Chapter three

3. Inter-process Communication

Inter-process communication is at the heart of all distributed systems. It makes no sense to study distributed systems without carefully examining the ways that processes on different machines can exchange information. Communication in distributed systems is always based on low-level message passing as offered by the underlying network. Expressing communication through message passing is harder than using primitives based on shared memory, as available for non-distributed platforms. Modem 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.

The three widely-used models for communication are: Remote Procedure Call (RPC), Message-Oriented Middleware (MOM), and data streaming. An **RPC** aims at hiding most of the intricacies of message passing, and is ideal for client-server applications.

Having powerful and flexible facilities for communication between processes is essential for any distributed system. In traditional network applications, communication is often based on the low-level message-passing primitives offered by the transport layer. An important issue in middleware systems is to offer a higher level of abstraction that will make it easier to express communication between processes than the support offered by the interface to the transport layer.

3.1. 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 A wants to communicate with process B, 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 B.

Although this basic idea sounds simple enough, in order to prevent chaos, A and B have to agree on the meaning of the bits being sent. If A sends a brilliant new novel written in French and encoded in IBM's EBCDIC character code, and B expects the inventory of a supermarket written in English and encoded in ASCII, communication will be less than optimal.

Many different agreements are needed. How many volts should be used to signal a O-bit, and how many volts for a I-bit? How does the receiver know which is the last bit of the message? How can it detect if a message has been damaged or lost, and what should it do if it finds out? How long are numbers, strings, and other data items, and how are they represented? In short, agreements are needed at a variety of levels, varying from the low-level details of bit transmission to the high-level details of how information is to be expressed.

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.

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 protocols. To allow a group of computers to communicate over a network, they must all agree on the protocols to be used. A distinction is made between two general types of protocols. With connection-oriented protocols, before exchanging data the sender and receiver first explicitly establish a connection, and possibly negotiate the protocol they will use. When they are done, they must release (terminate) the connection. The telephone is a connection-oriented communication system. With connectionless protocols, 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 connectionless communication. With computers, both connection-oriented and connectionless communication are common.

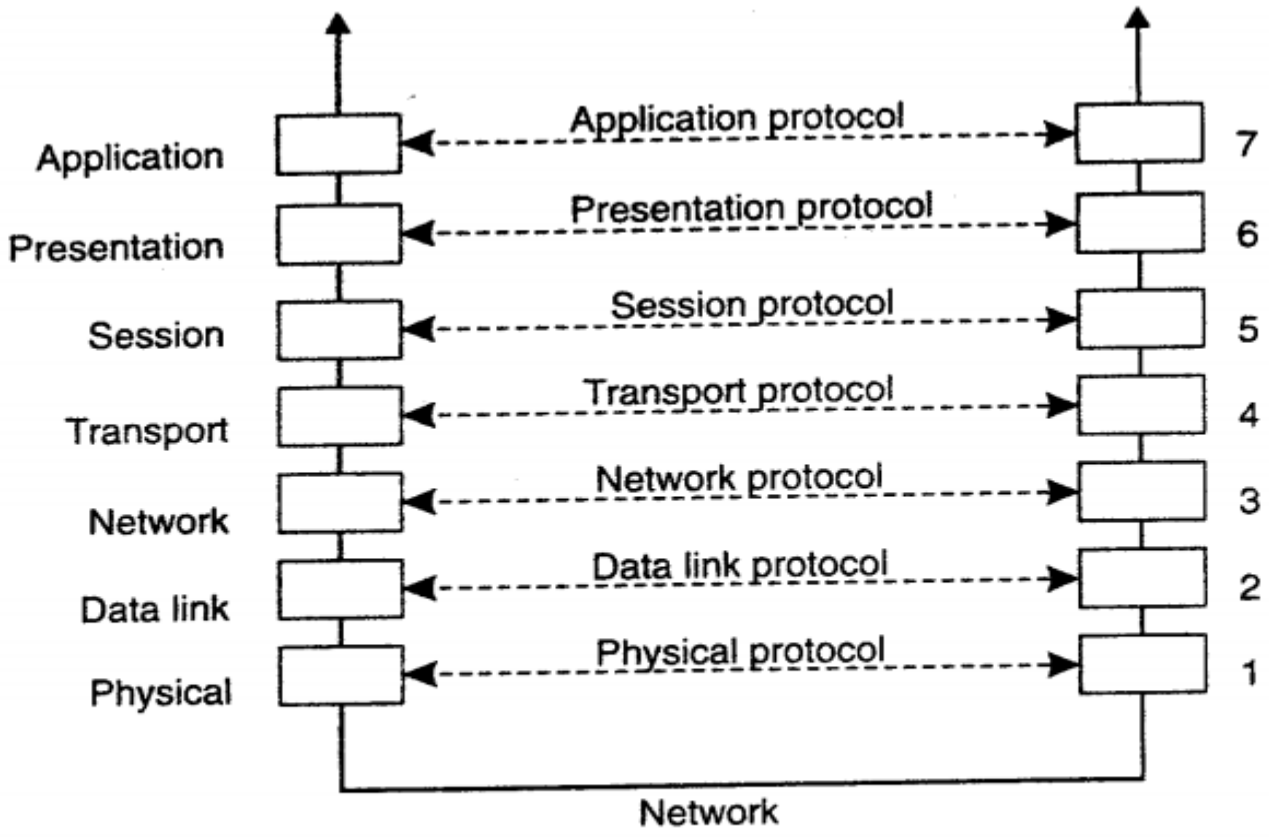

Figure 3-1. Layers, interfaces, and protocols in the OSI model.

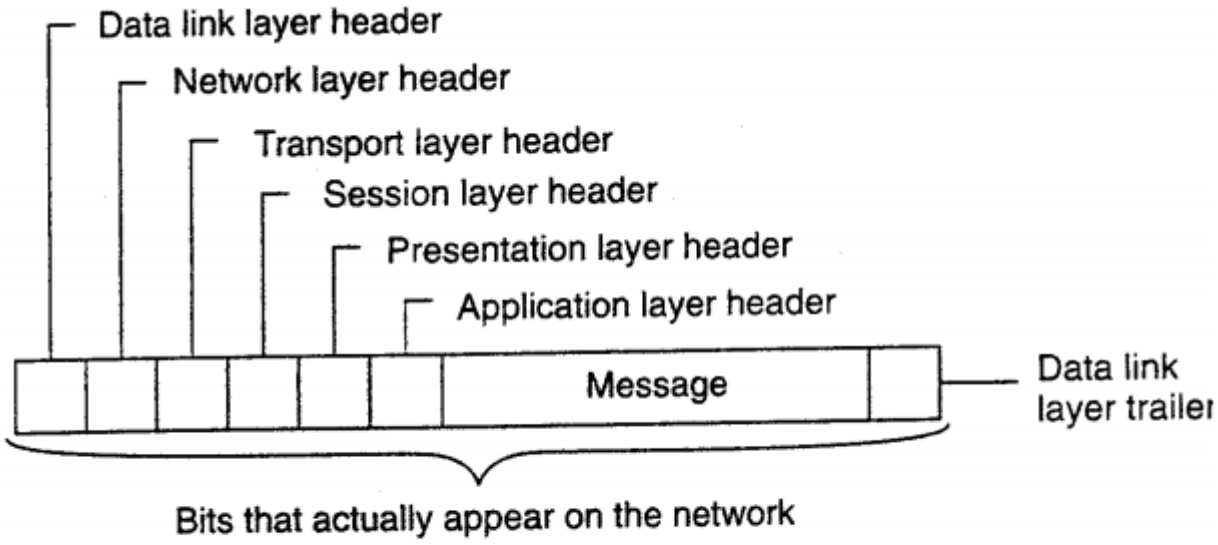

Figure 3-2. A typical message as it appears on the network.

When the message arrives at machine 2, it is passed upward, with each layer stripping off and examining its own header. Finally, the message arrives at the receiver, process B, which may reply to it using the reverse path. The information in the layer n header is used for the layer n protocol.

3.2. 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 Fig. 3.3.

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 tum, 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.

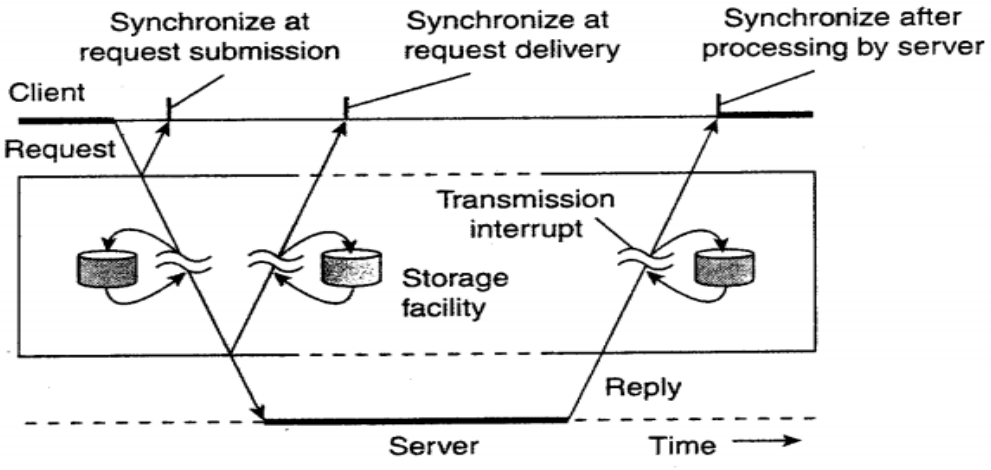

Figure 3.3. Viewing middleware as an intermediate (distributed) service in application-level communication.

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. In contrast, with **transient communication**, a message is stored by the communication system only as long as the sending and receiving application are executing.

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 the time that the recipient returns a response.

Besides persistence and synchronization, we should also make a distinction between **discrete** and **streaming** communication.

In discrete communication:

- the parties communicate by messages, each message forming a complete unit of information.

In contrast, streaming involves:

- sending multiple messages, one after the other, where the messages are related to each other by the order they are sent, or because there is a temporal relationship.

3.2.1. REMOTE PROCEDURE CALL

One of the most widely used abstractions is the Remote Procedure Call (RPC). The essence of an RPC is that a service is implemented by means of a procedure, of which the body is executed at a server. The client is offered only the signature of the procedure, that is, the procedure's name along with its parameters. When the client calls the procedure, the client-side implementation, called a stub, takes care of wrapping the parameter values into a message and sending that to the

server. The latter calls the actual procedure and returns the results, again in a message. The client's stub extracts the result values from the return message and passes it back to the calling client application. RPCs offer synchronous communication facilities, by which a client is blocked until the server has sent a reply. Although variations of either mechanism exist by which this strict synchronous model is relaxed, it turns out that general-purpose, high-level message-oriented models are often more convenient.

In a nutshell, what Birrell and Nelson suggested was allowing 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. To start with, because 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.

i. Client and Server Stubs

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.

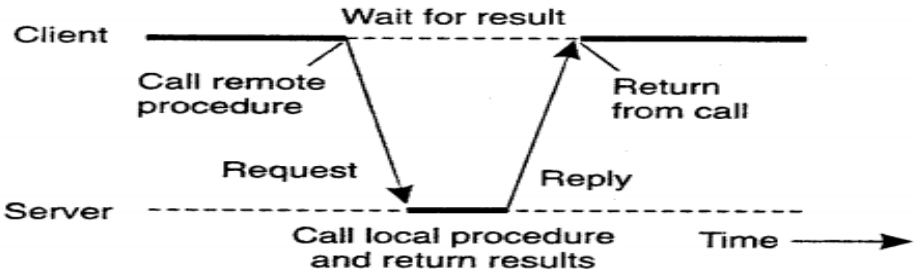

Figure 3.4. Principle of RPC between a client and server program.

ii. Basic RPC Operation

The general activities in remote procedure call occurs in the following steps:

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 as sends the message to the remote as.
4. The remote as gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local as.
8. The server's as sends the message to the client's as.
9. The client's as gives the message to the client stub.
10. The stub unpacks the result and returns to the client

3.2.2. 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 message-oriented models, the issues are whether or not communication is persistent, and whether or not communication is synchronous. The essence of persistent communication is that a message that is submitted for transmission, is stored by the communication system as long as it takes to deliver it. In other words, neither the sender nor the receiver needs to be up and running for message transmission to take place. In transient communication, no storage facilities are offered, so that the receiver must be prepared to accept the message when it is sent.

In asynchronous communication, the sender is allowed to continue immediately after the message has been submitted for transmission, possibly before it has even been sent. In synchronous communication, the sender is blocked at least until a message has been received. Alternatively, the sender may be blocked until message delivery has taken place or even until the receiver has responded as with RPCs.

Message-oriented middleware models generally offer persistent asynchronous communication, and are used where RPCs are not appropriate. They are often used to assist the integration of

(widely-dispersed) collections of databases into large-scale information systems. Other applications include e-mail and workflow.

3.2.3. STREAM-ORIENTED COMMUNICATION

Communication as discussed so far 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 be-tween 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.

A very different form of communication is that of streaming, in which the issue is whether or not two successive messages have a temporal relationship. In continuous data streams, a maximum end-to-end delay is specified for each message. In addition, it is also required that messages are sent subject to a minimum end-to-end delay. Typical examples of such continuous data streams are video and audio streams. Exactly what the temporal relations are, or what is expected from the underlying communication subsystem in terms of quality of service is often difficult to specify and to implement. A complicating factor is the role of jitter. Even if the average performance is acceptable, substantial variations in delivery time may lead to unacceptable performance.

3.2.4. MULTICAST COMMUNICATION

Finally, an important class of communication protocols in distributed systems is multicasting. The basic idea is to disseminate information from one sender to multiple receivers. We have discussed two different approaches. First, multicasting can be achieved by setting up a tree from the sender to the receivers. Considering that it is now well understood how nodes can self-organize into peer-to-peer system, solutions have also appeared to dynamically set up trees in a decentralized fashion.

Another important class of dissemination solutions deploys epidemic proto-cols. These protocols have proven to be very simple, yet extremely robust. Apart from merely spreading messages, epidemic protocols can also be efficiently deployed for aggregating information across a large distributed system.

End of Chapter Three