1

Introduction to Distributed Systems

Computer systems are undergoing a revolution. From 1945, when the modern computer era began, until about 1985, computers were large and expensive. Even minicomputers normally cost tens of thousands of dollars each. As a result, most organizations had only a handful of computers, and for lack of a way to connect them, these operated independently from one another.

Starting in the mid-1980s, however, two advances in technology began to change that situation. The first was the development of powerful microprocessors. Initially, these were 8-bit machines, but soon 16-, 32-, and even 64-bit CPUs became common. Many of these had the computing power of a decent-sized mainframe (i.e., large) computer, but for a fraction of the price.

The amount of improvement that has occurred in computer technology in the past half century is truly staggering and totally unprecedented in other industries. From a machine that cost 10 million dollars and executed 1 instruction per second, we have come to machines that cost 1000 dollars and execute 10 million instructions per second, a price/performance gain of 10^{11}. If cars had improved at this rate in the same time period, a Rolls Royce would now cost 10 dollars and get a billion miles per gallon. (Unfortunately, it would probably also have a 200-page manual telling how to open the door.)

The second development was the invention of high-speed computer networks. The **local area networks** or **LANs** allow dozens, or even hundreds, of machines within a building to be connected in such a way that small amounts of information can be transferred between machines in a millisecond or so. Larger amounts of data can be moved between machines at rates of 10 to 100 million bits/sec and sometimes more. The **wide area networks** or **WANs** allow millions of machines all over the earth to be connected at speeds varying from 64 Kbps (kilobits per second) to gigabits per second for some advanced experimental networks.

The result of these technologies is that it is now not only feasible, but easy, to put together computing systems composed of large numbers of CPUs connected by a high-speed network. They are usually called **distributed systems,** in contrast to the previous **centralized systems** (or **single-processor systems**) consisting of a single CPU, its memory, peripherals, and some terminals.

There is only one fly in the ointment: software. Distributed systems need radically different software than centralized systems do. In particular, the necessary operating systems are only beginning to emerge. The first few steps have been taken, but there is still a long way to go. Nevertheless, enough is already known about these distributed operating systems that we can present the basic ideas. The rest of this book is devoted to studying concepts, implementation, and examples of distributed operating systems.

1.1. WHAT IS A DISTRIBUTED SYSTEM?

Various definitions of distributed systems have been given in the literature, none of them satisfactory and none of them in agreement with any of the others. For our purposes it is sufficient to give a loose characterization:

A distributed system is a collection of independent computers that appear to the users of the system as a single computer.

This definition has two aspects. The first one deals with hardware: the machines are autonomous. The second one deals with software: the users think of the system as a single computer. Both are essential. We will come back to these points later in this chapter, after going over some background material on both the hardware and the software.

Rather than going further with definitions, it is probably more helpful to give several examples of distributed systems. As a first example, consider a network of workstations in a university or company department. In addition to each user's personal workstation, there might be a pool of processors in the machine room that are not assigned to specific users but are allocated dynamically as needed. Such a system might have a single file system, with all files accessible from all machines in the same way and using the same path name. Furthermore, when a user typed a command, the system could look for the best place to execute that command, possibly on the user's own workstation, possibly on an idle workstation belonging to someone else, and possibly on one of the unassigned processors in the machine room. If the system as a whole looked and acted like a classical single-processor timesharing system, it would qualify as a distributed system.

As a second example, consider a factory full of robots, each containing a powerful computer for handling vision, planning, communication, and other tasks. When a robot on the assembly line notices that a part it is supposed to install is defective, it asks another robot in the parts department to bring it a replacement. If all the robots act like peripheral devices attached to the same central computer and the system can be programmed that way, it too counts as a distributed system.

As a final example, think about a large bank with hundreds of branch offices all over the world. Each office has a master computer to store local accounts and handle local transactions. In addition, each computer has the ability to talk to all other branch computers and with a central computer at headquarters. If transactions can be done without regard to where a customer or account is, and the users do not notice any difference between this system and the old centralized mainframe that it replaced, it too would be considered a distributed system.

1.2. GOALS

Just because it is possible to build distributed systems does not necessarily mean that it is a good idea. After all, with current technology it is possible to put four floppy disk drives on a personal computer. It is just that doing so would be pointless. In this section we will discuss the motivation and goals of typical distributed systems and look at their advantages and disadvantages compared to traditional centralized systems.

1.2.1. Advantages of Distributed Systems over Centralized Systems

The real driving force behind the trend toward decentralization is economics. A quarter of a century ago, computer pundit and gadfly Herb Grosch stated what later came to be known as Grosch's law: The computing power of a CPU is proportional to the square of its price. By paying twice as much, you could get four times the performance. This observation fit the mainframe technology of its time quite well, and led most organizations to buy the largest single machine they could afford.

With microprocessor technology, Grosch's law no longer holds. For a few hundred dollars you can get a CPU chip that can execute more instructions per second than one of the largest 1980s mainframes. If you are willing to pay twice as much, you get the same CPU, but running at a somewhat higher clock speed.

As a result, the most cost-effective solution is frequently to harness a large number of cheap CPUs together in a system. Thus the leading reason for the trend toward distributed systems is that these systems potentially have a much better price/performance ratio than a single large centralized system would have. In effect, a distributed system gives more bang for the buck.

A slight variation on this theme is the observation that a collection of microprocessors cannot only give a better price/performance ratio than a single mainframe, but may yield an absolute performance that no mainframe can achieve at any price. For example, with current technology it is possible to build a system from 10,000 modern CPU chips, each of which runs at 50 MIPS (Millions of Instructions Per Second), for a total performance of 500,000 MIPS. For a single processor (i.e., CPU) to achieve this, it would have to execute an instruction in 0.002 nsec (2 picosec). No existing machine even comes close to this, and both theoretical and engineering considerations make it unlikely that any machine ever will. Theoretically, Einstein's theory of relativity dictates that nothing can travel faster than light, which can cover only 0.6 mm in 2 picosec. Practically, a computer of that speed fully contained in a 0.6-mm cube would generate so much heat that it would melt instantly. Thus whether the goal is normal performance at low cost or extremely high performance at greater cost, distributed systems have much to offer.

As an aside, some authors make a distinction between *distributed systems*, which are designed to allow many users to work together, and *parallel systems*, whose only goal is to achieve maximum speedup on a single problem, as our 500,000-MIPS machine might. We believe that this distinction is difficult to maintain because the design spectrum is really a continuum. We prefer to use the term "distributed system" in the broadest sense to denote any system in which multiple interconnected CPUs work together.

A next reason for building a distributed system is that some applications are inherently distributed. A supermarket chain might have many stores, each of which gets goods delivered locally (possibly from local farms), makes local sales, and makes local decisions about which vegetables are so old or rotten that they must be thrown out. It therefore makes sense to keep track of inventory at each store on a local computer rather than centrally at corporate headquarters. After all, most queries and updates will be done locally. Nevertheless, from time to time, top management may want to find out how many rutabagas it currently owns. One way to accomplish this goal is to make the complete system look like a single computer to the application programs, but implement decentrally, with one computer per store as we have described. This would then be a commercial distributed system.

Another inherently distributed system is what is often called **computer-supported cooperative work**, in which a group of people, located far from each other, are working together, for example, to produce a joint report. Given the long term trends in the computer industry, one can easily imagine a whole new area, **computer-supported cooperative games**, in which players at different locations play against each other in real time. One can imagine electronic hide-and-seek in a big multidimensional maze, and even electronic dogfights with each player using a local flight simulator to try to shoot down the other players, with each player's screen showing the view out of the player's plane, including other planes that fly within visual range.

Another potential advantage of a distributed system over a centralized system is higher reliability. By distributing the workload over many machines, a single chip failure will bring down at most one machine, leaving the rest intact. Ideally, if 5 percent of the machines are down at any moment, the system should be able to continue to work with a 5 percent loss in performance. For critical applications, such as control of nuclear reactors or aircraft, using a distributed system to achieve high reliability may be the dominant consideration.

Finally, incremental growth is also potentially a big plus. Often, a company will buy a mainframe with the intention of doing all its work on it. If the company prospers and the workload grows, at a certain point the mainframe will no longer be adequate. The only solutions are either to replace the mainframe with a larger one (if it exists) or to add a second mainframe. Both of these can wreak major havoc on the company's operations. In contrast, with a distributed system, it may be possible simply to add more processors to the system, thus allowing it to expand gradually as the need arises. These advantages are summarized in Fig. 1-1.

Item	Description
Economics	Microprocessors offer a better price/performance than mainframes
Speed	A distributed system may have more total computing power than a mainframe
Inherent distribution	Some applications involve spatially separated machines
Reliability	If one machine crashes, the system as a whole can still survive
Incremental growth	Computing power can be added in small increments

Fig. 1-1. Advantages of distributed systems over centralized systems.

In the long term, the main driving force will be the existence of large numbers of personal computers and the need for people to work together and share information in a convenient way without being bothered by geography or the physical distribution of people, data, and machines.

1.2.2. Advantages of Distributed Systems over Independent PCs

Given that microprocessors are a cost-effective way to do business, why not just give everyone his† own PC and let people work independently? For one thing, many users need to share data. For example, airline reservation clerks need access to the master data base of flights and existing reservations. Giving each clerk his own private copy of the entire data base would not work, since nobody would know which seats the other clerks had already sold. Shared data are absolutely essential to this and many other applications, so the machines must be interconnected. Interconnecting the machines leads to a distributed system.

Sharing often involves more than just data. Expensive peripherals, such as color laser printers, phototypesetters, and massive archival storage devices (e.g., optical jukeboxes), are also candidates.

A third reason to connect a group of isolated computers into a distributed system is to achieve enhanced person-to-person communication. For many people, electronic mail has numerous attractions over paper mail, telephone, and FAX. It is much faster than paper mail, does not require both parties to be available at the same time as does the telephone, and unlike FAX, produces documents that can be edited, rearranged, stored in the computer, and manipulated with text processing programs.

Finally, a distributed system is potentially more flexible than giving each user an isolated personal computer. Although one model is to give each person a personal computer and connect them all with a LAN, this is not the only possibility. Another one is to have a mixture of personal and shared computers, perhaps of different sizes, and let jobs run on the most appropriate one, rather than always on the owner's machine. In this way, the workload can be spread over the computers more effectively, and the loss of a few machines may be compensated for by letting people run their jobs elsewhere. Figure 1-2 summarizes these points.

1.2.3. Disadvantages of Distributed Systems

Although distributed systems have their strengths, they also have their weaknesses. In this section, we will point out a few of them. We have already hinted at the worst problem: software. With the current state-of-the-art, we do not have much experience in designing, implementing, and using distributed software. What kinds of operating systems, programming languages, and applications are appropriate for these systems? How much should the users know

† Please read "his" as "his or hers" throughout this book.

Item	Description
Data sharing	Allow many users access to a common data base
Device sharing	Allow many users to share expensive peripherals like color printers
Communication	Make human-to-human communication easier, for example, by electronic mail
Flexibility	Spread the workload over the available machines in the most cost effective way

Fig. 1-2. Advantages of distributed systems over isolated (personal) computers.

about the distribution? How much should the system do and how much should the users do? The experts differ (not that this is unusual with experts, but when it comes to distributed systems, they are barely on speaking terms). As more research is done, this problem will diminish, but for the moment it should not be underestimated.

A second potential problem is due to the communication network. It can lose messages, which requires special software to be able to recover, and it can become overloaded. When the network saturates, it must either be replaced or a second one must be added. In both cases, some portion of one or more buildings may have to be rewired at great expense, or network interface boards may have to be replaced (e.g., by fiber optics). Once the system comes to depend on the network, its loss or saturation can negate most of the advantages the distributed system was built to achieve.

Finally, the easy sharing of data, which we described above as an advantage, may turn out to be a two-edged sword. If people can conveniently access data all over the system, they may equally be able to conveniently access data that they have no business looking at. In other words, security is often a problem. For data that must be kept secret at all costs, it is often preferable to have a dedicated, isolated personal computer that has no network connections to any other machines, and is kept in a locked room with a secure safe in which all the floppy disks are stored. The disadvantages of distributed systems are summarized in Fig. 1-3.

Despite these potential problems, many people feel that the advantages outweigh the disadvantages, and it is expected that distributed systems will become increasingly important in the coming years. In fact, it is likely that within a few years, most organizations will connect most of their computers into large distributed systems to provide better, cheaper, and more convenient service for the users. An isolated computer in a medium-sized or large business or other organization will probably not even exist in ten years.

Item	Description
Software	Little software exists at present for distributed systems
Networking	The network can saturate or cause other problems
Security	Easy access also applies to secret data

Fig. 1-3. Disadvantages of distributed systems.

1.3. HARDWARE CONCEPTS

Even though all distributed systems consist of multiple CPUs, there are several different ways the hardware can be organized, especially in terms of how they are interconnected and how they communicate. In this section we will take a brief look at distributed system hardware, in particular, how the machines are connected together. In the next section we will examine some of the software issues related to distributed systems.

Various classification schemes for multiple CPU computer systems have been proposed over the years, but none of them have really caught on and been widely adopted. Probably the most frequently cited taxonomy is Flynn's (1972), although it is fairly rudimentary. Flynn picked two characteristics that he considered essential: the number of instruction streams and the number of data streams. A computer with a single instruction stream and a single data stream is called SISD. All traditional uniprocessor computers (i.e., those having only one CPU) fall in this category, from personal computers to large mainframes.

The next category is SIMD, single instruction stream, multiple data stream. This type refers to array processors with one instruction unit that fetches an instruction, and then commands many data units to carry it out in parallel, each with its own data. These machines are useful for computations that repeat the same calculation on many sets of data, for example, adding up all the elements of 64 independent vectors. Some supercomputers are SIMD.

The next category is MISD, multiple instruction stream, single data stream. No known computers fit this model. Finally, comes MIMD, which essentially means a group of independent computers, each with its own program counter, program, and data. All distributed systems are MIMD, so this classification system is not tremendously useful for our purposes.

Although Flynn stopped here, we will go further. In Fig. 1-4, we divide all MIMD computers into two groups: those that have shared memory, usually called **multiprocessors**, and those that do not, sometimes called **multicomputers**. The essential difference is this: in a multiprocessor, there is a single virtual address space that is shared by all CPUs. If any CPU writes, for example, the value 44 to address 1000, any other CPU subsequently reading from *its* address 1000 will get the value 44. All the machines share the same memory.

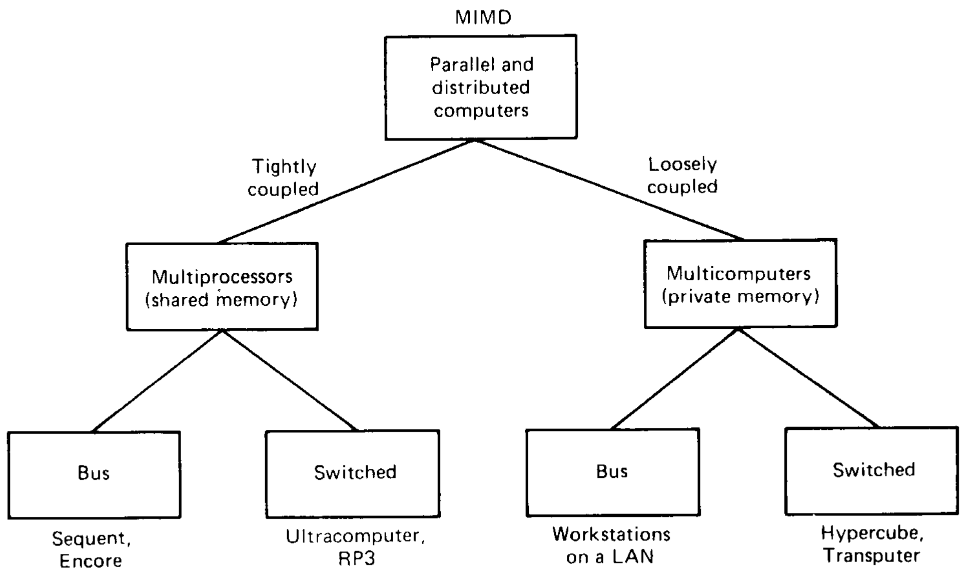

Fig. 1-4. A taxonomy of parallel and distributed computer systems.

In contrast, in a multicomputer, every machine has its own private memory. If one CPU writes the value 44 to address 1000, when another CPU reads address 1000 it will get whatever value was there before. The write of 44 does not affect *its* memory at all. A common example of a multicomputer is a collection of personal computers connected by a network.

Each of these categories can be further divided based on the architecture of the interconnection network. In Fig. 1-4 we describe these two categories as **bus** and **switched**. By bus we mean that there is a single network, backplane, bus, cable, or other medium that connects all the machines. Cable television uses a scheme like this: the cable company runs a wire down the street, and all the subscribers have taps running to it from their television sets.

Switched systems do not have a single backbone like cable television. Instead, there are individual wires from machine to machine, with many different wiring patterns in use. Messages move along the wires, with an explicit switching decision made at each step to route the message along one of the outgoing wires. The worldwide public telephone system is organized in this way.

Another dimension to our taxonomy is that in some systems the machines are **tightly coupled** and in others they are **loosely coupled**. In a tightly-coupled system, the delay experienced when a message is sent from one computer to another is short, and the data rate is high; that is, the number of bits per second that can be transferred is large. In a loosely-coupled system, the opposite is true: the intermachine message delay is large and the data rate is low. For example, two CPU chips on the same printed circuit board and connected by wires etched onto the board are likely to be tightly coupled, whereas two computers connected by a 2400 bit/sec modem over the telephone system are certain to be loosely coupled.

Tightly-coupled systems tend to be used more as parallel systems (working on a single problem) and loosely-coupled ones tend to be used as distributed systems (working on many unrelated problems), although this is not always true. One famous counterexample is a project in which hundreds of computers all over the world worked together trying to factor a huge number (about 100 digits). Each computer was assigned a different range of divisors to try, and they all worked on the problem in their spare time, reporting the results back by electronic mail when they finished.

On the whole, multiprocessors tend to be more tightly coupled than multicomputers, because they can exchange data at memory speeds, but some fiber-optic based multicomputers can also work at memory speeds. Despite the vagueness of the terms "tightly coupled" and "loosely coupled," they are useful concepts, just as saying "Jack is fat and Jill is thin" conveys information about girth even though one can get into a fair amount of discussion about the concepts of "fatness" and "thinness."

In the following four sections, we will look at the four categories of Fig. 1-4 in more detail, namely bus multiprocessors, switched multiprocessors, bus multicomputers, and switched multicomputers. Although these topics are not directly related to our main concern, distributed operating systems, they will shed some light on the subject because as we shall see, different categories of machines use different kinds of operating systems.

1.3.1. Bus-Based Multiprocessors

Bus-based multiprocessors consist of some number of CPUs all connected to a common bus, along with a memory module. A simple configuration is to have a high-speed backplane or motherboard into which CPU and memory cards can be inserted. A typical bus has 32 or 64 address lines, 32 or 64 data lines, and perhaps 32 or more control lines, all of which operate in parallel. To read a word of memory, a CPU puts the address of the word it wants on the bus address lines, then puts a signal on the appropriate control lines to indicate that it wants to read. The memory responds by putting the value of the word on the data lines to allow the requesting CPU to read it in. Writes work in a similar way.

Since there is only one memory, if CPU *A* writes a word to memory and then CPU *B* reads that word back a microsecond later, *B* will get the value just written. A memory that has this property is said to be **coherent**. Coherence plays an important role in distributed operating systems in a variety of ways that we will study later.

The problem with this scheme is that with as few as 4 or 5 CPUs, the bus will usually be overloaded and performance will drop drastically. The solution is to add a high-speed **cache memory** between the CPU and the bus, as shown in Fig. 1-5. The cache holds the most recently accessed words. All memory requests go through the cache. If the word requested is in the cache, the cache itself responds to the CPU, and no bus request is made. If the cache is large enough, the probability of success, called the **hit rate**, will be high, and the amount of bus traffic per CPU will drop dramatically, allowing many more CPUs in the system. Cache sizes of 64K to 1M are common, which often gives a hit rate of 90 percent or more.

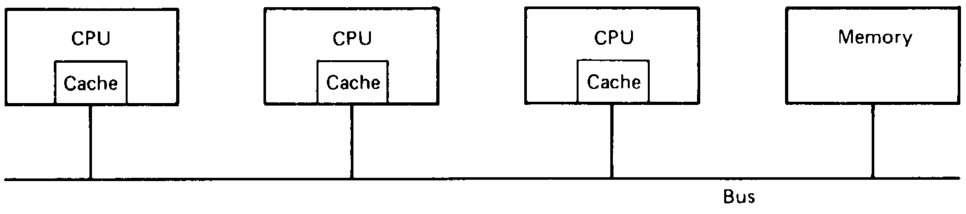

Fig. 1-5. A bus-based multiprocessor.

However, the introduction of caches also brings a serious problem with it. Suppose that two CPUs, *A* and *B*, each read the same word into their respective caches. Then *A* overwrites the word. When *B* next reads that word, it gets the old value from its cache, not the value *A* just wrote. The memory is now incoherent, and the system is difficult to program.

Many researchers have studied this problem, and various solutions are known. Below we will sketch one of them. Suppose that the cache memories are designed so that whenever a word is written to the cache, it is written through to memory as well. Such a cache is, not surprisingly, called a **write-through cache.** In this design, cache hits for reads do not cause bus traffic, but cache misses for reads, and all writes, hits and misses, cause bus traffic.

In addition, all caches constantly monitor the bus. Whenever a cache sees a write occurring to a memory address present in its cache, it either removes that entry from its cache, or updates the cache entry with the new value. Such a cache is called a **snoopy cache** (or sometimes, a **snooping cache**) because it is always "snooping" (eavesdropping) on the bus. A design consisting of snoopy write-through caches is coherent and is invisible to the programmer. Nearly all bus-based multiprocessors use either this architecture or one closely related to it.

Using it, it is possible to put about 32 or possibly 64 CPUs on a single bus. For more about bus-based multiprocessors, see Lilja (1993).

1.3.2. Switched Multiprocessors

To build a multiprocessor with more than 64 processors, a different method is needed to connect the CPUs with the memory. One possibility is to divide the memory up into modules and connect them to the CPUs with a **crossbar switch**, as shown in Fig. 1-6(a). Each CPU and each memory has a connection coming out of it, as shown. At every intersection is a tiny electronic **crosspoint switch** that can be opened and closed in hardware. When a CPU wants to access a particular memory, the crosspoint switch connecting them is closed momentarily, to allow the access to take place. The virtue of the crossbar switch is that many CPUs can be accessing memory at the same time, although if two CPUs try to access the same memory simultaneously, one of them will have to wait.

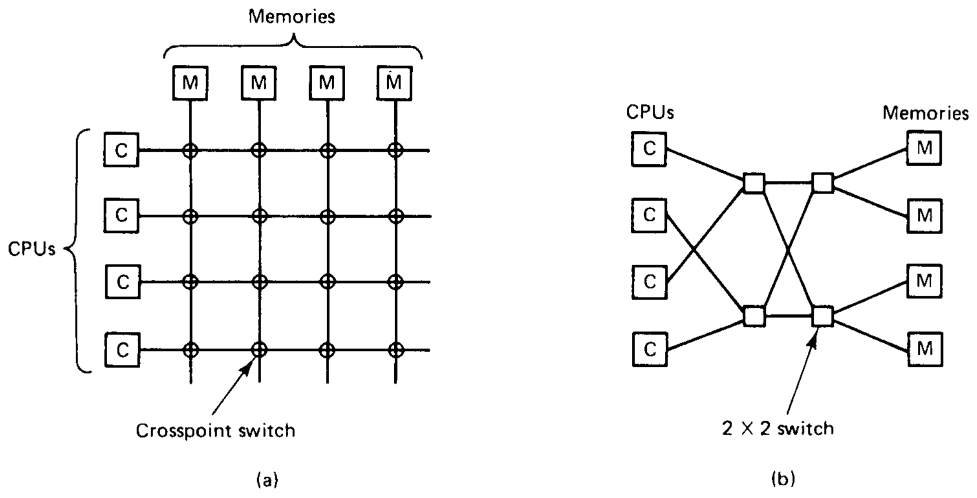

Fig. 1-6. (a) A crossbar switch. (b) An omega switching network.

The downside of the crossbar switch is that with n CPUs and n memories, n^2 crosspoint switches are needed. For large n, this number can be prohibitive. As a result, people have looked for, and found, alternative switching networks that require fewer switches. The **omega network** of Fig. 1-6(b) is one example. This network contains four 2×2 switches, each having two inputs and two outputs. Each switch can route either input to either output. A careful look at the figure will show that with proper settings of the switches, every CPU can access every memory. These switches can be set in nanoseconds or less.

In the general case, with n CPUs and n memories, the omega network requires $\log_2 n$ switching stages, each containing $n/2$ switches, for a total of $(n \log_2 n)/2$ switches. Although for large n this is much better than n^2, it is still substantial.

Furthermore, there is another problem: delay. For example, for $n = 1024$, there are 10 switching stages from the CPU to the memory, and another 10 for the word requested to come back. Suppose that the CPU is a modern RISC chip running at 100 MIPS; that is, the instruction execution time is 10 nsec. If a memory request is to traverse a total of 20 switching stages (10 outbound and 10 back) in 10 nsec, the switching time must be 500 picosec (0.5 nsec). The complete multiprocessor will need 5120 500-picosec switches. This is not going to be cheap.

People have attempted to reduce the cost by going to hierarchical systems. Some memory is associated with each CPU. Each CPU can access its own local memory quickly, but accessing anybody else's memory is slower. This design gives rise to what is known as a **NUMA (NonUniform Memory Access)** machine. Although NUMA machines have better average access times than machines based on omega networks, they have the new complication that the placement of the programs and data becomes critical in order to make most access go to the local memory.

To summarize, bus-based multiprocessors, even with snoopy caches, are limited by the amount of bus capacity to about 64 CPUs at most. To go beyond that requires a switching network, such as a crossbar switch, an omega switching network, or something similar. Large crossbar switches are very expensive, and large omega networks are both expensive and slow. NUMA machines require complex algorithms for good software placement. The conclusion is clear: building a large, tightly-coupled, shared memory multiprocessor is possible, but is difficult and expensive.

1.3.3. Bus-Based Multicomputers

On the other hand, building a multicomputer (i.e., no shared memory) is easy. Each CPU has a direct connection to its own local memory. The only problem left is how the CPUs communicate with each other. Clearly, some interconnection scheme is needed here, too, but since it is only for CPU-to-CPU communication, the volume of traffic will be several orders of magnitude lower than when the interconnection network is also used for CPU-to-memory traffic.

In Fig. 1-7 we see a bus-based multicomputer. It looks topologically similar to the bus-based multiprocessor, but since there will be much less traffic over it, it need not be a high-speed backplane bus. In fact, it can be a much lower speed LAN (typically, 10–100 Mbps, compared to 300 Mbps and up for a backplane bus). Thus Fig. 1-7 is more often a collection of workstations on a LAN than a collection of CPU cards inserted into a fast bus (although the latter configuration is definitely a possible design).

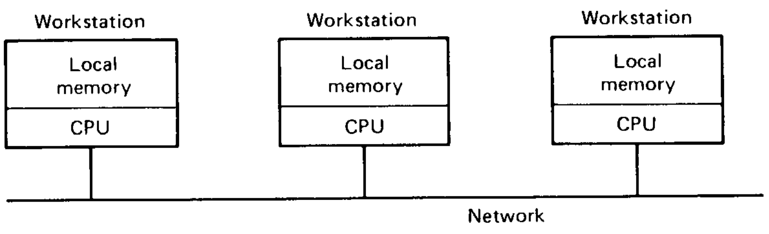

Fig. 1-7. A multicomputer consisting of workstations on a LAN.

1.3.4. Switched Multicomputers

Our last category consists of switched multicomputers. Various interconnection networks have been proposed and built, but all have the property that each CPU has direct and exclusive access to its own, private memory. Figure 1-8 shows two popular topologies, a grid and a hypercube. Grids are easy to understand and lay out on printed circuit boards. They are best suited to problems that have an inherent two-dimensional nature, such as graph theory or vision (e.g., robot eyes or analyzing photographs).

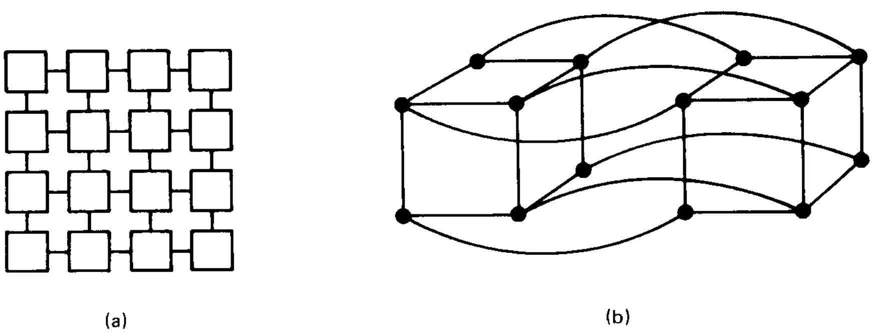

Fig. 1-8. (a) Grid. (b) Hypercube.

A **hypercube** is an *n*-dimensional cube. The hypercube of Fig. 1-8(b) is four-dimensional. It can be thought of as two ordinary cubes, each with 8 vertices and 12 edges. Each vertex is a CPU. Each edge is a connection between two CPUs. The corresponding vertices in each of the two cubes are connected.

To expand the hypercube to five dimensions, we would add another set of two interconnected cubes to the figure, connect the corresponding edges in the two halves, and so on. For an n-dimensional hypercube, each CPU has n connections to other CPUs. Thus the complexity of the wiring increases only logarithmically with the size. Since only nearest neighbors are connected, many messages have to make several hops to reach their destination. However, the longest possible path also grows logarithmically with the size, in contrast to the grid, where it grows as the square root of the number of CPUs. Hypercubes with 1024 CPUs have been commercially available for several years, and hypercubes with as many as 16,384 CPUs are starting to become available.

1.4. SOFTWARE CONCEPTS

Although the hardware is important, the software is even more important. The image that a system presents to its users, and how they think about the system, is largely determined by the operating system software, not the hardware. In this section we will introduce the various types of operating systems for the multiprocessors and multicomputers we have just studied, and discuss which kind of software goes with which kind of hardware.

Operating systems cannot be put into nice, neat pigeonholes like hardware. By nature software is vague and amorphous. Still, it is more-or-less possible to distinguish two kinds of operating systems for multiple CPU systems: loosely coupled and tightly coupled. As we shall see, loosely and tightly-coupled software is roughly analogous to loosely and tightly-coupled hardware.

Loosely-coupled software allows machines and users of a distributed system to be fundamentally independent of one another, but still to interact to a limited degree where that is necessary. Consider a group of personal computers, each of which has its own CPU, its own memory, its own hard disk, and its own operating system, but which share some resources, such as laser printers and data bases, over a LAN. This system is loosely coupled, since the individual machines are clearly distinguishable, each with its own job to do. If the network should go down for some reason, the individual machines can still continue to run to a considerable degree, although some functionality may be lost (e.g., the ability to print files).

To show how difficult it is to make definitions in this area, now consider the same system as above, but without the network. To print a file, the user writes the file on a floppy disk, carries it to the machine with the printer, reads it in, and then prints it. Is this still a distributed system, only now even more loosely coupled? It's hard to say. From a fundamental point of view, there is not really any theoretical difference between communicating over a LAN and communicating by carrying floppy disks around. At most one can say that the delay and data rate are worse in the second example.

At the other extreme we might find a multiprocessor dedicated to running a single chess program in parallel. Each CPU is assigned a board to evaluate, and it spends its time examining that board and all the boards that can be generated from it. When the evaluation is finished, the CPU reports back the results and is given a new board to work on. The software for this system, both the application program and the operating system required to support it, is clearly much more tightly coupled than in our previous example.

We have now seen four kinds of distributed hardware and two kinds of distributed software. In theory, there should be eight combinations of hardware and software. In fact, only four are worth distinguishing, because to the user, the interconnection technology is not visible. For most purposes, a multiprocessor is a multiprocessor, whether it uses a bus with snoopy caches or uses an omega network. In the following sections we will look at some of the most common combinations of hardware and software.

1.4.1. Network Operating Systems

Let us start with loosely-coupled software on loosely-coupled hardware, since this is probably the most common combination at many organizations. A typical example is a network of workstations connected by a LAN. In this model, each user has a workstation for his exclusive use. It may or may not have a hard disk. It definitely has its own operating system. All commands are normally run locally, right on the workstation.

However, it is sometimes possible for a user to log into another workstation remotely by using a command such as

```
rlogin machine
```

The effect of this command is to turn the user's own workstation into a remote terminal logged into the remote machine. Commands typed on the keyboard are sent to the remote machine, and output from the remote machine is displayed on the screen. To switch to a different remote machine, it is necessary first to log out, then to use the *rlogin* command to connect to another machine. At any instant, only one machine can be used, and the selection of the machine is entirely manual.

Networks of workstations often also have a remote copy command to copy files from one machine to another. For example, a command like

```
rcp machine1:file1 machine2:file2
```

might copy the file *file1* from *machine1* to *machine2* and give it the name *file2* there. Again here, the movement of files is explicit and requires the user to be completely aware of where all files are located and where all commands are being executed.

While better than nothing, this form of communication is extremely primitive and has led system designers to search for more convenient forms of communication and information sharing. One approach is to provide a shared, global file system accessible from all the workstations. The file system is supported by one or more machines called **file servers**. The file servers accept requests from user programs running on the other (nonserver) machines, called **clients**, to read and write files. Each incoming request is examined and executed, and the reply is sent back, as illustrated in Fig. 1-9.

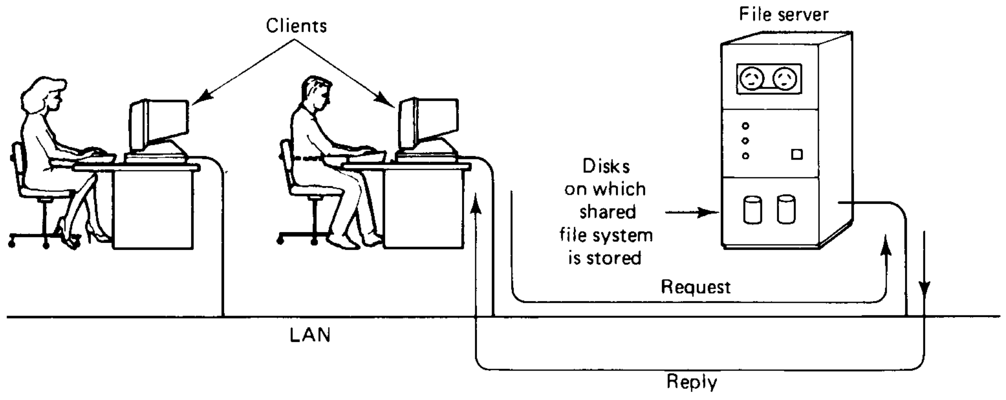

Fig. 1-9. Two clients and a server in a network operating system.

File servers generally maintain hierarchical file systems, each with a root directory containing subdirectories and files. Workstations can import or mount these file systems, augmenting their local file systems with those located on the servers. For example, in Fig. 1-10, two file servers are shown. One has a directory called *games*, while the other has a directory called *work*. These directories each contain several files. Both of the clients shown have mounted both of the servers, but they have mounted them in different places in their respective file systems. Client 1 has mounted them in its root directory, and can access them as */games* and */work*, respectively. Client 2, like client 1, has mounted *games* in its root directory, but regarding the reading of mail and news as a kind of game, has created a directory */games/work* and mounted *work* there. Consequently, it can access *news* using the path */games/work/news* rather than */work/news*.

While it does not matter where a client mounts a server in its directory hierarchy, it is important to notice that different clients can have a different view of the file system. The name of a file depends on where it is being accessed from, and how that machine has set up its file system. Because each workstation operates relatively independently of the others, there is no guarantee that they all present the same directory hierarchy to their programs.

The operating system that is used in this kind of environment must manage the individual workstations and file servers and take care of the communication

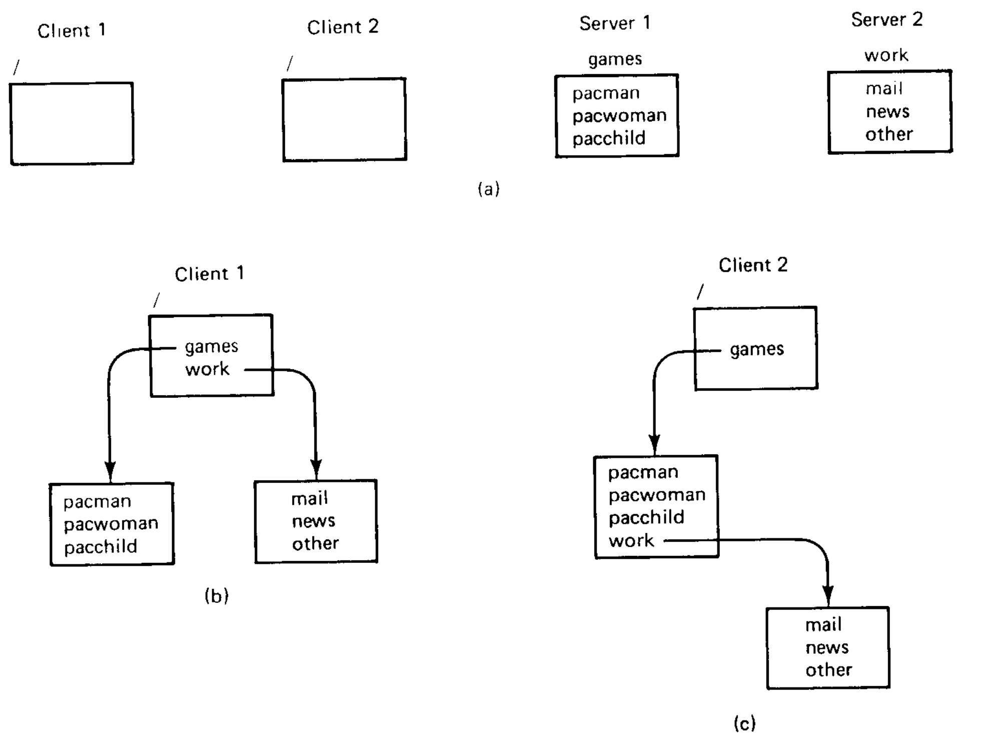

Fig. 1-10. Different clients may mount the servers in different places.

between them. It is possible that the machines all run the same operating system, but this is not required. If the clients and servers run on different systems, as a bare minimum they must agree on the format and meaning of all the messages that they may potentially exchange. In a situation like this, where each machine has a high degree of autonomy and there are few system-wide requirements, people usually speak of a **network operating system**.

1.4.2. True Distributed Systems

Network operating systems are loosely-coupled software on loosely-coupled hardware. Other than the shared file system, it is quite apparent to the users that such a system consists of numerous computers. Each can run its own operating system and do whatever its owner wants. There is essentially no coordination at all, except for the rule that client-server traffic must obey the system's protocols.

The next evolutionary step beyond this is tightly-coupled software on the same loosely-coupled (i.e., multicomputer) hardware. The goal of such a system is to create the illusion in the minds of the users that the entire network of computers is a single timesharing system, rather than a collection of distinct machines. Some authors refer to this property as the **single-system image**. Others put it slightly differently, saying that a distributed system is one that runs on a collection of networked machines but acts like a **virtual uniprocessor**. No matter how it is expressed, the essential idea is that the users should not have to be aware of the existence of multiple CPUs in the system. No current system fulfills this requirement entirely, but a number of candidates are on the horizon. These will be discussed later in the book.

What are some characteristics of a distributed system? To start with, there must be a single, global interprocess communication mechanism so that any process can talk to any other process. It will not do to have different mechanisms on different machines or different mechanisms for local communication and remote communication. There must also be a global protection scheme. Mixing access control lists, the UNIX® protection bits, and capabilities will not give a single system image.

Process management must also be the same everywhere. How processes are created, destroyed, started, and stopped must not vary from machine to machine. In short, the idea behind network operating systems, namely that any machine can do whatever it wants to as long as it obeys the standard protocols when engaging in client-server communication, is not enough. Not only must there be a single set of system calls available on all machines, but these calls must be designed so that they make sense in a distributed environment.

The file system must look the same everywhere, too. Having file names restricted to 11 characters in some locations and being unrestricted in others is undesirable. Also, every file should be visible at every location, subject to protection and security constraints, of course.

As a logical consequence of having the same system call interface everywhere, it is normal that identical kernels run on all the CPUs in the system. Doing so makes it easier to coordinate activities that must be global. For example, when a process has to be started up, all the kernels have to cooperate in finding the best place to execute it. In addition, a global file system is needed.

Nevertheless, each kernel can have considerable control over its own local resources. For example, since there is no shared memory, it is logical to allow each kernel to manage its own memory. For example, if swapping or paging is used, the kernel on each CPU is the logical place to determine what to swap or page. There is no reason to centralize this authority. Similarly, if multiple processes are running on some CPU, it makes sense to do the scheduling right there, too.

A considerable body of knowledge is now available about designing and implementing distributed operating systems. Rather than going into these issues here, we will first finish off our survey of the different combinations of hardware and software, and come back to them in Sec. 1.5.

1.4.3. Multiprocessor Timesharing Systems

The last combination we wish to discuss is tightly-coupled software on tightly-coupled hardware. While various special-purpose machines exist in this category (such as dedicated data base machines), the most common general-purpose examples are multiprocessors that are operated as a UNIX timesharing system, but with multiple CPUs instead of one CPU. To the outside world, a multiprocessor with 32 30-MIPS CPUs acts very much like a single 960-MIPS CPU (this is the single-system image discussed above). Except that implementing it on a multiprocessor makes life much easier, since the entire design can be centralized.

The key characteristic of this class of system is the existence of a single run queue: a list of all the processes in the system that are logically unblocked and ready to run. The run queue is a data structure kept in the shared memory. As an example, consider the system of Fig. 1-11, which has three CPUs and five processes that are ready to run. All five processes are located in the shared memory, and three of them are currently executing: process *A* on CPU 1, process *B* on CPU 2, and process *C* on CPU 3. The other two processes, *D* and *E*, are also in memory, waiting their turn.

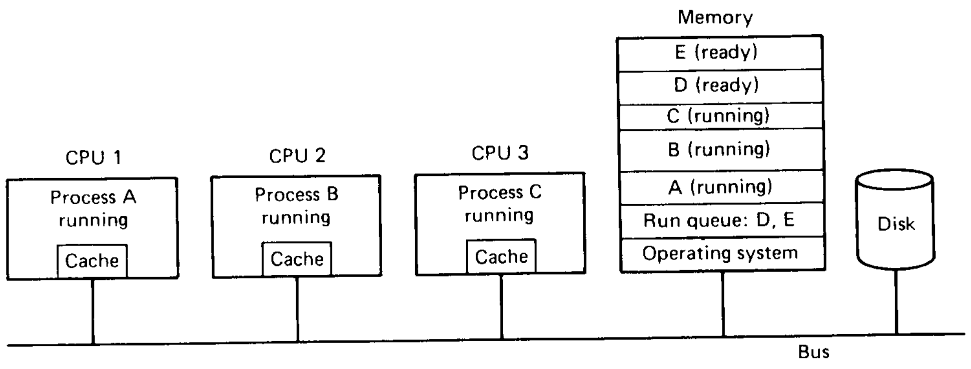

Fig. 1-11. A multiprocessor with a single run queue.

Now suppose that process *B* blocks waiting for I/O or its quantum runs out. Either way, CPU 2 must suspend it, and find another process to run. CPU 2 will normally begin executing operating system code (located in the shared memory). After having saved all of *B*'s registers, it will enter a critical region to run the scheduler to look for another process to run. It is essential that the scheduler be run as a critical region to prevent two CPUs from choosing the same process to run next. The necessary mutual exclusion can be achieved by using monitors, semaphores, or any other standard construction used in singleprocessor systems.

Once CPU 2 has gained exclusive access to the run queue, it can remove the first entry, *D*, exit from the critical region, and begin executing *D*. Initially, execution will be slow, since CPU 2's cache is full of words belonging to that part of the shared memory containing process *B*, but after a little while, these will have been purged and the cache will be full of *D*'s code and data, so execution will speed up.

Because none of the CPUs have local memory and all programs are stored in the global shared memory, it does not matter on which CPU a process runs. If a long-running process is scheduled many times before it completes, on the average, it will spend about the same amount of time running on each CPU. The only factor that has any effect at all on CPU choice is the slight gain in performance when a process starts up on a CPU that is currently caching part of its address space. In other words, if all CPUs are idle, waiting for I/O, and one process becomes ready, it is slightly preferable to allocate it to the CPU it was last using, assuming that no other process has used that CPU since (Vaswani and Zahorjan, 1991).

As an aside, if a process blocks for I/O on a multiprocessor, the operating system has the choice of suspending it or just letting it do busy waiting. If most I/O is completed in less time than it takes to do a process switch, busy waiting is preferable. Some systems let the process keep its processor for a few milliseconds, in the hope that the I/O will complete soon, but if that does not occur before the timer runs out, a process switch is made (Karlin et al., 1991). If most critical regions are short, this approachcan avoid many expensive process switches.

An area in which this kind of multiprocessor differs appreciably from a network or distributed system is in the organization of the file system. The operating system normally contains a traditional file system, including a single, unified block cache. When any process executes a system call, a trap is made to the operating system, which carries it out, using semaphores, monitors, or something equivalent, to lock out other CPUs while critical sections are being executed or central tables are being accessed. In this way, when a WRITE system call is done, the central block cache is locked, the new data entered into the cache, and the lock released. Any subsequent READ call will see the new data, just as on a single-processor system. On the whole, the file system is hardly different from a single-processor file system. In fact, on some multiprocessors, one of the CPUs is dedicated to running the operating system; the other ones run user programs. This situation is undesirable, however, as the operating system machine is often a bottleneck. This point is discussed in detail by Boykin and Langerman (1990).

It should be clear that the methods used on the multiprocessor to achieve the appearance of a virtual uniprocessor are not applicable to machines that do not have shared memory. Centralized run queues and block only caches work when all CPUs have access to them with very low delay. Although these data structures could be simulated on a network of machines, the communication costs make this approach prohibitively expensive.

Figure 1-12 shows some of the differences between the three kinds of systems we have examined above.

Item	Network operating system	Distributed operating system	Multiprocessor operating system
Does it look like a virtual uniprocessor?	No	Yes	Yes
Do all have to run the same operating system?	No	Yes	Yes
How many copies of the operating system are there?	N	N	1
How is communication achieved?	Shared files	Messages	Shared memory
Are agreed upon network protocols required?	Yes	Yes	No
Is there a single run queue?	No	No	Yes
Does file sharing have well-defined semantics?	Usually no	Yes	Yes

Fig. 1-12. Comparison of three different ways of organizing *n* CPUs.

1.5. DESIGN ISSUES

In the preceding sections we have looked at distributed systems and related topics from both the hardware and software points of view. In the remainder of this chapter we will briefly look at some of the key design issues that people contemplating building a distributed operating system must deal with. We will come back to them in more detail later in the book.

1.5.1. Transparency

Probably the single most important issue is how to achieve the single-system image. In other words, how do the system designers fool everyone into thinking that the collection of machines is simply an old-fashioned timesharing system? A system that realizes this goal is often said to be **transparent**.

Transparency can be achieved at two different levels. Easiest to do is to hide the distribution from the users. For example, when a UNIX user types *make* to recompile a large number of files in a directory, he need not be told that all the compilations are proceeding in parallel on different machines and are using a variety of file servers to do it. To him, the only thing that is unusual is that the performance of the system is halfway decent for a change. In terms of commands issued from the terminal and results displayed on the terminal, the distributed system can be made to look just like a single-processor system.

At a lower level, it is also possible, but harder, to make the system look transparent to programs. In other words, the system call interface can be designed so that the existence of multiple processors is not visible. Pulling the wool over the programmer's eyes is harder than pulling the wool over the terminal user's eyes, however.

What does transparency really mean? It is one of those slippery concepts that sounds reasonable but is more subtle than it at first appears. As an example, imagine a distributed system consisting of workstations each running some standard operating system. Normally, system services (e.g., reading files) are obtained by issuing a system call that traps to the kernel. In such a system, remote files should be accessed the same way. A system in which remote files are accessed by explicitly setting up a network connection to a remote server and then sending messages to it is not transparent because remote services are then being accessed differently than local ones. The programmer can tell that multiple machines are involved, and this is not allowed.

The concept of transparency can be applied to several aspects of a distributed system, as shown in Fig. 1-13. **Location transparency** refers to the fact that in a true distributed system, users cannot tell where hardware and software resources such as CPUs, printers, files, and data bases are located. The name of the resource must not secretly encode the location of the resource, so names like *machine1:prog.c* or */machine1/prog.c* are not acceptable.

Kind	Meaning
Location transparency	The users cannot tell where resources are located
Migration transparency	Resources can move at will without changing their names
Replication transparency	The users cannot tell how many copies exist
Concurrency transparency	Multiple users can share resources automatically
Parallelism transparency	Activities can happen in parallel without users knowing

Fig. 1-13. Different kinds of transparency in a distributed system.

Migration transparency means that resources must be free to move from one location to another without having their names change. In the example of

Fig. 1-10 we saw how server directories could be mounted in arbitrary places in the clients' directory hierarchy. Since a path like */work/news* does not reveal the location of the server, it is location transparent. However, now suppose that the folks running the servers decide that reading network news really falls in the category "games" rather than in the category "work." Accordingly, they move *news* from server 2 to server 1. The next time client 1 boots and mounts the servers in his customary way, he will notice that */work/news* no longer exists. Instead, there is a new entry, */games/news*. Thus the mere fact that a file or directory has migrated from one server to another has forced it to acquire a new name because the system of remote mounts is not migration transparent.

If a distributed system has **replication transparency**, the operating system is free to make additional copies of files and other resources on its own without the users noticing. Clearly, in the previous example, automatic replication is impossible because the names and locations are so closely tied together. To see how replication transparency might be achievable, consider a collection of *n* servers logically connected to form a ring. Each server maintains the entire directory tree structure but holds only a subset of the files themselves. To read a file, a client sends a message containing the full path name to any of the servers. That server checks to see if it has the file. If so, it returns the data requested. If not, it forwards the request to the next server in the ring, which then repeats the algorithm. In this system, the servers can decide by themselves to replicate any file on any or all servers, without the users having to know about it. Such a scheme is replication transparent because it allows the system to make copies of heavily used files without the users even being aware that this is happening.

Distributed systems usually have multiple, independent users. What should the system do when two or more users try to access the same resource at the same time? For example, what happens if two users try to update the same file at the same time? If the system is **concurrency transparent**, the users will not notice the existence of other users. One mechanism for achieving this form of transparency would be for the system to lock a resource automatically once someone had started to use it, unlocking it only when the access was finished. In this manner, all resources would only be accessed sequentially, never concurrently.

Finally, we come to the hardest one, **parallelism transparency**. In principle, a distributed system is supposed to appear to the users as a traditional, uniprocessor timesharing system. What happens if a programmer knows that his distributed system has 1000 CPUs and he wants to use a substantial fraction of them for a chess program that evaluates boards in parallel? The theoretical answer is that together the compiler, runtime system, and operating system should be able to figure out how to take advantage of this potential parallelism without the programmer even knowing it. Unfortunately, the current state-of-the-art is nowhere near allowing this to happen. Programmers who actually want to use multiple CPUs for a single problem will have to program this explicitly, at least for the foreseeable future. Parallelism transparency can be regarded as the holy grail for distributed systems designers. When that has been achieved, the work will have been completed, and it will be time to move on to new fields.

All this notwithstanding, there are times when users do *not* want complete transparency. For example, when a user asks to print a document, he often prefers to have the output appear on the local printer, not one 1000 km away, even if the distant printer is fast, inexpensive, can handle color and smell, and is currently idle.

1.5.2. Flexibility

The second key design issue is flexibility. It is important that the system be flexible because we are just beginning to learn about how to build distributed systems. It is likely that this process will incur many false starts and considerable backtracking. Design decisions that now seem reasonable may later prove to be wrong. The best way to avoid problems is thus to keep one's options open.

Flexibility, along with transparency, is like parenthood and apple pie: who could possibly be against them? It is hard to imagine anyone arguing in favor of an inflexible system. However, things are not as simple as they seem. There are two schools of thought concerning the structure of distributed systems. One school maintains that each machine should run a traditional kernel that provides most services itself. The other maintains that the kernel should provide as little as possible, with the bulk of the operating system services available from user-level servers. These two models, known as the monolithic kernel and microkernel, respectively, are illustrated in Fig. 1-14.

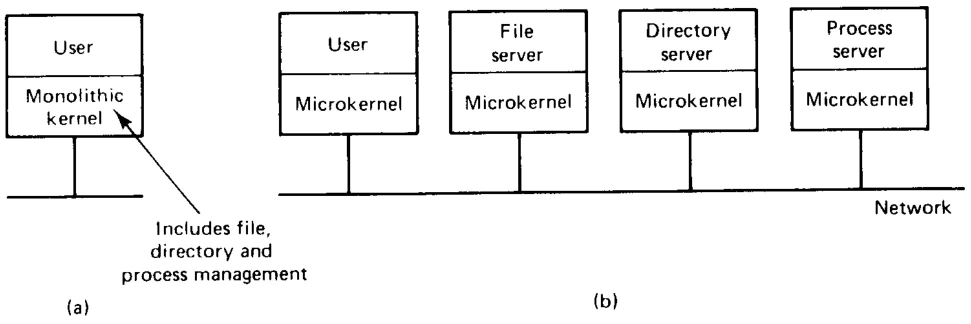

Fig. 1-14. (a) Monolithic kernel. (b) Microkernel.

The monolithic kernel is basically today's centralized operating system augmented with networking facilities and the integration of remote services. Most system calls are made by trapping to the kernel, having the work performed there, and having the kernel return the desired result to the user process. With this approach, most machines have disks and manage their own local file systems. Many distributed systems that are extensions or imitations of UNIX use this approach because UNIX itself has a large, monolithic kernel.

If the monolithic kernel is the reigning champion, the microkernel is the up-and-coming challenger. Most distributed systems that have been designed from scratch use this method. The microkernel is more flexible because it does almost nothing. It basically provides just four minimal services:

1. An interprocess communication mechanism.
2. Some memory management.
3. A small amount of low-level process management and scheduling.
4. Low-level input/output.

In particular, unlike the monolithic kernel, it does not provide the file system, directory system, full process management, or much system call handling. The services that the microkernel does provide are included because they are difficult or expensive to provide anywhere else. The goal is to keep it small.

All the other operating system services are generally implemented as user-level servers. To look up a name, read a file, or obtain some other service, the user sends a message to the appropriate server, which then does the work and returns the result. The advantage of this method is that it is highly modular: there is a well-defined interface to each service (the set of messages the server understands), and every service is equally accessible to every client, independent of location. In addition, it is easy to implement, install, and debug new services, since adding or changing a service does not require stopping the system and booting a new kernel, as is the case with a monolithic kernel. It is precisely this ability to add, delete, and modify services that gives the microkernel its flexibility. Furthermore, users who are not satisfied with any of the official services are free to write their own.

As a simple example of this power, it is possible to have a distributed system with multiple file servers, one supporting MS-DOS file service and another supporting UNIX file service. Individual programs can use either or both, if they choose. In contrast, with a monolithic kernel, the file system is built into the kernel, and users have no choice but to use it.

The only potential advantage of the monolithic kernel is performance. Trapping to the kernel and doing everything there may well be faster than sending messages to remote servers. However, a detailed comparison of two distributed operating systems, one with a monolithic kernel (Sprite), and one with a microkernel (Amoeba), has shown that in practice this advantage is nonexistent

(Douglis et al., 1991). Other factors tend to dominate, and the small amount of time required to send a message and get a reply (typically, about 1 msec) is usually negligible. As a consequence, it is likely that microkernel systems will gradually come to dominate the distributed systems scheme, and monolithic kernels will eventually vanish or evolve into microkernels. Perhaps future editions of Silberschatz and Galvin's book on operating systems (1994) will feature hummingbirds and swifts on the cover instead of stegasauruses and triceratopses.

1.5.3. Reliability

One of the original goals of building distributed systems was to make them more reliable than single-processor systems. The idea is that if a machine goes down, some other machine takes over the job. In other words, theoretically the overall system reliability could be the Boolean OR of the component reliabilities. For example, with four file servers, each with a 0.95 chance of being up at any instant, the probability of all four being down simultaneously is $0.05^4 = 0.000006$, so the probability of at least one being available is 0.999994, far better than that of any individual server.

That is the theory. The practice is that to function at all, current distributed systems count on a number of specific servers being up. As a result, some of them have an availability more closely related to the Boolean AND of the components than to the Boolean OR. In a widely-quoted remark, Leslie Lamport once defined a distributed system as "one on which I cannot get any work done because some machine I have never heard of has crashed." While this remark was (presumably) made somewhat tongue-in-cheek, there is clearly room for improvement here.

It is important to distinguish various aspects of reliability. **Availability**, as we have just seen, refers to the fraction of time that the system is usable. Lamport's system apparently did not score well in that regard. Availability can be enhanced by a design that does not require the simultaneous functioning of a substantial number of critical components. Another tool for improving availability is redundancy: key pieces of hardware and software should be replicated, so that if one of them fails the others will be able to take up the slack.

A highly reliable system must be highly available, but that is not enough. Data entrusted to the system must not be lost or garbled in any way, and if files are stored redundantly on multiple servers, all the copies must be kept consistent. In general, the more copies that are kept, the better the availability, but the greater the chance that they will be inconsistent, especially if updates are frequent. The designers of all distributed systems must keep this dilemma in mind all the time.

Another aspect of overall reliability is security. Files and other resources must be protected from unauthorized usage. Although the same issue occurs in single-processor systems, in distributed systems it is more severe. In a single-processor system, the user logs in and is authenticated. From then on, the system knows who the user is and can check whether each attempted access is legal. In a distributed system, when a message comes in to a server asking for something, the server has no simple way of determining who it is from. No name or identification field in the message can be trusted, since the sender may be lying. At the very least, considerable care is required here.

Still another issue relating to reliability is **fault tolerance**. Suppose that a server crashes and then quickly reboots. What happens? Does the server crash bring users down with it? If the server has tables containing important information about ongoing activities, recovery will be difficult at best.

In general, distributed systems can be designed to mask failures, that is, to hide them from the users. If a file service or other service is actually constructed from a group of closely cooperating servers, it should be possible to construct it in such a way that users do not notice the loss of one or two servers, other than some performance degradation. Of course, the trick is to arrange this cooperation so that it does not add substantial overhead to the system in the normal case, when everything is functioning correctly.

1.5.4. Performance

Always lurking in the background is the issue of performance. Building a transparent, flexible, reliable distributed system will not win you any prizes if it is as slow as molasses. In particular, when running a particular application on a distributed system, it should not be appreciably worse than running the same application on a single processor. Unfortunately, achieving this is easier said than done.

Various performance metrics can be used. Response time is one, but so are throughput (number of jobs per hour), system utilization, and amount of network capacity consumed. Furthermore, the results of any benchmark are often highly dependent on the nature of the benchmark. A benchmark that involves a large number of independent highly CPU-bound computations may give radically different results from a benchmark that consists of scanning a single large file for some pattern.

The performance problem is compounded by the fact that communication, which is essential in a distributed system (and absent in a single-processor system) is typically quite slow. Sending a message and getting a reply over a LAN takes about 1 msec. Most of this time is due to unavoidable protocol handling on both ends, rather than the time the bits spend on the wire. Thus to optimize performance, one often has to minimize the number of messages. The difficulty with this strategy is that the best way to gain performance is to have many activities running in parallel on different processors, but doing so requires sending many messages. (Another solution is to do all the work on one machine, but that is hardly appropriate in a distributed system.)

One possible way out is to pay considerable attention to the **grain size** of all computations. Starting up a small computation remotely, such as adding two integers, is rarely worth it, because the communication overhead dwarfs the extra CPU cycles gained. On the other hand, starting up a long compute-bound job remotely may be worth the trouble. In general, jobs that involve a large number of small computations, especially ones that interact highly with one another, may cause trouble on a distributed system with relatively slow communication. Such jobs are said to exhibit **fine-grained parallelism**. On the other hand, jobs that involve large computations, low interaction rates, and little data, that is, **coarse-grained parallelism**, may be a better fit.

Fault tolerance also exacts its price. Good reliability is often best achieved by having several servers closely cooperating on a single request. For example, when a request comes in to a server, it could immediately send a copy of the message to one of its colleagues so that if it crashes before finishing, the colleague can take over. Naturally, when it is done, it must inform the colleague that the work has been completed, which takes another message. Thus we have at least two extra messages, which in the normal case cost time and network capacity and produce no tangible gain.

1.5.5. Scalability

Most current distributed systems are designed to work with a few hundred CPUs. It is possible that future systems will be orders of magnitude larger, and solutions that work well for 200 machines will fail miserably for 200,000,000. Consider the following. The French PTT (Post, Telephone and Telegraph administration) is in the process of installing a terminal in every household and business in France. The terminal, known as a **minitel**, will allow online access to a data base containing all the telephone numbers in France, thus eliminating the need for printing and distributing expensive telephone books. It will also vastly reduce the need for information operators who do nothing but give out telephone numbers all day. It has been calculated that the system will pay for itself within a few years. If the system works in France, other countries will inevitably adopt similar systems.

Once all the terminals are in place, the possibility of also using them for electronic mail (especially in conjunction with printers) is clearly present. Since postal services lose a huge amount of money in every country in the world, and telephone services are enormously profitable, there are great incentives to having electronic mail replace paper mail.

Next comes interactive access to all kinds of data bases and services, from electronic banking to reserving places in planes, trains, hotels, theaters, and restaurants, to name just a few. Before long, we have a distributed system with tens of millions of users. The question is: Will the methods we are currently developing scale to such large systems?

Although little is known about such huge distributed systems, one guiding principle is clear: avoid centralized components, tables, and algorithms (see Fig. 1-15). Having a single mail server for 50 million users would not be a good idea. Even if it had enough CPU and storage capacity, the network capacity into and out of it would surely be a problem. Furthermore, the system would not tolerate faults well. A single power outage could bring the entire system down. Finally, most mail is local. Having a message sent by a user in Marseille to another user two blocks away pass through a machine in Paris is not the way to go.

Concept	Example
Centralized components	A single mail server for all users
Centralized tables	A single on-line telephone book
Centralized algorithms	Doing routing based on complete information

Fig. 1-15. Potential bottlenecks that designers should try to avoid in very large distributed systems.

Centralized tables are almost as bad as centralized components. How should one keep track of the telephone numbers and addresses of 50 million people? Suppose that each data record could be fit into 50 characters. A single 2.5-gigabyte disk would provide enough storage. But here again, having a single data base would undoubtedly saturate all the communication lines into and out of it. It would also be vulnerable to failures (a single speck of dust could cause a head crash and bring down the entire directory service). Furthermore, here too, valuable network capacity would be wasted shipping queries far away for processing.

Finally, centralized algorithms are also a bad idea. In a large distributed system, an enormous number of messages have to be routed over many lines. From a theoretical point of view, the optimal way to do this is collect complete information about the load on all machines and lines, and then run a graph theory algorithm to compute all the optimal routes. This information can then be spread around the system to improve the routing.

The trouble is that collecting and transporting all the input and output information would again be a bad idea for the reasons discussed above. In fact, any algorithm that operates by collecting information from all sites, sends it to a single machine for processing, and then distributes the results must be avoided.

Only decentralized algorithms should be used. These algorithms generally have the following characteristics, which distinguish them from centralized algorithms:

1. No machine has complete information about the system state.
2. Machines make decisions based only on local information.
3. Failure of one machine does not ruin the algorithm.
4. There is no implicit assumption that a global clock exists.

The first three follow from what we have said so far. The last is perhaps less obvious, but also important. Any algorithm that starts out with: "At precisely 12:00:00 all machines shall note the size of their output queue" will fail because it is impossible to get all the clocks exactly synchronized. Algorithms should take into account the lack of exact clock synchronization. The larger the system, the larger the uncertainty. On a single LAN, with considerable effort it may be possible to get all clocks synchronized down to a few milliseconds, but doing this nationally is tricky. We will discuss distributed clock synchronization in Chap. 3.

1.6. SUMMARY

Distributed systems consist of autonomous CPUs that work together to make the complete system look like a single computer. They have a number of potential selling points, including good price/performance ratios, the ability to match distributed applications well, potentially high reliability, and incremental growth as the workload grows. They also have some disadvantages, such as more complex software, potential communication bottlenecks, and weak security. Nevertheless, there is considerable interest worldwide in building and installing them.

Modern computer systems often have multiple CPUs. These can be organized as multiprocessors (with shared memory) or as multicomputers (without shared memory). Both types can be bus-based or switched. The former tend to be tightly coupled, while the latter tend to be loosely coupled.

The software for multiple CPU systems can be divided into three rough classes. Network operating systems allow users at independent workstations to communicate via a shared file system but otherwise leave each user as the master of his own workstation. Distributed operating systems turn the entire collection of hardware and software into a single integrated system, much like a traditional timesharing system. Shared-memory multiprocessors also offer a single system image, but do so by centralizing everything, so there really is only a single system. Shared-memory multiprocessors are not distributed systems.

Distributed systems have to be designed carefully, since there are many pitfalls for the unwary. A key issue is transparency—hiding all the distribution from the users and even from the application programs. Another issue is flexibility. Since the field is only now in its infancy, the design should be made with the idea of making future changes easy. In this respect, microkernels are superior to monolithic kernels. Other important issues are reliability, performance, and scalability.

PROBLEMS

1. The price/performance ratio of computers has improved by something like 11 orders of magnitude since the first commercial mainframes came out in the early 1950s. The text shows what a similar gain would have meant in the automobile industry. Give another example of what such a large gain means.

2. Name two advantages and two disadvantages of distributed systems over centralized ones.

3. What is the difference between a multiprocessor and a multicomputer?

4. The terms *loosely-coupled system* and *tightly-coupled system* are often used to described distributed computer systems. What is the different between them?

5. What is the different between an MIMD computer and an SIMD computer?

6. A bus-based multiprocessor uses snoopy caches to achieve a coherent memory. Will semaphores work on this machine?

7. Crossbar switches allow a large number of memory requests to be processed at once, giving excellent performance. Why are they rarely used in practice?

8. A multicomputer with 256 CPUs is organized as a 16×16 grid. What is the worst-case delay (in hops) that a message might have to take?

9. Now consider a 256-CPU hypercube. What is the worst-case delay here, again in hops?

10. A multiprocessor has 4096 50-MIPS CPUs connected to memory by an omega network. How fast do the switches have to be to allow a request to