send some bits, then stop the sending process and run something else while waiting for the reply. The trouble with this strategy is that computers are becoming so inexpensive, that for many applications, each process has its own computer, so there is nothing else to run. Wasting the CPU time is not important, since it is cheap, but it is clear that going from 64 Kbps to 622 Mbps has not bought a 10,000-fold gain in performance, even in communication-limited applications.

The effect of the transcontinental delay can show up in various ways. For example, if some application program in New York has to make 20 sequential requests from a server in California to get an answer, the 600-msec delay will be noticeable to the user, as people find delays above 200 msec annoying.

Alternatively, we could move the computation itself to the machine in California and let each user keystroke be sent as a separate cell across the country and come back to be displayed. Doing this will add 60 msec to each keystroke, which no one will notice. However, this reasoning quickly leads us to abandoning the idea of a distributed system and putting all the computing in one place, with remote users. In effect, we have built a big centralized timesharing system with just the users distributed.

One observation that does relate to specific properties of ATM is the fact that switches are permitted to drop cells if they get congested. Dropping even one cell probably means waiting for a timeout and having the whole packet be retransmitted. For services that need a uniform rate, such as playing music, this could be a problem. (Oddly enough, the ear is far more sensitive than the eye to irregular delivery.)

As a consequence of these and other problems, while high-speed networks in general and ATM in particular introduce new opportunities, taking advantage of them will not be simple. Considerable research will be needed before we know how to deal with them effectively.

## 2.3. THE CLIENT-SERVER MODEL

While ATM networks are going to be important in the future, for the moment they are too expensive for most applications, so let us go back to more conventional networking. At first glance, layered protocols along the OSI lines look like a fine way to organize a distributed system. In effect, a sender sets up a connection (a bit pipe) with the receiver, and then pumps the bits in, which arrive without error, in order, at the receiver. What could be wrong with this?

Plenty. To start with, look at Fig. 2-2. The existence of all those headers generates a considerable amount of overhead. Every time a message is sent it must be processed by about half a dozen layers, each one generating and adding a header on the way down or removing and examining a header on the way up. All of this work takes time. On wide-area networks, where the number of bits/sec that can be sent is typically fairly low (often as little as 64K bits/sec), this overhead is not serious. The limiting factor is the capacity of the lines, and even with all the header manipulation, the CPUs are fast enough to keep the lines running at full speed. Thus a wide-area distributed system can probably use the OSI or TCP/IP protocols without any loss in (the already meager) performance. Aith ATM, even here serious problems may arise.

However, for a LAN-based distributed system, the protocol overhead is often substantial. So much CPU time is wasted running protocols that the effective throughput over the LAN is often only a fraction of what the LAN can do. As a consequence, most LAN-based distributed systems do not use layered protocols at all, or if they do, they use only a subset of the entire protocol stack.

In addition, the OSI model addresses only a small aspect of the problem—getting the bits from the sender to the receiver (and in the upper layers, what they mean). It does not say anything about how the distributed system should be structured. Something more is needed.

### 2.3.1. Clients and Servers

This something is often the client-server model that we introduced in the preceding chapter. The idea behind this model is to structure the operating system as a group of cooperating processes, called **servers**, that offer services to the users, called **clients**. The client and server machines normally all run the same microkernel, with both the clients and servers running as user processes, as we saw earlier. A machine may run a single process, or it may run multiple clients, multiple servers, or a mixture of the two.

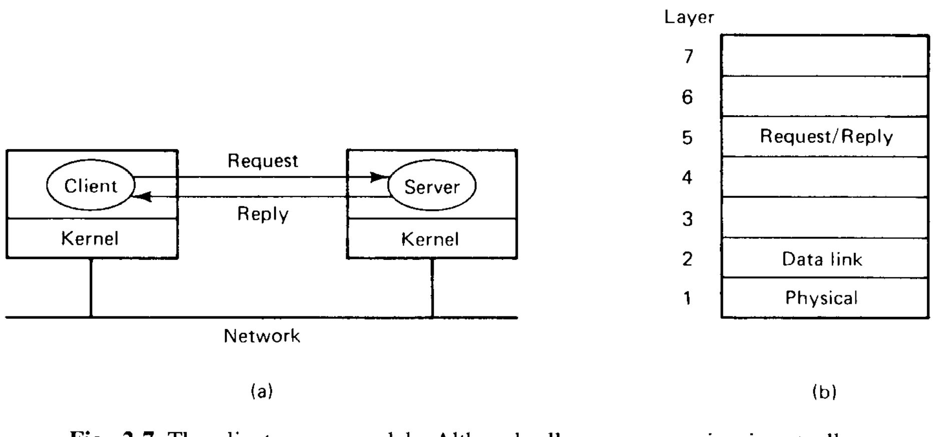

**Fig. 2-7.** The client-server model. Although all message passing is actually done by the kernels, this simplified form of drawing will be used when there is no ambiguity.

To avoid the considerable overhead of the connection-oriented protocols such as OSI or TCP/IP, the client server model is usually based on a simple, connectionless **request/reply protocol**. The client sends a request message to the server asking for some service (e.g., read a block of a file). The server does the work and returns the data requested or an error code indicating why the work could not be performed, as depicted in Fig. 2-7(a).

The primary advantage of Fig. 2-7(a) is the simplicity. The client sends a request and gets an answer. No connection has to be established before use or torn down afterward. The reply message serves as the acknowledgement to the request.

From the simplicity comes another advantage: efficiency. The protocol stack is shorter and thus more efficient. Assuming that all the machines are identical, only three levels of protocol are needed, as shown in Fig. 2-7(b). The physical and data link protocols take care of getting the packets from client to server and back. These are always handled by the hardware, for example, an Ethernet or token ring chip. No routing is needed and no connections are established, so layers 3 and 4 are not needed. Layer 5 is the request/reply protocol. It defines the set of legal requests and the set of legal replies to these requests. There is no session management because there are no sessions. The upper layers are not needed either.

Due to this simple structure, the communication services provided by the (micro)kernel can, for example, be reduced to two system calls, one for sending messages and one for receiving them. These system calls can be invoked through library procedures, say, *send(dest, &mptr)* and *receive(addr, &mptr)*. The former sends the message pointed to by *mptr* to a process identified by *dest* and causes the caller to be blocked until the message has been sent. The latter causes the caller to be blocked until a message arrives. When one does, the message is copied to the buffer pointed to by *mptr* and the caller is unblocked. The *addr* parameter specifies the address to which the receiver is listening. Many variants of these two procedures and their parameters are possible. We will discuss some of these later in this chapter.

## 2.3.2. An Example Client and Server

To provide more insight into how clients and servers work, in this section we will present an outline of a client and a file server in C. Both the client and the server need to share some definitions, so we will collect these into a file called *header.h*, which is shown in Fig. 2-8. Both the client and server include these using the

```
#include <header.h>
```

statement. This statement has the effect of causing a preprocessor to literally acknowledgement, reducing the sequence to three packets. Finally, in Fig. 2-16(d), we see a nervous client checking to see if the server is still there.

## 2.4. REMOTE PROCEDURE CALL

Although the client-server model provides a convenient way to structure a distributed operating system, it suffers from one incurable flaw: the basic paradigm around which all communication is built is input/output. The procedures *send* and *receive* are fundamentally engaged in doing I/O. Since I/O is not one of the key concepts of centralized systems, making it the basis for distributed computing has struck many workers in the field as a mistake. Their goal is to make distributed computing look like centralized computing. Building everything around I/O is not the way to do it.

This problem has long been known, but little was done about it until a paper by Birrell and Nelson (1984) introduced a completely different way of attacking the problem. 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, 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 or I/O 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, 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 operating systems.

### 2.4.1. Basic RPC Operation

To understand how RPC works, it is important first to fully understand how a conventional (i.e., single machine) procedure call works. Consider a call like

```
count = read(fd, buf, nbytes);
```

where *fd* is an integer, *buf* is an array of characters, and *nbytes* is another integer. If the call is made from the main program, the stack will be as shown in Fig. 2-17(a) before the call. To make the call, the caller pushes the parameters onto the stack in order, last one first, as shown in Fig. 2-17(b). (The reason that C compilers push the parameters in reverse order has to do with *printf*—by doing so, *printf* can always locate its first parameter, the format string.) After *read* has finished running, it puts the return value in a register, removes the return address, and transfers control back to the caller. The caller then removes the parameters from the stack, returning it to the original state, as shown in Fig. 2-17(c).

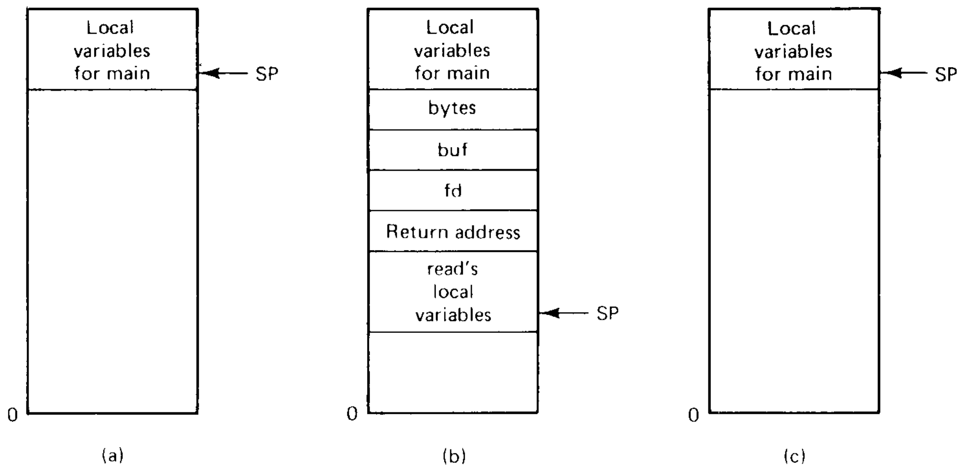

**Fig. 2-17.** (a) The stack before the call to *read*. (b) The stack while the called procedure is active. (c) The stack after the return to the caller.

Several things are worth noting. For one, in C, parameters can be **call-by-value** or **call-by-reference**. A value parameter, such as *fd* or *nbytes*, is simply copied to the stack as shown in Fig. 2-17(b). To the called procedure, a value parameter is just an initialized local variable. The called procedure may modify it, but such changes do not affect the original value at the calling side.

A reference parameter in C is a pointer to a variable (i.e., the address of the variable), rather than the value of the variable. In the call to *read*, the second parameter is a reference parameter because arrays are always passed by reference in C. What is actually pushed onto the stack is the address of the character array. If the called procedure uses this parameter to store something into the character array, it *does* modify the array in the calling procedure. The difference between call-by-value and call-by-reference is quite important for RPC, as we shall see.

One other parameter passing mechanism also exists, although it is not used in C. It is called **call-by-copy/restore**. It consists of having the variable copied to the stack by the caller, as in call-by-value, and then copied back after the call, overwriting the caller's original value. Under most conditions, this achieves the same effect as call-by-reference, but in some situations, such as the same parameter being present multiple times in the parameter list, the semantics are different.

The decision of which parameter passing mechanism to use is normally made by the language designers and is a fixed property of the language. Sometimes it depends on the data type being passed. In C, for example, integers and other scalar types are always passed by value, whereas arrays are always passed by reference, as we have seen. In contrast, Pascal programmers can choose which mechanism they want for each parameter. The default is call-by-value, but programmers can force call-by-reference by inserting the keyword **var** before specific parameters. Some Ada® compilers use copy/restore for **in out** parameters, but others use call-by-reference. The language definition permits either choice, which makes the semantics a bit fuzzy.

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 needs to read some data from a file. The programmer puts a call to *read* in the code to get the data. In a traditional (single-processor) system, the *read* routine is extracted from the library by the linker and inserted into the object program. It is a short procedure, usually written in assembly language, that puts the parameters in registers and then issues a READ system call by trapping to the kernel. In essence, the *read* procedure is a kind of interface between the user code and the operating system.

Even though *read* issues a kernel trap, it is called in the usual way, by pushing the parameters onto the stack, as shown in Fig. 2-17. Thus the programmer does not know that *read* is actually doing something fishy.

RPC achieves its transparency in an analogous way. When *read* is actually a remote procedure (e.g., one that will run on the file server's machine), a different version of *read*, called a **client stub**, is put into the library. Like the original one, it too, is called using the calling sequence of Fig. 2-17. Also like the original one, it too, traps to the kernel. Only unlike the original one, it does not put the parameters in registers and ask the kernel to give it data. Instead, it packs the parameters into a message and asks the kernel to send the message to the server as illustrated in Fig. 2-18. 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 kernel passes it up to a **server stub** that is bound with the actual server. Typically the server stub will have called *receive* and be blocked waiting for incoming messages. The server stub

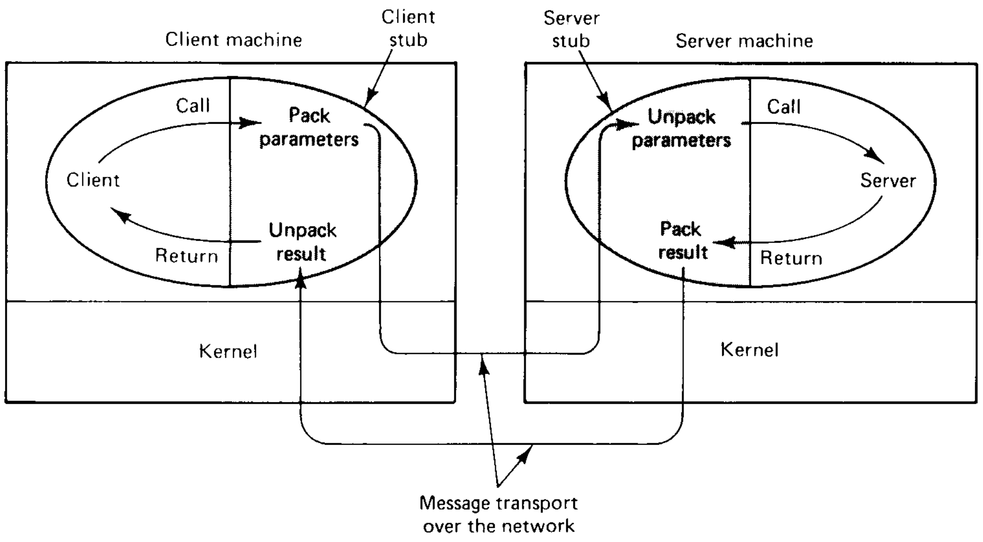

**Fig. 2-18.** Calls and messages in an RPC. Each ellipse represents a single process, with the shaded portion being the stub.

unpacks the parameters from the message and then calls the server procedure in the usual way (i.e., as in Fig. 2-17). 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 the usual way. For example, in the case of *read*, the server will fill the buffer, pointed to by the second parameter, with the data. This buffer will be internal to the server stub.

When the server stub gets control back after the call has completed, it packs the result (the buffer) in a message and calls *send* to return it to the client. Then it goes back to the top of its own loop to call *receive*, waiting for the next message.

When the message gets back to the client machine, the kernel sees that it is addressed to the client process (to the stub part of that process, but the kernel does not know that). The message is copied to the waiting buffer and the client process 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 *read*, all it knows is that its data are available. It has no idea that the work was done remotely instead of by the local kernel.

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* as in Fig. 2-9.

All the details of the message passing are hidden away in the two library procedures, just as the details of actually making system call traps are hidden away in traditional libraries.

To summarize, a 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 traps to the kernel.
3. The kernel sends the message to the remote kernel.
4. The remote kernel 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 traps to the kernel.
8. The remote kernel sends the message to the client's kernel.
9. The client's kernel gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

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.

### 2.4.2. Parameter Passing

The function of the client stub is to take its parameters, pack them into a message, and send it to the server stub. While this sounds straightforward, it is not quite as simple as it at first appears. In this section we will look at some of the issues concerned with parameter passing in RPC systems. Packing parameters into a message is called **parameter marshaling**.

As the simplest possible example, consider a remote procedure, *sum*(*i*, *j*), that takes two integer parameters and returns their arithmetic sum. (As a practical matter, one would not normally make such a simple procedure remote due to the overhead, but as an example it will do.) The call to *sum*, with parameters 4 and 7, is shown in the left-hand portion of the client process in Fig. 2-19. The client stub takes its two parameters and puts them in a message as indicated. It also puts the name or number of the procedure to be called in the message because the server might support several different calls, and it has to be told which one is required.

When the message arrives at the server, the stub examines the message to see which procedure is needed, and then makes the appropriate call. If the

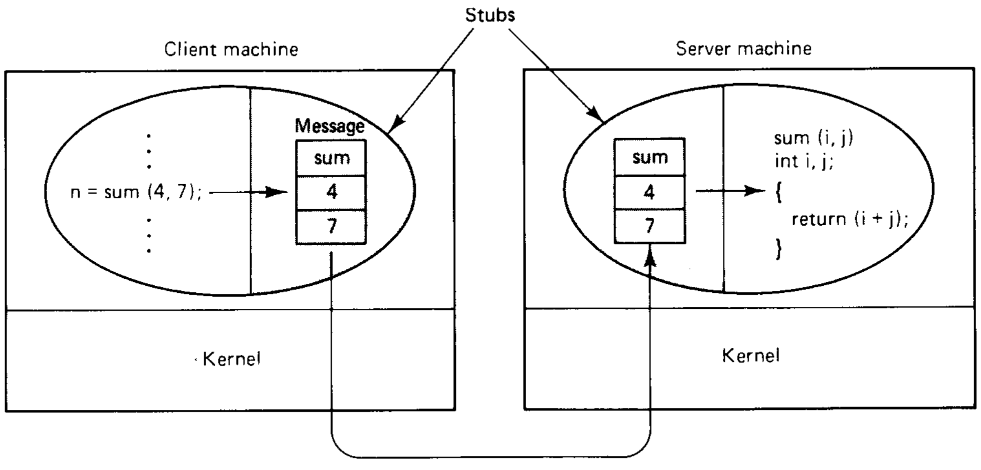

**Fig. 2-19.** Computing *sum*(4, 7) remotely.

server also supports the remote procedures *difference*, *product*, and *quotient*, the server stub might have a switch statement in it, to select the procedure to be called, depending on the first field of the message. The actual call from the stub to the server looks much like the original client call, except that the parameters are variables initialized from the incoming message, rather than constants.

When the server has finished, the server stub gains control again. It takes the result, provided by the server, and packs it into a message. This message is sent back to the client stub, which unpacks it and returns the value to the client procedure (not shown in the figure).

As long as the client and server machines are identical and all the parameters and results are scalar types, such as integers, characters, and Booleans, this model works fine. However, in a large distributed system, it is common that multiple machine types are present. Each machine often has its own representation for numbers, characters, and other data items. For example, IBM mainframes use the EBCDIC character code, whereas IBM personal computers use ASCII. As a consequence, it is not possible to pass a character parameter from an IBM PC client to an IBM mainframe server using the simple scheme of Fig. 2-19: the server will interpret the character incorrectly.

Similar problems can occur with the representation of integers (1s complement versus 2s complement), and especially with floating-point numbers. In addition, an even more annoying problem exists because some machines, such as the Intel 486, number their bytes from right to left, whereas others, such as the Sun SPARC, number them the other way. The Intel format is called **little**