sockets, and is established by having both the client and server call the close operation. Although there are many exceptions to the rule, the general pattern followed by a client and server for connection-oriented communication using sockets is as shown in Figure.

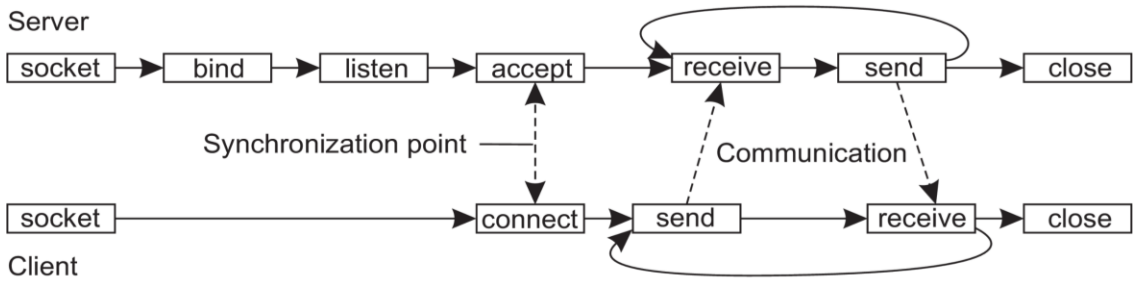


Figure Connection-oriented communication pattern using sockets.

Advanced transient messaging

The standard socket-based approach toward transient messaging is very basic and as such, rather brittle: a mistake is easily made. Furthermore, sockets essentially support only TCP or UDP, meaning that any extra facility for messaging needs to be implemented separately by an application programmer. In practice, we do often need more advanced approaches for message-oriented communication to make network programming easier, to expand beyond the functionality offered by existing networking protocols, to make better use of local resources, and so on.

The Message-Passing Interface (MPI)

With the arrival of high-performance multicomputers, developers have been looking for message-oriented operations that would allow them to easily write highly efficient applications. This means that the operations 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 operations. 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 multicomputers were shipped with proprietary communication libraries. These libraries offered a wealth of high-level and generally efficient communication operations. 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 lead 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 operations to support transient communication, of which the most intuitive ones are summarized in table.

Table: Some of the most intuitive message-passing operations of MPI

Operation	Description
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 transmission 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.

Message-oriented persistent communication

Message-queuing systems, or just Message-Oriented Middleware (MOM) is an important class of message-oriented middleware service. 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 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.

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.

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 Table.

Table: Basic interface to a queue in a message-queuing system

Operation	Description
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.

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

Queues are managed by queue managers. A queue manager is either a separate process, or is implemented by means of a library that is linked with an application. Secondly, as a rule of thumb, an application can put messages only into a local queue. Likewise, getting a message is possible by extracting it from a local queue only. As a consequence, if a queue manager QMA handling the queues for an application A runs as a separate process, both processes QMA and A will generally be placed on the same machine, or at worst on the same LAN. Also note that if all queue managers are linked into their respective applications, we can no longer speak of a persistent asynchronous messaging system.

If applications can put messages only into local queues, then clearly each message will have to carry information concerning its destination. It is the queue manager's task to make sure that a message reaches its destination. This brings us to a number of issues.

In the first place, we need to consider how the destination queue is addressed. Obviously, to enhance location transparency, it is preferable that queues have logical, location-independent names. Assuming that a queue manager is implemented as a separate process, using logical names implies that each name should be associated with a contact address, such as a (host,port)- pair, and that the name-to-address mapping is readily available to a queue manager, as shown in Figure. In practice, a contact address carries more information, notably the protocol to be used, such as TCP or UDP.

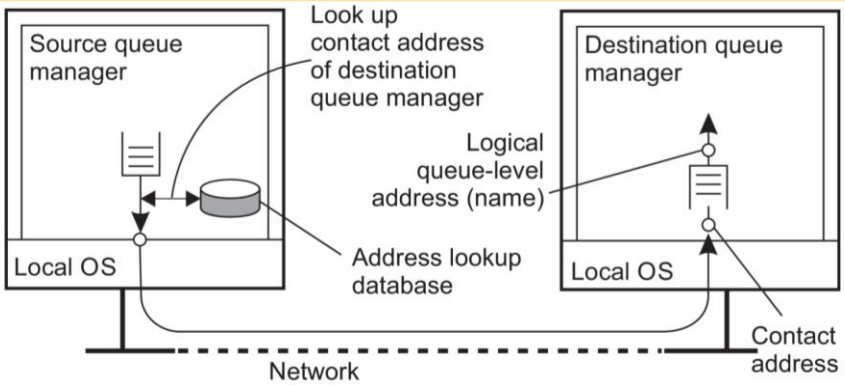


Figure: The relation between queue-level naming and network-level addressing

A second issue that we need to consider is how the name-to-address mapping is actually made available to a queue manager. A common approach is to simply implement the mapping as a lookup table and copy

that table to all managers. Obviously, this leads to a maintenance problem, for every time that a new queue is added or named, many, if not all tables need to be updated.

Now the third issue, related to the problems of efficiently maintaining name-to-address mappings. We have implicitly assumed that if a destination queue at manager QMB is known to queue manager QMA, then QMA can directly contact QMB to transfer messages. In effect, this means that (the contact address of) each queue manager should be known to all others. Obviously, when dealing with very large message-queuing systems, we will have a scalability problem. In practice, there are often special queue managers that operate as routers: 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.

Message brokers

An important application area of message-queuing systems is integrating existing and new applications into a single, coherent distributed information system. If we assume that communication with an application takes place through messages, then 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, but also that its messages adhere to the same semantics as those expected by the receiver. Sender and receiver essentially need to speak the same language, that is, adhere to the same messaging protocol.

The problem with this approach is that each time an application A is added to the system having its own messaging protocol, then for each other application B that is to communicate with A we will need to provide the means for converting their respective messages. In a system with N applications, we will thus need N × N messaging protocol converters.

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 Figure. In other words, a message broker is generally not considered to be an integral part of the queuing system.

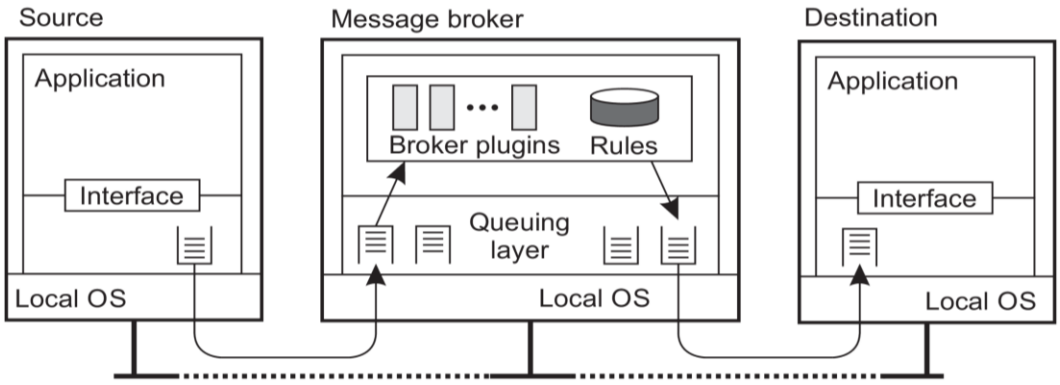


Figure: The general organization of Message broker in a message-queuing system

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, in which information on the messaging protocol of several applications has been encoded. In general, for each pair of applications, we will have a separate subprogram capable of converting messages between the two applications.

Finally, note that in many cases a message broker is used for advanced enterprise application integration (EAI). In this case, rather than (only) converting messages, a broker is responsible for matching applications based on the messages that are being exchanged.

At the heart of a message broker lies a repository of rules for transforming a message of one type to another. The problem is defining the rules and developing the plugins. 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.

4.4 Multicast Communication

Multicast communication in distributed systems refers to the support for sending data to multiple receivers. Traditionally network-level and transport-level solutions have been implemented and evaluated. A major issue in all solutions was setting up the communication paths for information dissemination. In practice, this involved a huge management effort, in many cases requiring human intervention. Multicast communication can be implemented in many other ways such as:

Application-level tree-based multicasting

The basic idea in application-level multicasting is that nodes organize into an overlay network, which is then used to disseminate information to its members. An important observation is that network routers are not involved in group membership. As a consequence, the connections between nodes in the overlay network may cross several physical links, and as such, routing messages within the overlay may not be optimal in comparison to what could have been achieved by network-level routing.

A crucial design issue is the construction of the overlay network. In essence, there are two approaches:

- First, nodes may organize themselves directly into a tree, meaning that there is a unique (overlay) path between every pair of nodes.
- An alternative approach is that nodes organize into a mesh network in which every node will have multiple neighbors and, in general, there exist multiple paths between every pair of nodes.

The main difference between the two is that the latter generally provides higher robustness: if a connection breaks (e.g., because a node fails), there will still be an opportunity to disseminate information without having to immediately reorganize the entire overlay network.

Flooding-based multicasting

So far, we have assumed that when a message is to be multicast, it is to be received by every node in the overlay network. Strictly speaking, this corresponds to broadcasting. In general, multicasting refers to sending a message to a subset of all the nodes, that is, a specific group of nodes. A key design issue when it comes to multicasting is to minimize the use of intermediate nodes for which the message is not

intended. To make this clear, if the overlay is organized as a multi-level tree, yet only the leaf nodes are the ones who should receive a multicast message, then clearly there may be quite some nodes who need to store and subsequently forward a message that is not meant for them.

One simple way to avoid such inefficiency, is to construct an overlay network per multicast group. As a consequence, multicasting a message m to a group G is the same as broadcasting m to G. The drawback of this solution is that a node belonging to several groups, will, in principle, need to maintain a separate list of its neighbors for each group of which it is a member.

If we assume that an overlay corresponds to a multicast group, and thus that we need to broadcast a message, a naive way of doing so is to apply flooding. In this case, each node simply forwards a message m to each of its neighbors, except to the one from which it received m. Furthermore, if a node keeps track of the messages it received and forwarded, it can simply ignore duplicates. We will roughly see twice as many messages being sent as there are links in the overlay network, making flooding quite inefficient.

Gossip-based data dissemination

An important technique for disseminating information is to rely on epidemic behavior, also referred to as gossiping. Observing how diseases spread among people, researchers have since long investigated whether simple techniques could be developed for spreading information in very large-scale distributed systems. The main goal of these epidemic protocols is to rapidly propagate information among a large collection of nodes using only local information. In other words, there is no central component by which information dissemination is coordinated.

Information dissemination models

As the name suggests, epidemic algorithms are based on the theory of epidemics, which studies the spreading of infectious diseases. In the case of large-scale distributed systems, instead of spreading diseases, they spread information. Research on epidemics for distributed systems also aims at a completely different goal: whereas health organizations will do their utmost best to prevent infectious diseases from spreading across large groups of people, designers of epidemic algorithms for distributed systems will try to “infect” all nodes with new information as fast as possible.

Using the terminology from epidemics, a node that is part of a distributed system is called infected if it holds data that it is willing to spread to other nodes. A node that has not yet seen this data is called susceptible. Finally, an updated node that is not willing or able to spread its data is said to have been removed. Note that we assume we can distinguish old from new data, for example, because it has been timestamped or versioned. In this light, nodes are also said to spread updates.

A popular propagation model is that of anti-entropy. In this model, a node P picks another node Q at random, and subsequently exchanges updates with Q. There are three approaches to exchanging updates:

1. P only pulls in new updates from Q
2. P only pushes its own updates to Q
3. P and Q send updates to each other (i.e., a push-pull approach)

When it comes to rapidly spreading updates, only pushing updates turns out to be a bad choice. Intuitively, this can be understood as follows. First, note that in a pure push-based approach, updates can be propagated only by infected nodes. However, if many nodes are infected, the probability of each one selecting a susceptible node is relatively small. Consequently, chances are that a particular node remains susceptible for a long period simply because it is not selected by an infected node.

In contrast, the pull-based approach works much better when many nodes are infected. In that case, spreading updates is essentially triggered by susceptible nodes. Chances are big that such a node will contact an infected one to subsequently pull in the updates and become infected as well.

4.5 Case Study: Java RMI and Message Passing Interface (MPI)

(Task)

Practice Questions

1. What are the different middleware protocols used in distributed system? Explain.
2. Explain different types of communications in a distributed system?
3. What is RPC? Explain the RPC Process in brief.
4. Explain message oriented communication with suitable example.
5. Explain Multicast communication in distributed system.
6. Describe general architecture of a Message-Queuing system.
7. Explain Transient Asynchronous communication with suitable example.