Chapter-2

Communication

2.1 Layered protocols

The basic idea for the layered style is simple: components are organized in a layered fashion where a component at layer L_j can make a downcall to a component at a lower-level layer L_i (with i < j) and generally expects a response. Only in exceptional cases will an upcall be made to a higher-level component. The three common cases are shown in Figure 2.1.

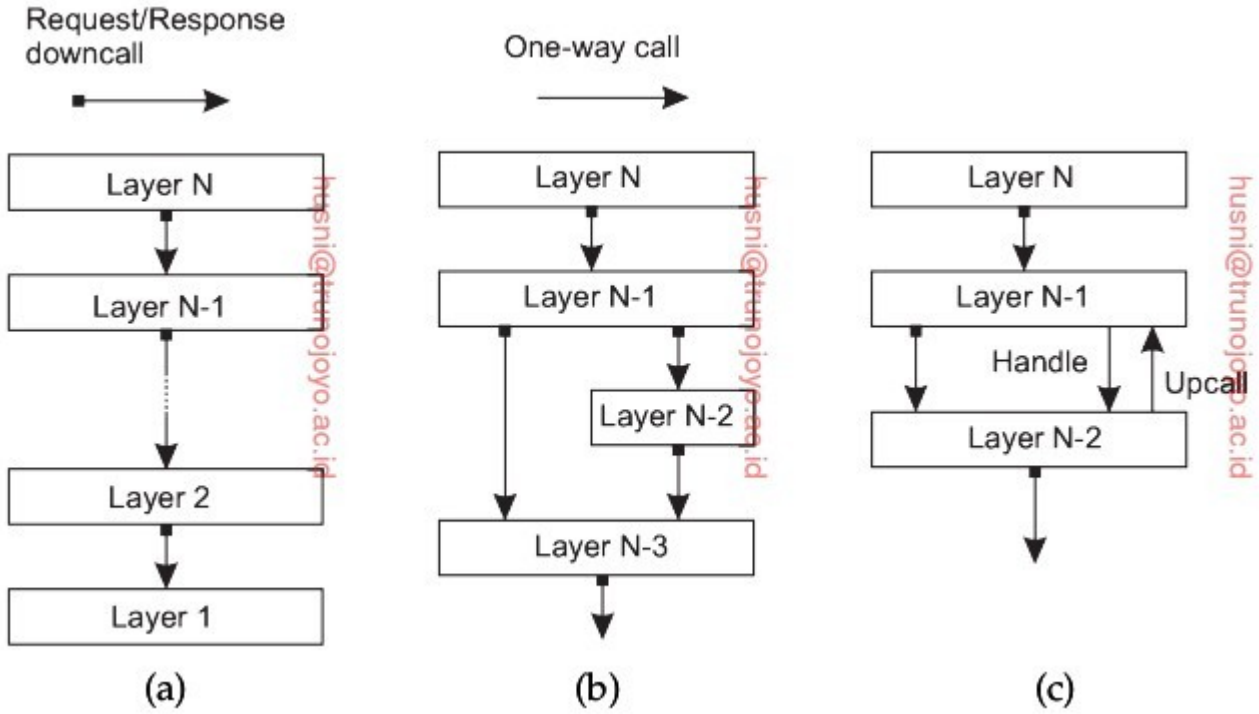

Figure 2.1 (a) Pure layered organization. (b) Mixed layered organization. (c) Layered organization with upcalls

Figure 2.1(a) shows a standard organization in which only downcalls to the next lower layer are made. This organization is commonly deployed in the case of network communication.

In many situations we also encounter the organization shown in Figure 2.1(b). Consider, for example, an application A that makes use of a library L_{OS} to interface to an operating system. At the same time, the application uses a specialized mathematical library L_{math} that has been

implemented by also making use of L_{OS} . In this case, referring to Figure 2.1(b), A is implemented at layer N − 1, L_{math} at layer N − 2, and L_{OS} which is common to both of them, at layer N − 3.

Finally, a special situation is shown in Figure 2.1(c). In some cases, it is convenient to have a lower layer do an upcall to its next higher layer. A typical example is when an operating system signals the occurrence of an event, to which end it calls a user-defined operation for which an application had previously passed a reference (typically referred to as a handle).

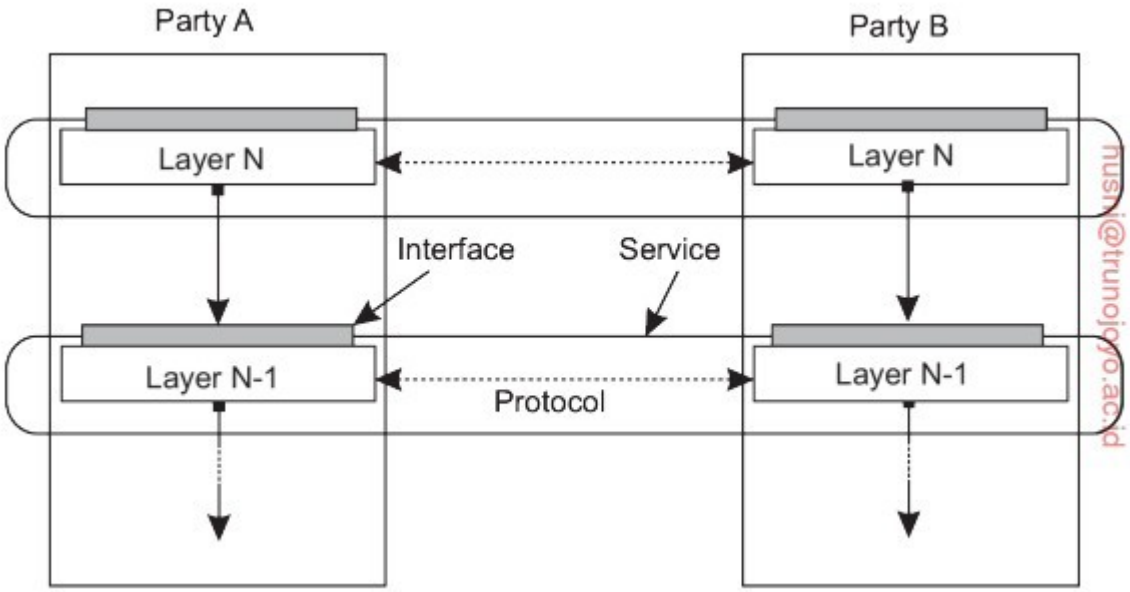

Figure 2.2: A layered communication-protocol stack, showing the difference between a service, its interface, and the protocol it deploys.

A well-known and ubiquitously applied layered architecture is that of so-called communication-protocol stacks.

In communication-protocol stacks, each layer implements one or several communication services allowing data to be sent from a destination to one or several targets. To this end, each layer offers an interface specifying the functions that can be called. In principle, the interface should completely hide the actual implementation of a service. Another important concept in the case of communication is that of a (communication) protocol, which describes the rules that parties will follow in order to exchange information.

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, usually abbreviated as **ISO OSI** or sometimes just the OSI model. It should be emphasized that the protocols that were developed as part of the OSI model were never widely used and are essentially dead. However, the underlying model itself has proved to be quite useful for understanding computer networks.

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.

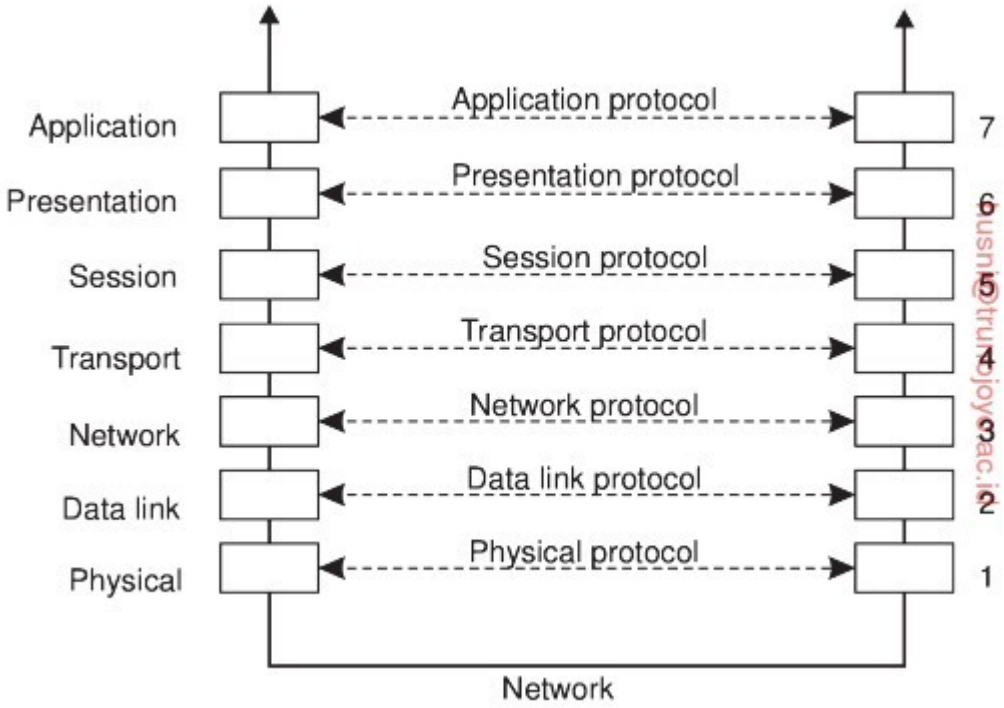

Figure 2.3: Layers, interfaces, and protocols in the OSI model.

In the OSI model, communication is divided into seven levels or layers, as shown in Figure 2.3. 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.

2.2 Client-server TCP

In many distributed systems, reliable point-to-point communication is established by making use of a reliable transport protocol, such as TCP. TCP masks omission failures, which occur in the form of lost messages, by using acknowledgments and retransmissions. Such failures are completely hidden from a TCP client.

However, crash failures of connections are not masked. A crash failure may occur when (for whatever reason) a TCP connection is abruptly broken so that no more messages can be transmitted through the channel. In most cases, the client is informed that the channel has crashed by raising an exception. The only way to mask such failures is to let the distributed system attempt to automatically set up a new connection, by simply resending a connection request. The underlying assumption is that the other side is still, or again, responsive to such requests.

2.3 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.

Types of Communication

In the remainder of this chapter, we concentrate on high-level middleware communication services. Before doing so, there are other general criteria for distinguishing (middleware) 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 2.4. 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.

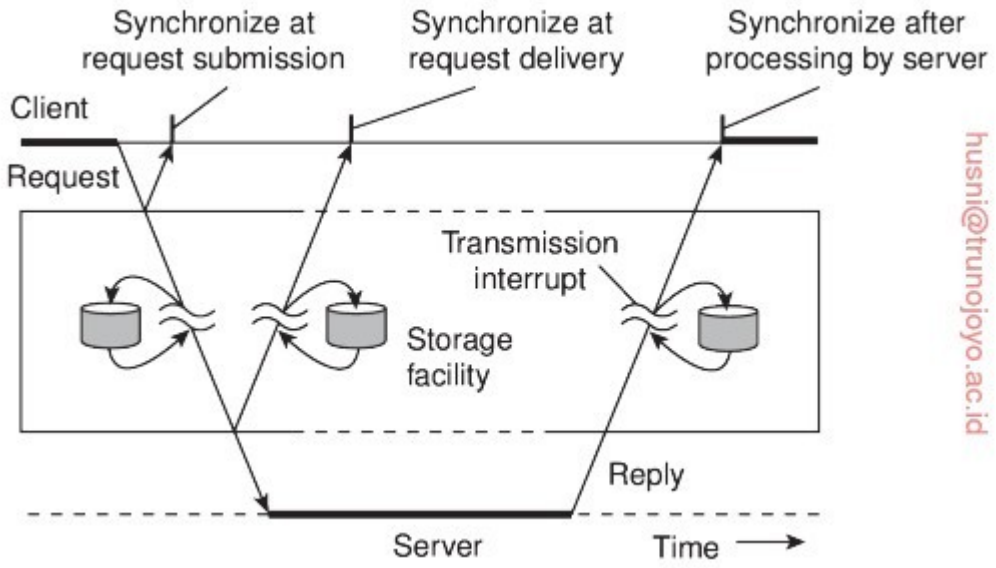

Figure 2.4: Viewing middleware as an intermediate (distributed) service in application-level communication.

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 shown in Figure 2.4. 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 2.4, 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, which we discuss later in this chapter. Likewise, transient communication with synchronization after the request has been fully processed is also widely used. This scheme corresponds with remote procedure calls, which we discuss next.

2.4 Remote procedure call and remote object invocation

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 this section we will examine the concept, its implementation, its strengths, and its weaknesses.

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. 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.

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 2.5. Following the call to send, the client stub calls receive, blocking itself until the reply comes back.

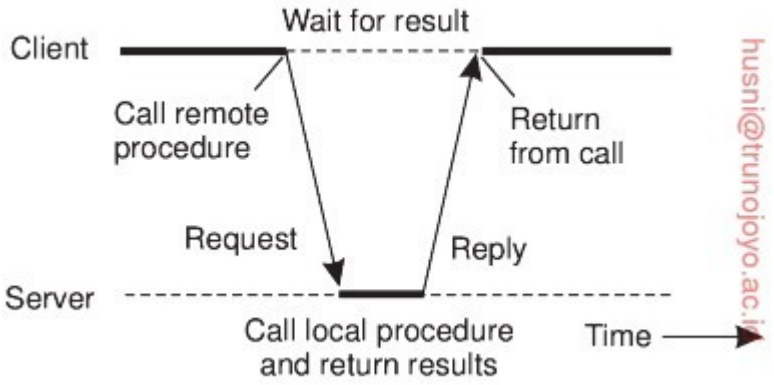

Figure 2.5: The principle of RPC between a client and server program.

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.

This blissful ignorance on the part of the client is the beauty of the whole scheme. As far as it is concerned, 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.

To summarize, a remote procedure call occurs in the following steps:

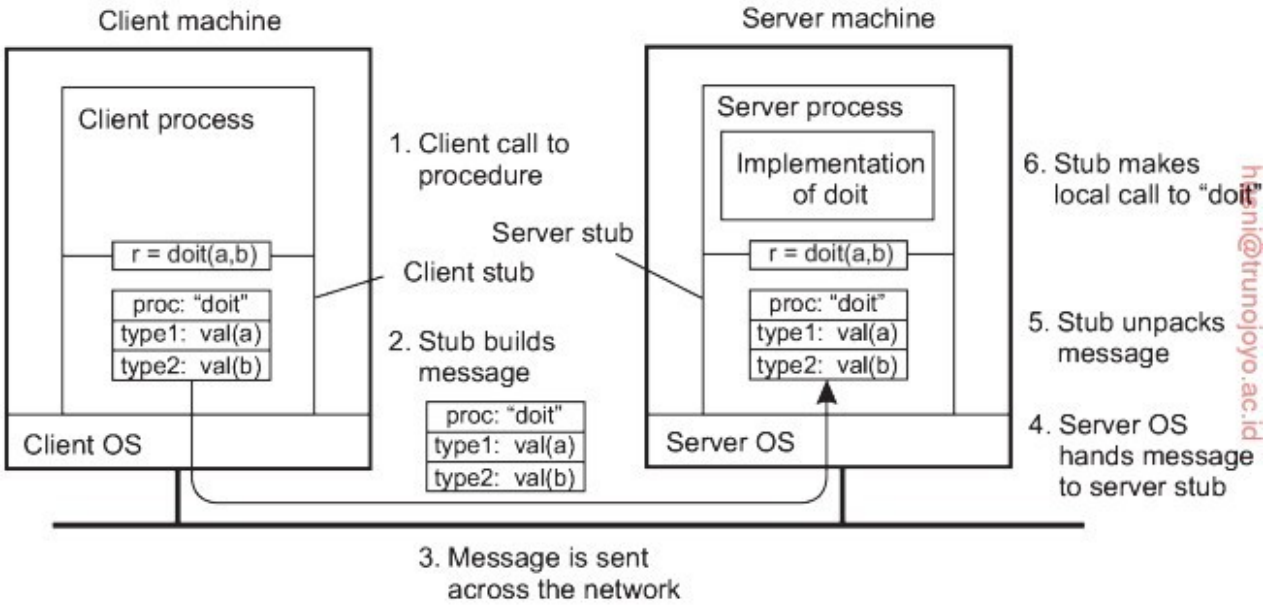

Figure 2.6: The steps involved in calling a remote procedure doit(a,b). The return path for the result is not shown.

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.

The first steps are shown in Figure 2.6 for an abstract two-parameter procedure doit(a,b), where we assume that parameter a is of type type1, and b of type type2. The net effect of all these steps is to convert the local call by the client procedure to the client stub, to a local call to the server procedure without either client or server being aware of the intermediate steps or the existence of the network.

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. Returning to our append operation, we thus need to ensure that its two parameters (data and dbList) are sent over the network and correctly interpreted by the server. The thing to realize here is that, in the end, the server will just be seeing a series of bytes coming in that constitute the original message sent by the client. However, no additional information on what those bytes mean is normally provided with the message, let alone that we would be facing the same problem again: how should the meta-information be recognized as such by the server?

Besides this interpretation problem, we also need to handle the case that the placement of bytes in memory may differ between machine architectures. In particular, we need to account for the fact that some machines, such as the Intel Pentium, number their bytes from right to left, whereas many others, such as the older ARM processors, number them the other way (ARM now supports both). The Intel format is called little endian and the (older) ARM format is called big endian. Byte ordering is also important for networking: also here we can witness that machines may use a different ordering when transmitting (and thus receiving) bits and bytes. However, big endian is what is normally used for transferring bytes across a network.

The solution to this problem is to transform data that is to be sent to a machine- and network-independent format, next to making sure that both communicating parties expect the same message data type to be transmitted. The latter can typically be solved at the level of programming languages. The former is accomplished by using machine-dependent routines that transform data to and from machine- and network-independent formats.

Marshaling and unmarshaling is all about this transformation to neutral formats and forms an essential part of remote procedure calls.

We now come to a difficult problem: How are pointers, or in general, references passed? The answer is: only with the greatest of difficulty, if at all. A pointer is meaningful only within the address space of the process in which it is being used. Getting back to our append example, we stated that the second parameter, dbList, is implemented by means of a reference to a list stored

in a database. If that reference is just a pointer to a local data structure somewhere in the caller's main memory, we cannot simply pass it to the server. The transferred pointer value will most likely be referring to something completely different.

One solution is just to forbid pointers and reference parameters in general. However, these are so important that this solution is highly undesirable. In fact, it is often not necessary either. First, reference parameters are often used with fixed-sized data types, such as static arrays, or with dynamic data types for which it is easy to compute their size at runtime, such as strings or dynamic arrays. In such cases, we can simply copy the entire data structure to which the parameter is referring, effectively replacing the copy-by-reference mechanism by copy-by-value/restore. Although this is semantically not always identical, it frequently is good enough. An obvious optimization is that when the client stub knows the referred data will be only read, there is no need to copy it back when the call has finished. Copy-by-value is thus good enough.

2.5 Message oriented and stream 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.

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. Then, we discuss messaging systems that assume that parties are executing at the time of communication. Finally, we will examine message-queuing systems that allow processes to exchange information, even if the other party is not executing at the time communication is initiated.

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.

pecial 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 operations. Also, standard interfaces make it easier to port an application to a different machine.

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 Figure 2.7.

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

Figure 2.7: The socket operations for TCP/IP.

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 2.8.

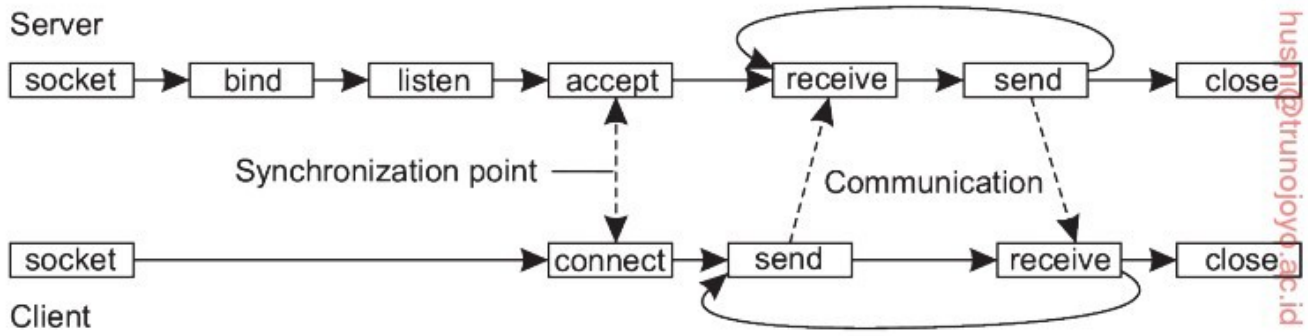

Figure 2.8: Connection-oriented communication pattern using sockets.