UNIT-I

WHAT IS A DISTRIBUTED SYSTEM?

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.

Example 1: 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 types 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.

Example 2: 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.

Example 3: 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.

Advantages of Distributed Systems over Centralized Systems

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Economics</td>
<td>Microprocessors offer a better price/performance than mainframes</td>
</tr>
<tr>
<td>Speed</td>
<td>A distributed system may have more total computing power than a mainframe</td>
</tr>
<tr>
<td>Inherent distribution</td>
<td>Some applications involve spatially separated machines</td>
</tr>
<tr>
<td>Reliability</td>
<td>If one machine crashes, the system as a whole can still survive</td>
</tr>
<tr>
<td>Incremental growth</td>
<td>Computing power can be added in small increments</td>
</tr>
</tbody>
</table>
Advantages of Distributed Systems over Independent PCs

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data sharing</td>
<td>Allow many users access to a common database</td>
</tr>
<tr>
<td>Device sharing</td>
<td>Allow many users to share expensive peripherals like color printers</td>
</tr>
<tr>
<td>Communication</td>
<td>Make human-to-human communication easier, for example, by electronic mail</td>
</tr>
<tr>
<td>Flexibility</td>
<td>Spread the workload over the available machines in the most cost effective way</td>
</tr>
</tbody>
</table>
A taxonomy of parallel and distributed computer systems.

In a multicomputer, every machine has its own private memory. 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 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.

In **Switched** systems 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.

Another dimension to the taxonomy is that in some systems the machines are **tightly coupled** and in others they are **loosely coupled**.

**Bus-Based Multiprocessors**

Bus-based multiprocessors consist of some number of CPUs all connected to a common bus, along with a memory module.

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 following figure. 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.
However, the introduction of caches makes memory incoherent.

The solution is to use write through cache. 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**.

### 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 below. 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.

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. The **omega network** contains four \( 2 \times 2 \) 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.

**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 following figure we see a bus-based multicomputer. It 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).

![Bus-based Multicomputer](image)

A multicomputer consisting of workstations on a LAN.

**Switched Multicomputers**

In following figure we show 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).
A hypercube is an $n$–dimensional cube. The hypercube of Fig. (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.

**Design Issues:**

**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**.

The concept of transparency can be applied to several aspects of a distributed system, as shown in following figure.

**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.

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

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.

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.

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**
**transparency**, 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.

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.

**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.

There are two kinds of DS. These two models, known as the monolithic kernel and microkernel, respectively, are illustrated in following figure.

![Diagram](image)

(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.

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.

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.

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. A 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.

**Performance**

when running a particular application on a distributed system, it should not be appreciably worse than running the same application on a single processor.

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.

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. 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.

**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

Although little is known about such huge distributed systems, one guiding principle is clear: avoid centralized components, tables, and algorithms.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized components</td>
<td>A single mail server for all users</td>
</tr>
<tr>
<td>Centralized tables</td>
<td>A single on-line telephone book</td>
</tr>
<tr>
<td>Centralized algorithms</td>
<td>Doing routing based on complete information</td>
</tr>
</tbody>
</table>

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.
2.1. LAYERED PROTOCOLS

Due to the absence of shared memory, all communication in distributed systems is based on message passing. When process A wants to communicate with process B, it first builds a message in its own address space. Then it executes a system call that causes the operating system to fetch the message and send it over the network to B. Although this basic idea sounds simple enough, in order to prevent chaos, A and B have to agree on the meaning of the bits being sent. If A sends a brilliant new novel written in French and encoded in IBM's EBCDIC character code, and B expects the inventory of a supermarket written in English and encoded in ASCII, communication will be less than optimal.

Many different agreements are needed. How many volts should be used to signal a 0-bit, and how many volts for a 1-bit? How does the receiver know which is the last bit of the message? How can it detect if a message has been damaged or lost, and what should it do if it finds out? How long are numbers, strings, and other data items, and how are they represented? In short, agreements are needed at a variety of levels, varying from the low-level details of bit transmission to the high-level details of how information is to be expressed.

To make it easier to deal with the numerous levels and issues involved in communication, the International Standards Organization (ISO) has developed a reference model that clearly identifies the various levels involved, gives them standard names, and points out which level should do which job. This model is called the Open Systems Interconnection Reference Model (Day and Zimmerman, 1983), usually abbreviated as ISO OSI or sometimes just the OSI model. Although we do not intend to give a full description of this model and all of its implications here, a short introduction will be helpful. For more details, see (Tanenbaum, 1988).

To start with, the OSI model is designed to allow open systems to communicate. An open system is one that is prepared to communicate with any other open system by using standard rules that govern the format, contents, and meaning of the messages sent and received. These rules are formalized in what are called protocols. Basically, a protocol is an agreement between the communicating parties on how communication is to proceed. When a woman is introduced to a man, she may choose to stick out her hand. He, in turn, may decide either to shake it or kiss it, depending, for example, whether she is an American lawyer at a business meeting or a European princess at a formal ball. Violating the protocol will make communication more difficult, if not impossible.

At a more technological level, many companies make memory boards for the IBM PC. When the CPU wants to read a word from memory, it puts the address and certain control signals on the bus. The memory board is expected to see these signals and respond by putting the word requested on the bus within a certain time interval. If the memory board observes the required bus protocol, it will work correctly, otherwise it will not.

Similarly, to allow a group of computers to communicate over a network, they must all agree on the protocols to be used. The OSI model distinguishes between two general types of protocols. With connection-oriented protocols, before exchanging data, the sender and receiver first explicitly establish a connection, and possibly negotiate the protocol they will use. When they are done, they must release (terminate) the connection. The telephone is a connection-oriented communication system. With connectionless protocols, no setup in advance is needed. The sender just transmits the first message when it is ready. Dropping a letter in a mailbox is an example of connectionless communication. With computers, both connection-oriented and connectionless communication are common.

In the OSI model, communication is divided up into seven levels or layers, as shown in Fig. 2-1. Each layer deals with one specific aspect of the communication. In this way, the problem can be divided up into manageable pieces, each of which can be solved independent of the others. Each
layer provides an interface to the one above it. The interface consists of a set of operations that together define the service the layer is prepared to offer its users.

In the OSI model, when process A on machine 1 wants to communicate with process B on machine 2, it builds a message and passes the message to the application layer on its machine. This layer might be a library procedure, for example, but it could also be implemented in some other way (e.g., inside the operating system, on an external coprocessor chip, etc.). The application layer software then adds a header to the front of the message and passes the resulting message across the layer 6/7 interface to the presentation layer. The presentation layer in turn adds its own header and passes the result down to the session layer, and so on. Some layers add not only a header to the front, but also a trailer to the end. When it hits bottom, the physical layer actually transmits the message, which by now might look as shown in Fig. 2-2.

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

As an example of why layered protocols are important, consider communication between two companies, Zippy Airlines and its caterer, Mushy Meals, Inc. Every month, the head of passenger service at Zippy asks her secretary to contact the sales manager’s secretary at Mushy to order 100,000 boxes of rubber chicken. Traditionally, the orders have gone via the post office. However, as the postal service deteriorates, at some point the two secretaries decide to abandon it and
communicate by FAX. They can do this without bothering their bosses, since their protocol deals with the physical transmission of the orders, not their contents.

Similarly, the head of passenger service can decide to drop the rubber chicken and go for Mushy's new special, prime rib of goat, without that decision affecting the secretaries. The thing to notice is that we have two layers here, the bosses and the secretaries. Each layer has its own protocol (subjects of discussion and technology) that can be changed independently of the other one. It is precisely this independence that makes layered protocols attractive. Each one can be changed as technology improves, without the other ones being affected.

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

In the OSI model, there are not two layers, but seven, as we saw in Fig. 2-1. The collection of protocols used in a particular system is called a protocol suite or protocol stack. In the following sections, we will briefly examine each of the layers in turn, starting at the bottom. Where appropriate, we will also point out some of the protocols used in each layer.

2.1.1. The Physical Layer

The physical layer is concerned with transmitting the 0s and 1s. How many volts to use for 0 and 1, how many bits per second can be sent, and whether transmission can take place in both directions simultaneously are key issues in the physical layer. In addition, the size and shape of the network connector (plug), as well as the number of pins and meaning of each are of concern here.

The physical layer protocol deals with standardizing the electrical, mechanical, and signaling interfaces so that when one machine sends a 0 bit it is actually received as a 0 bit and not a 1 bit. Many physical layer standards have been developed (for different media), for example, the RS-232-C standard for serial communication lines.

2.1.2. The Data Link Layer

The physical layer just sends bits. As long as no errors occur, all is well. However, real communication networks are subject to errors, so some mechanism is needed to detect and correct them. This mechanism is the main task of the data link layer. What it does is to group the bits into units, sometimes called frames, and see that each frame is correctly received.

2.1.3. The Network Layer

On a LAN, there is usually no need for the sender to locate the receiver. It just puts the message out on the network and the receiver takes it off. A wide-area network, however, consists of a large number of machines, each with some number of lines to other machines, rather like a large-scale map showing major cities and roads connecting them. For a message to get from the sender to the receiver it may have to make a number of hops, at each one choosing an outgoing
line to use. The question of how to choose the best path is called routing, and is the primary task of the network layer.

2.1.4. The Transport Layer

Packets can be lost on the way from the sender to the receiver. Although some applications can handle their own error recovery, others prefer a reliable connection. The job of the transport layer is to provide this service. The idea is that the session layer should be able to deliver a message to the transport layer with the expectation that it will be delivered without loss.

Upon receiving a message from the session layer, the transport layer breaks it into pieces small enough for each to fit in a single packet, assigns each one a sequence number, and then sends them all. The discussion in the transport layer header concerns which packets have been sent, which have been received, how many more the receiver has room to accept, and similar topics.

2.1.5. The Session Layer

The session layer is essentially an enhanced version of the transport layer. It provides dialog control, to keep track of which party is currently talking, and it provides synchronization facilities. The latter are useful to allow users to insert checkpoints into long transfers, so that in the event of a crash it is only necessary to go back to the last checkpoint, rather than all the way back to the beginning. In practice, few applications are interested in the session layer and it is rarely supported. It is not even present in the DoD protocol suite.

2.1.6. The Presentation Layer

Unlike the lower layers, which are concerned with getting the bits from the sender to the receiver reliably and efficiently, the presentation layer is concerned with the meaning of the bits. Most messages do not consist of random bit strings, but more structured information such as people’s names, addresses, amounts of money, and so on. In the presentation layer it is possible to define records containing fields like these and then have the sender notify the receiver that a message contains a particular record in a certain format. This makes it easier for machines with different internal representations to communicate.

2.1.7. The Application Layer

The application layer is really just a collection of miscellaneous protocols for common activities such as electronic mail, file transfer, and connecting remote terminals to computers over a network. The best known of these are the X.400 electronic mail protocol and the X.500 directory server. Neither this layer nor the two layers directly under it will be of interest to us in this book.

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.

![](image)

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.

2.3.3. Addressing

In order for a client to send a message to a server, it must know the server’s address. In the example of the preceding section, the server’s address was simply hardwired into header.h as a constant. While this strategy might work in an especially simple system, usually a more sophisticated form of addressing is needed. In this section we will describe some issues concerning addressing.

In our example, the file server has been assigned a numerical address (243), but we have not really specified what this means. In particular, does it refer to a specific machine, or to a specific process? If it refers to a specific machine, the sending kernel can extract it from the message structure and use it as the hardware address for sending the packet to the server. All the sending kernel has to do then is build a frame using the 243 as the data link address and put the frame out on the LAN. The server’s interface board will see the frame, recognize 243 as its own address, and accept it.

If there is only one process running on the destination machine, the kernel will know what to do with the incoming message — give it to the one and only process running there. However, what happens if there are several processes running on the destination machine? Which one gets the message? The kernel has no way of knowing. Consequently, a scheme that uses network addresses to identify processes means that only one process can run on each machine. While this limitation is not fatal, it is sometimes a serious restriction.

An alternative addressing system sends messages to processes rather than to machines. Although this method eliminates all ambiguity about who the real recipient is, it does introduce the problem of how processes are identified. One common scheme is to use two part names, specifying both a machine and a process number. Thus 243.4 or 4@243 or something similar designates process 4 on machine 243. The machine number is used by the kernel to get the message correctly delivered to the proper machine, and the process number is used by the kernel on that machine to determine which process the message is intended for. A nice feature of this approach is that every machine can number its processes starting at 0. No global coordination is needed because there is never any ambiguity between process 0 on machine 243 and process 0 on machine 199. The former is 243.0 and the latter is 199.0. This scheme is illustrated in Fig. 2-10(a).

A slight variation on this addressing scheme uses machine.local-id instead of machine.process. The local-id field is normally a randomly chosen 16-bit or 32-bit integer (or the next one in sequence). One process, typically a server, starts up by making a system call to tell the kernel that it wants to listen to local-id. Later, when a message comes in addressed to machine.local_id, the kernel knows which process to give the message to. Most communication in Berkeley UNIX, for example, uses this method, with 32-bit Internet addresses used for specifying machines and 16-bit numbers for the local-id fields.
2.3.4. Blocking versus Nonblocking Primitives

The message-passing primitives we have described so far are what are called **blocking primitives** (sometimes called **synchronous primitives**). When a process calls `send` it specifies a destination and a buffer to send to that destination. While the message is being sent, the sending process is blocked (i.e., suspended). The instruction following the call to `send` is not executed until the message has been completely sent, as shown in Fig. 2-11(a). Similarly, a call to `receive` does not return control until a message has actually been received and put in the message buffer pointed to by the parameter. The process remains suspended in `receive` until a message arrives, even if it takes hours. In some systems, the receiver can specify from whom it wishes to receive, in which case it remains blocked until a message from that sender arrives.

![Diagram of blocking and nonblocking primitives](image)

**Fig. 2-11.** (a) A blocking send primitive. (b) A nonblocking send primitive.

An alternative to blocking primitives are **nonblocking primitives** (sometimes called **asynchronous primitives**). If `send` is nonblocking, it returns control to the caller
immediately, before the message is sent. The advantage of this scheme is that the sending process can continue computing in parallel with the message transmission, instead of having the CPU go idle (assuming no other process is runnable). The choice between blocking and nonblocking primitives is normally made by the system designers (i.e., either one primitive is available or the other), although in a few systems both are available and users can choose their favorite.

However, the performance advantage offered by nonblocking primitives is offset by a serious disadvantage: the sender cannot modify the message buffer until the message has been sent. The consequences of the process overwriting the message during transmission are too horrible to contemplate. Worse yet, the sending process has no idea of when the transmission is done, so it never knows when it is safe to reuse the buffer. It can hardly avoid touching it forever.

There are two possible ways out. The first solution is to have the kernel copy the message to an internal kernel buffer and then allow the process to continue, as shown in Fig. 2-11(b). From the sender's point of view, this scheme is the same as a blocking call: as soon as it gets control back, it is free to reuse the buffer. Of course, the message will not yet have been sent, but the sender is not hindered by this fact. The disadvantage of this method is that every outgoing message has to be copied from user space to kernel space. With many network interfaces, the message will have to be copied to a hardware transmission buffer later anyway, so the first copy is essentially wasted. The extra copy can reduce the performance of the system considerably.

The second solution is to interrupt the sender when the message has been sent to inform it that the buffer is once again available. No copy is required here, which saves time, but user-level interrupts make programming tricky, difficult, and subject to race conditions, which makes them irreproducible. Most experts agree that although this method is highly efficient and allows the most parallelism, the disadvantages greatly outweigh the advantages: programs based on interrupts are difficult to write correctly and nearly impossible to debug when they are wrong.

Sometimes the interrupt can be disguised by starting up a new thread of control (to discussed in Chap. 4) within the sender's address space. Although this is somewhat cleaner than a raw interrupt, it is still far more complicated than synchronous communication. If only a single thread of control is available, the choices come down to:

1. Blocking send (CPU idle during message transmission).
2. Nonblocking send with copy (CPU time wasted for the extra copy).
3. Nonblocking send with interrupt (makes programming difficult).

Under normal conditions, the first choice is the best. It does not maximize the parallelism, but is simple to understand and simple to implement. It also does not require any kernel buffers to manage. Furthermore, as can be seen from comparing Fig. 2-1 l(a) to Fig. 2-1 l(b), the message will usually be out the door faster if no copy is required. On the other hand, if overlapping processing and transmission are essential for some application, a nonblocking send with copying is the best choice.

For the record, we would like to point out that some authors use a different criterion to distinguish synchronous from asynchronous primitives (Andrews, 1991). In our view, the essential difference between a synchronous primitive and an asynchronous one is whether the sender can reuse the message buffer immediately after getting control back without fear of messing up the send. When the message actually gets to the receiver is irrelevant.

In the alternative view, a synchronous primitive is one in which the sender is blocked until the receiver has accepted the message and the acknowledgement has gotten back to the sender. Everything else is asynchronous in this view. There is complete agreement that if the sender gets control back before the message has been copied or sent, the primitive is asynchronous. Similarly, everyone agrees that when the sender is blocked until the receiver has acknowledged the message, we have a synchronous primitive.
The disagreement comes on whether the intermediate cases (message copied or copied and sent, but not acknowledged) counts as one or the other. Operating systems designers tend to prefer our way, since their concern is with buffer management and message transmission. Programming language designers tend to prefer the alternative definition, because that is what counts at the language level.

Just as send can be blocking or nonblocking, so can receive. A nonblocking receive just tells the kernel where the buffer is, and returns control almost immediately. Again here, how does the caller know when the operation has completed? One way is to provide an explicit wait primitive that allows the receiver to block when it wants to. Alternatively (or in addition to wait), the designers may provide a test primitive to allow the receiver to poll the kernel to check on the status. A variant on this idea is a conditional receive, which either gets a message or signals failure, but in any event returns immediately, or within some timeout interval. Finally, here too, interrupts can be used to signal completion. For the most part, a blocking version of receive is much simpler and greatly preferred.

If multiple threads of control are present within a single address space, the arrival of a message can cause a thread to be created spontaneously. We will come back to this issue after we have looked at threads in Chap. 4.

An issue closely related to blocking versus nonblocking calls is that of timeouts. In a system in which send calls block, if there is no reply, the sender will block forever. To prevent this situation, in some systems the caller may specify a time interval within which it expects a reply. If none arrives in that interval, the send call terminates with an error status.

### 2.3.5. Buffered versus Unbuffered Primitives

Just as system designers have a choice between blocking and nonblocking primitives, they also have a choice between buffered and unbuffered primitives. The primitives we have described so far are essentially unbuffered primitives. What this means is that an address refers to a specific process, as in Fig. 2-9. A call receive(addr, &m) tells the kernel of the machine on which it is running that the calling process is listening to address addr and is prepared to receive one message sent to that address. A single message buffer, pointed to by m, is provided to hold the incoming message. When the message comes in, the receiving kernel copies it to the buffer and unblocks the receiving process. The use of an address to refer to a specific process is illustrated in Fig. 2-12(a).

![Fig. 2-12. (a) Unbuffered message passing. (b) Buffered message passing.](image)

This scheme works fine as long as the server calls receive before the client calls send. The call to receive is the mechanism that tells the server's kernel which address the server is using and where to put the incoming message. The problem arises when the send is done before
the receive. How does the server's kernel know which of its processes (if any) is using the address in the newly arrived message, and how does it know where to copy the message? The answer is simple: it does not.

One implementation strategy is to just discard the message, let the client time out, and hope the server has called receive before the client retransmits. This approach is easy to implement, but with bad luck, the client (or more likely, the client's kernel) may have to try several times before succeeding. Worse yet, if enough consecutive attempts fail, the client's kernel may give up, falsely concluding that the server has crashed or that the address is invalid.

In a similar vein, suppose that two or more clients are using the server of Fig. 2-9(a). After the server has accepted a message from one of them, it is no longer listening to its address until it has finished its work and gone back to the top of the loop to call receive again. If it takes a while to do the work, the other clients may make multiple attempts to send to it, and some of them may give up, depending on the values of their retransmission timers and how impatient they are.

The second approach to dealing with this problem is to have the receiving kernel keep incoming messages around for a little while, just in case an appropriate receive is done shortly. Whenever an "unwanted" message arrives, a timer is started. If the timer expires before a suitable receive happens, the message is discarded.

Although this method reduces the chance that a message will have to be thrown away, it introduces the problem of storing and managing prematurely arriving messages. Buffers are needed and have to be allocated, freed, and generally managed. A conceptually simple way of dealing with this buffer management is to define a new data structure called a mailbox. A process that is interested in receiving messages tells the kernel to create a mailbox for it, and specifies an address to look for in network packets. Henceforth, all incoming messages with that address are put in the mailbox. The call to receive now just removes one message from the mailbox, or blocks (assuming blocking primitives) if none is present. In this way, the kernel knows what to do with incoming messages and has a place to put them. This technique is frequently referred to as a buffered primitive, and is illustrated in Fig. 2-12(b).

At first glance, mailboxes appear to eliminate the race conditions caused by messages being discarded and clients giving up. However, mailboxes are finite and can fill up. When a message arrives for a mailbox that is full, the kernel once again is confronted with the choice of either keeping it around for a while, hoping that at least one message will be extracted from the mailbox in time, or discarding it. These are precisely the same choices we had in the unbuffered case. Although we have perhaps reduced the probability of trouble, we have not eliminated it, and have not even managed to change its nature.

In some systems, another option is available: do not let a process send a message if there is no room to store it at the destination. To make this scheme work, the sender must block until an acknowledgement comes back saying that the message has been received. If the mailbox is full, the sender can be backed up and retroactively suspended as though the scheduler had decided to suspend it just before it tried to send the message. When space becomes available in the mailbox, the sender is allowed to try again.

2.3.6. Reliable versus Unreliable Primitives

So far we have tacitly assumed that when a client sends a message, the server will receive it. As usual, reality is more complicated than our abstract model. Messages can get lost, which affects the semantics of the message passing model. Suppose that blocking primitives are being used. When a client sends a message, it is suspended until the message has been sent. However, when it is restarted, there is no guarantee that the message has been delivered. The message might have been lost.

Three different approaches to this problem are possible. The first one is just to redefine the semantics of send to be unreliable. The system gives no guarantee about messages being
delivered. Implementing reliable communication is entirely up to the users. The post office works this way. When you drop a letter in a letterbox, the post office does its best (more or less) to deliver it, but it promises nothing.

The second approach is to require the kernel on the receiving machine to send an acknowledgement back to the kernel on the sending machine. Only when this acknowledgement is received will the sending kernel free the user (client) process. The acknowledgement goes from kernel to kernel; neither the client nor the server ever sees an acknowledgement. Just as the request from client to server is acknowledged by the server’s kernel, the reply from the server back to the client is acknowledged by the client’s kernel. Thus a request and reply now take four messages, as shown in Fig. 2-13(a).

Fig. 2-13. (a) Individually acknowledged messages. (b) Reply being used as the acknowledgement of the request. Note that the ACKs are handled entirely within the kernels.

The third approach is to take advantage of the fact that client-server communication is structured as a request from the client to the server followed by a reply from the server to the client. In this method, the client is blocked after sending a message. The server’s kernel does not send back an acknowledgement. Instead, the reply itself acts as the acknowledgement. Thus the sender remains blocked until the reply comes in. If it takes too long, the sending kernel can resend the request to guard against the possibility of a lost message. This approach is shown in Fig. 2-13(b).

Although the reply functions as an acknowledgement for the request, there is no acknowledgement for the reply. Whether this omission is serious or not depends on the nature of the request. If, for example, the client asks the server to read a block of a file and the reply is lost, the client will just repeat the request and the server will send the block again. No damage is done and little time is lost.

On the other hand, if the request requires extensive computation on the part of the server, it would be a pity to discard the answer before the server is sure that the client has received the reply. For this reason, an acknowledgement from the client’s kernel to the server’s kernel is sometimes used. Until this packet is received, the server’s send does not complete and the server remains blocked (assuming blocking primitives are used). In any event, if the reply is lost and the request is retransmitted, the server’s kernel can see that the request is an old one and just send the reply again without waking up the server. Thus in some systems the reply is acknowledged and in others it is not [see Fig. 2-13(b)].

A compromise between Fig. 2-13(a) and Fig. 2-13(b) that often works goes like this. When a request arrives at the server’s kernel, a timer is started. If the server sends the reply quickly enough (i.e., before the timer expires), the reply functions as the acknowledgement. If the timer goes off, a separate acknowledgement is sent. Thus in most cases, only two messages are needed, but when a complicated request is being carried out, a third one is used.
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

```c
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).
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.

![Fig. 2-18. Calls and messages in an RPC. Each ellipse represents a single process, with the shaded portion being the stub.](image)

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 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 \textit{send} and \textit{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 \textbf{parameter marshaling}.

As the simplest possible example, consider a remote procedure, \textit{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 \textit{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.

![Fig. 2-19. Computing \textit{sum}(4, 7) remotely.](image-url)
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 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 (is 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 endian and the sparc format is called big endian, after the politicians in Gulliver's Travels who went to war over which end of an egg to break (Cohen, 1981). As an example, consider a server with two parameters, an integer and a four-character string. Each parameter requires one 32-bit word. Figure 2-20(a) shows what the parameter portion of a message built by a client stub on an Intel 486 might look like. The first word contains the integer parameter, 5 in this case, and the second contains the string "JILL".

![Fig. 2-20.](image)

(a) The original message on the 486. (b) The message after receipt on the SPARC. (c) The message after being inverted. The little numbers in boxes indicate the address of each byte.

Since messages are transferred byte for byte (actually, bit for bit) over the network, the first byte sent is the first byte to arrive. In Fig. 2-20(b) we show what the message of Fig. 2-20(a) would look like if received by a SPARC, which numbers its bytes with byte 0 at the left (high-order byte) instead of at the right (low-order byte) as do all the Intel chips. When the server stub reads the parameters at addresses 0 and 4, respectively, it will find an integer equal to 83,886,080 (5×224) and a string "JILL".

One obvious, but unfortunately incorrect, approach is to invert the bytes of each word after they are received, leading to Fig. 2-20(c). Now the integer is 5 and the string is "LLIJ". The problem here is that integers are reversed by the different byte ordering, but strings are not.
Without additional information about what is a string and what is an integer, there is no way to repair the damage.

Fortunately, this information is implicitly available. Remember that the items in the message correspond to the procedure identifier and parameters. Both the client and server know what the types of the parameters are. Thus a message corresponding to a remote procedure with $n$ parameters will have $n+1$ fields, one identifying the procedure and one for each of the $n$ parameters. Once a standard has been agreed upon for representing each of the basic data types, given a parameter list and a message, it is possible to deduce which bytes belong to which parameter, and thus to solve the problem.

As a simple example, consider the procedure of Fig. 2-21 (a). It has three parameters, a character, a floating-point number, and an array of five integers. We might decide to transmit a character in the rightmost byte of a word (leaving the next 3 bytes empty), a float as a whole word, and an array as a group of words equal to the array length, preceded by a word giving the length, as shown in Fig. 2-21(b). Thus given these rules, the client stub for `foobar` knows that it must use the format of Fig. 2-21(b), and the server stub knows that incoming messages for `foobar` will have the format of Fig. 2-21(b). Having the type information for the parameters makes it possible to make any necessary conversions.

![Fig. 2-21.](image)

---

Even with this additional information, there are still some issues open. In particular, how should information be represented in the messages? One way is to devise a network standard or **canonical form** for integers, characters, booleans, floating-point numbers, and so on, and require all senders to convert their internal representation to this form while marshaling. For example, suppose that it is decided to use two's complement for integers, ASCII for characters, 0 (false) and 1 (true) for Booleans, and IEEE format for floating-point numbers, with everything stored in little endian. For any list of integers, characters, Booleans, and floating-point numbers, the exact pattern required is now deterministic down to the last bit. As a result, the server stub no longer has to worry about which byte ordering the client has because the order of the bits in the message is now fixed, independent of the client's hardware.

The problem with this method is that it is sometimes inefficient. Suppose that a big endian client is talking to a big endian server. According to the rules, the client must convert everything to little endian in the message, and the server must convert it back again when it arrives. Although this is unambiguous, it requires two conversions when in fact none were necessary. This observation gives rise to a second approach: the client uses its own native format and indicates in the first byte of the message which format this is. Thus a little endian client builds a little endian message and a big endian client builds a big endian message. As soon as a message comes in, the

---
server stub examines the first byte to see what the client is. If it is the same as the server, no conversion is needed. Otherwise, the server stub converts everything. Although we have only discussed converting from one endian to the other, conversions between one's and two's complement, EBCDIC to ASCII, and so on, can be handled in the same way. The trick is knowing what the message layout is and what the client is. Once these are known, the rest is easy (provided that everyone can convert from everyone else's format).

Now we come to the question of where the stub procedures come from. In many RPC-based systems, they are generated automatically. As we have seen, given a specification of the server procedure and the encoding rules, the message format is uniquely determined. Thus it is possible to have a compiler read the server specification and generate a client stub that packs its parameters into the officially approved message format. Similarly, the compiler can also produce a server stub that unpacks them and calls the server. Having both stub procedures generated from a single formal specification of the server not only makes life easier for the programmers, but reduces the chance of error and makes the system transparent with respect to differences in internal representation of data items.

Finally, we come to our last and most difficult problem: How are pointers passed? The answer is: only with the greatest of difficulty, if at all. Remember that a pointer is meaningful only within the address space of the process in which it is being used. Getting back to our read example discussed earlier, if the second parameter (the address of the buffer) happens to be 1000 on the client, one cannot just pass the number 1000 to the server and expect it to work. Address 1000 on the server might be in the middle of the program text.

One solution is just to forbid pointers and reference parameters in general. However, these are so important that this solution is highly undesirable. In fact, it is not necessary either. In the read example, the client stub knows that the second parameter points to an array of characters. Suppose, for the moment, that it also knows how big the array is. One strategy then becomes apparent: copy the array into the message and send it to the server. The server stub can then call the server with a pointer to this array, even though this pointer has a different numerical value than the second parameter of read has. Changes the server makes using the pointer (e.g., storing data into it) directly affect the message buffer inside the server stub. When the server finishes, the original message can be sent back to the client stub, which then copies it back to the client. In effect, call-by-reference has been replaced by copy/restore. Although this is not always identical, it frequently is good enough.

One optimization makes this mechanism twice as efficient. If the stubs know whether the buffer is an input parameter or an output parameter to the server, one of the copies can be eliminated. If the array is input to the server (e.g., in a call to write) it need not be copied back. If it is output, it need not be sent over in the first place. The way to tell them is in the formal specification of the server procedure. Thus associated with every remote procedure is a formal specification of the procedure, written in some kind of specification language, telling what the parameters are, which are input and which are output (or both), and what their (maximum) sizes are. It is from this formal specification that the stubs are generated by a special stub compiler.

As a final comment, it is worth noting that although we can now handle pointers to simple arrays and structures, we still cannot handle the most general case of a pointer to an arbitrary data structure such as a complex graph. Some systems attempt to deal with this case by actually passing the pointer to the server stub and generating special code in the server procedure for using pointers.

Normally, a pointer is followed (dereferenced) by putting it in a register and indirecing through the register. When this special technique is used, a pointer is dereferenced by sending a message back to the client stub asking it to fetch and send the item being pointed to (reads) or store a value at the address pointed to (writes). While this method works, it is often highly
inefficient. Imagine having the file server store the bytes in the buffer by sending back each one in a separate message. Still, it is better than nothing, and some systems use it.

### 2.4.3. Dynamic Binding

An issue that we have glossed over so far is how the client locates the server. One method is just to hardwire the network address of the server into the client. The trouble with this approach is that it is extremely inflexible. If the server moves or if the server is replicated or if the interface changes, numerous programs will have to be found and recompiled. To avoid all these problems, some distributed systems use what is called **dynamic binding** to match up clients and servers. In this section we will describe the ideas behind dynamic binding.

The starting point for dynamic binding is the server's formal specification. As an example, consider the server of Fig. 2-9(a), specified in Fig. 2-22. The specification tells the name of the server (*file_server*), the version number (3.1), and a list of procedures provided by the server (*read*, *write*, *create*, and *delete*).

```c
#include <header.h>

specification of file_server, version 3.1:
  long read(in char name[MAX_PATH], out char buf[BUF_SIZE], in long bytes,
            in long position);
  long write(in char name[MAX_PATH], in char buf[BUF_SIZE], in long bytes,
             in long position);
  int create(in char[MAX_PATH], in int mode);
  int delete(in char[MAX_PATH]);
end;
```

**Fig. 2-22.** A specification of the stateless server of Fig. 2-9.

For each procedure, the types of the parameters are given. Each parameter is specified as being an *in* parameter, an *out* parameter, or an *in out* parameter. The direction is relative to the server. An *in* parameter, such as the file name, *name*, is sent from the client to the server. This one is used to tell the server which file to read from, write to, create, or delete. Similarly, *bytes* tells the server how many bytes to transfer and *position* tells where in the file to begin reading or writing. An *out* parameter such as *buf in read*, is sent from the server to the client. *Buf* is the place where the file server puts the data that the client has requested. An *in out* parameter, of which there are none in this example, would be sent from the client to the server, modified there, and then sent back to the client (copy/restore). Copy/restore is typically used for pointer parameters in cases where the server both reads and modifies the data structure being pointed to. The directions are crucial, so the client stub knows which parameters to send to the server, and the server stub knows which ones to send back.

As we pointed out earlier, this particular example is a stateless server. For a UNIX-like server, one would have additional procedures *open* and *close*, and different parameters for *read* and *write*. The concept of RPC itself is neutral, permitting the system designers to build any kind of servers they desire.

The primary use of the formal specification of Fig. 2-22 is as input to the stub generator, which produces both the client stub and the server stub. Both are then put into the appropriate libraries. When a user (client) program calls any of the procedures defined by this specification, the corresponding client stub procedure is linked into its binary. Similarly, when the server is compiled, the server stubs are linked with it too.
When the server begins executing, the call to \texttt{initialize} outside the main loop [see Fig. 2-9(a)] \textbf{exports} the server interface. What this means is that the server sends a message to a program called a \texttt{binder}, to make its existence known. This process is referred to as \textbf{registering} the server. To register, the server gives the binder its name, its version number, a unique identifier, typically 32 bits long, and a \texttt{handle} used to locate it. The handle is system dependent, and might be an Ethernet address, an IP address, an X.500 address, a sparse process identifier, or something else. In addition, other information, for example, concerning authentication, might also be supplied. A server can also deregister with the binder when it is no longer prepared to offer service. The binder interface is shown in Fig. 2-23.

<table>
<thead>
<tr>
<th>Call</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Name, version, handle, unique id</td>
<td></td>
</tr>
<tr>
<td>Deregister</td>
<td>Name, version, unique id</td>
<td></td>
</tr>
<tr>
<td>Lookup</td>
<td>Name, version</td>
<td>Handle, unique id</td>
</tr>
</tbody>
</table>

\textbf{Fig. 2-23.} The binder interface.

Given this background, now consider how the client locates the server. When the client calls one of the remote procedures for the first time, say, \texttt{read}, the client stub sees that it is not yet bound to a server, so it sends a message to the binder asking to \textbf{import} version 3.1 of the \texttt{file_server} interface. The binder checks to see if one or more servers have already exported an interface with this name and version number. If no currently running server is willing to support this interface, the \texttt{read} call fails. By including the version number in the matching process, the binder can ensure that clients using obsolete interfaces will fail to locate a server rather than locate one and get unpredictable results due to incorrect parameters.

On the other hand, if a suitable server exists, the binder gives its handle and unique identifier to the client stub. The client stub uses the handle as the address to send the request message to. The message contains the parameters and the unique identifier, which the server's kernel uses to direct the incoming message to the correct server in the event that several servers are running on that machine.

This method of exporting and importing interfaces is highly flexible. For example, it can handle multiple servers that support the same interface. The binder can spread the clients randomly over the servers to even the load if it wants to. It can also poll the servers periodically, automatically deregistering any server that fails to respond, to achieve a degree of fault tolerance. Furthermore, it can also assist in authentication. A server could specify, for example, that it only wished to be used by a specific list of users, in which case the binder would refuse to tell users not on the list about it. The binder can also verify that both client and server are using the same version of the interface.

However, this form of dynamic binding also has its disadvantages. The extra overhead of exporting and importing interfaces costs time. Since many client processes are short lived and each process has to start all over again, the effect may be significant. Also, in a large distributed system, the binder may become a bottleneck, so multiple binders are needed. Consequently, whenever an interface is registered or deregistered, a substantial number of messages will be needed to keep all the binders synchronized and up to date, creating even more overhead.

\textbf{2.4.4. RPC Semantics in the Presence of Failures}

The goal of RPC is to hide communication by making remote procedure calls look just like local ones. With a few exceptions, such as the inability to handle global variables and the subtle differences introduced by using copy/restore for pointer parameters instead of call-by-reference, so
far we have come fairly close. Indeed, as long as both client and server are functioning perfectly, RPC does its job remarkably well. The problem comes in when errors occur. It is then that the differences between local and remote calls are not always easy to mask. In this section we will examine some of the possible errors and what can be done about them.

To structure our discussion, let us distinguish between five different classes of failures that can occur in RPC systems, as follows:

1. The client is unable to locate the server.
2. The request message from the client to the server is lost.
3. The reply message from the server to the client is lost.
4. The server crashes after receiving a request.
5. The client crashes after sending a request.

Each of these categories poses different problems and requires different solutions.

### Client Cannot Locate the Server

To start with, it can happen that the client cannot locate a suitable server. The server might be down, for example. Alternatively, suppose that the client is compiled using a particular version of the client stub, and the binary is not used for a considerable period of time. In the meantime, the server evolves and a new version of the interface is installed and new stubs are generated and put into use. When the client is finally run, the binder will be unable to match it up with a server and will report failure. While this mechanism is used to protect the client from accidentally trying to talk to a server that may not agree with it in terms of what parameters are required or what it is supposed to do, the problem remains of how this failure should be dealt with.

With the server of Fig. 2-9(a), each of the procedures returns a value, with the code –1 conventionally used to indicate failure. For such procedures, just returning –1 will clearly tell the caller that something is amiss. In UNIX, a global variable, *errno*, is also assigned a value indicating the error type. In such a system, adding a new error type "Cannot locate server" is simple.

The trouble is, this solution is not general enough. Consider the *sum* procedure of Fig. 2-19. Here –1 is a perfectly legal value to be returned, for example, the result of adding 7 to –8. Another error-reporting mechanism is needed.

One possible candidate is to have the error raise an exception. In some languages (e.g., Ada), programmers can write special procedures that are invoked upon specific errors, such as division by zero. In C, signal handlers can be used for this purpose. In other words, we could define a new signal type *SIGNOSERVER*, and allow it to be handled in the same way as other signals.

This approach, too, has drawbacks. To start with, not every language has exceptions or signals. To name one, Pascal does not. Another point is that having to write an exception or signal handler destroys the transparency we have been trying to achieve. Suppose that you are a programmer and your boss tells you to write the *sum* procedure. You smile and tell her it will be written, tested, and documented in five minutes. Then she mentions that you also have to write an exception handler as well, just in case the procedure is not there today. At this point it is pretty hard to maintain the illusion that remote procedures are no different from local ones, since writing an exception handler for "Cannot locate server" would be a rather unusual request in a single-processor system.
**Lost Request Messages**

The second item on the list is dealing with lost request messages. This is the easiest one to deal with: just have the kernel start a timer when sending the request. If the timer expires before a reply or acknowledgement comes back, the kernel sends the message again. If the message was truly lost, the server will not be able to tell the difference between the retransmission and the original, and everything will work fine. Unless, of course, so many request messages are lost that the kernel gives up and falsely concludes that the server is down, in which case we are back to "Cannot locate server."

**Lost Reply messages**

Lost replies are considerably more difficult to deal with. The obvious solution is just to rely on the timer again. If no reply is forthcoming within a reasonable period, just send the request once more. The trouble with this solution is that the client's kernel is not really sure why there was no answer. Did the request or reply get lost, or is the server merely slow? It may make a difference.

In particular, some operations can safely be repeated as often as necessary with no damage being done. A request such as asking for the first 1024 bytes of a file has no side effects and can be executed as often as necessary without any harm being done. A request that has this property is said to be **idempotent**.

Now consider a request to a banking server asking to transfer a million dollars from one account to another. If the request arrives and is carried out, but the reply is lost, the client will not know this and will retransmit the message. The bank server will interpret this request as a new one, and will carry it out too. Two million dollars will be transferred. Heaven forbid that the reply is lost 10 times. Transferring money is not idempotent.

One way of solving this problem is to try to structure all requests in an idempotent way. In practice, however, many requests (e.g., transferring money) are inherently nonidempotent, so something else is needed. Another method is to have the client's kernel assign each request a sequence number. By having each server's kernel keep track of the most recently received sequence number from each client's kernel that is using it, the server's kernel can tell the difference between an original request and a retransmission and can refuse to carry out any request a second time. An additional safeguard is to have a bit in the message header that is used to distinguish initial requests from retransmissions (the idea being that it is always safe to perform an original request; retransmissions may require more care).

**Server Crashes**

The next failure on the list is a server crash. It too relates to idempotency, but unfortunately it cannot be solved using sequence numbers. The normal sequence of events at a server is shown in Fig. 2-24(a). A request arrives, is carried out, and a reply is sent. Now consider Fig. 2-24(b). A request arrives and is carried out, just as before, but the server crashes before it can send the reply. Finally, look at Fig. 2-24(c). Again a request arrives, but this time the server crashes before it can even be carried out.
The annoying part of Fig. 2-24 is that the correct treatment differs for (b) and (c). In (b) the system has to report failure back to the client (e.g., raise an exception), whereas in (c) it can just retransmit the request. The problem is that the client's kernel cannot tell which is which. All it knows is that its timer has expired.

Three schools of thought exist on what to do here. One philosophy is to wait until the server reboots (or rebinds to a new server) and try the operation again. The idea is to keep trying until a reply has been received, then give it to the client. This technique is called **at least once semantics** and guarantees that the RPC has been carried out at least one time, but possibly more.

The second philosophy gives up immediately and reports back failure. This way is called **at most once semantics** and guarantees that the rpc has been carried out at most one time, but possibly none at all.

The third philosophy is to guarantee nothing. When a server crashes, the client gets no help and no promises. The RPC may have been carried out anywhere from 0 to a large number of times. The main virtue of this scheme is that it is easy to implement.

None of these are terribly attractive. What one would like is **exactly once semantics**, but as can be seen fairly easily, there is no way to arrange this in general. Imagine that the remote operation consists of printing some text, and is accomplished by loading the printer buffer and then setting a single bit in some control register to start the printer. The crash can occur a microsecond before setting the bit, or a microsecond afterward. The recovery procedure depends entirely on which it is, but there is no way for the client to discover it.

In short, the possibility of server crashes radically changes the nature of RPC and clearly distinguishes single-processor systems from distributed systems. In the former case, a server crash also implies a client crash, so recovery is neither possible nor necessary. In the latter it is both possible and necessary to take some action.

### Client Crashes

The final item on the list of failures is the client crash. What happens if a client sends a request to a server to do some work and crashes before the server replies? At this point a computation is active and no parent is waiting for the result. Such an unwanted computation is called an **orphan**.

Orphans can cause a variety of problems. As a bare minimum, they waste CPU cycles. They can also lock files or otherwise tie up valuable resources. Finally, if the client reboots and does the RPC again, but the reply from the orphan comes back immediately afterward, confusion can result.

What can be done about orphans? Nelson (1981) proposed four solutions. In solution 1, before a client stub sends an **RPC** message, it makes a log entry telling what it is about to do. The log is kept on disk or some other medium that survives crashes. After a reboot, the log is checked and the orphan is explicitly killed off. This solution is called **extermination**.

The disadvantage of this scheme is the horrendous expense of writing a disk record for every RPC. Furthermore, it may not even work, since orphans themselves may do RPCs, thus...
creating grandorphans or further descendants that are impossible to locate. Finally, the network may be partitioned, due to a failed gateway, making it impossible to kill them, even if they can be located. All in all, this is not a promising approach.

In solution 2, called reincarnation, all these problems can be solved without the need to write disk records. The way it works is to divide time up into sequentially numbered epochs. When a client reboots, it broadcasts a message to all machines declaring the start of a new epoch. When such a broadcast comes in, all remote computations are killed. Of course, if the network is partitioned, some orphans may survive. However, when they report back, their replies will contain an obsolete epoch number, making them easy to detect.

Solution 3 is a variant on this idea, but less Draconian. It is called gentle reincarnation. When an epoch broadcast comes in, each machine checks to see if it has any remote computations, and if so, tries to locate their owner. Only if the owner cannot be found is the computation killed.

Finally, we have solution 4, expiration, in which each RPC is given a standard amount of time, $T$, to do the job. If it cannot finish, it must explicitly ask for another quantum, which is a nuisance. On the other hand, if after a crash the server waits a time $T$ before rebooting, all orphans are sure to be gone. The problem to be solved here is choosing a reasonable value of $T$ in the face of RPCs with wildly differing requirements.

In practice, none of these methods are desirable. Worse yet, killing an orphan may have unforeseen consequences. For example, suppose that an orphan has obtained locks on one or more files or database records. If the orphan is suddenly killed, these locks may remain forever. Also, an orphan may have already made entries in various remote queues to start up other processes at some future time, so even killing the orphan may not remove all traces of it. Orphan elimination is discussed in more detail by Panzieri and Shrivastava (1988).

2.5. GROUP COMMUNICATION

An underlying assumption intrinsic to RPC is that communication involves only two parties, the client and the server. Sometimes there are circumstances in which communication involves multiple processes, not just two. For example, consider a group of file servers cooperating to offer a single, fault-tolerant file service. In such a system, it might be desirable for a client to send a message to all the servers, to make sure that the request could be carried out even if one of them crashed. RPC cannot handle communication from one sender to many receivers, other than by performing separate RPCs with each one. In this section we will discuss alternative communication mechanisms in which a message can be sent to multiple receivers in one operation.

2.5.1. Introduction to Group Communication

A group is a collection of processes that act together in some system or user-specified way. The key property that all groups have is that when a message is sent to the group itself, all members of the group receive it. It is a form of one-to-many communication (one sender, many receivers), and is contrasted with point-to-point communication in fig. 2-30.
Fig. 2-30. (a) Point-to-point communication is from one sender to one receiver. (b) One-to-
many communication is from one sender to multiple receivers.

Groups are dynamic. New groups can be created and old groups can be destroyed. A process
can join a group or leave one. A process can be a member of several groups at the same time.
Consequently, mechanisms are needed for managing groups and group membership.

Groups are roughly analogous to social organizations. A person might be a member of a book
club, a tennis club, and an environmental organization. On a particular day, he might receive
mailings (messages) announcing a new birthday cake cookbook from the book club, the annual
Mother’s Day tennis tournament from the tennis club, and the start of a campaign to save the
Southern groundhog from the environmental organization. At any moment, he is free to leave any
or all of these groups, and possibly join other groups.

Although in this book we will study only operating system (i.e., process) groups, it is worth
mentioning that other groups are also commonly encountered in computer systems. For example,
on the USENET computer network, there are hundreds of news groups, each about a specific
subject. When a person sends a message to a particular news group, all members of the group
receive it, even if there are tens of thousands of them. These higher-level groups usually have
looser rules about who is a member, what the exact semantics of message delivery are, and so on,
than do operating system groups. In most cases, this looseness is not a problem.

The purpose of introducing groups is to allow processes to deal with collections of processes
as a single abstraction. Thus a process can send a message to a group of servers without having to
know how many there are or where they are, which may change from one call to the next.

How group communication is implemented depends to a large extent on the hardware. On
some networks, it is possible to create a special network address (for example, indicated by setting
one of the high-order bits to 1), to which multiple machines can listen. When a packet is sent to
one of these addresses, it is automatically delivered to all machines listening to the address. This
technique is called multicasting. Implementing groups using multicast is straightforward: just
assign each group a different multicast address.

Networks that do not have multicasting sometimes still have broadcasting, which means
that packets containing a certain address (e.g., 0) are delivered to all machines. Broadcasting can
also be used to implement groups, but it is less efficient. Each machine receives each broadcast, so
its software must check to see if the packet is intended for it. If not, the packet is discarded, but
some time is wasted processing the interrupt. Nevertheless, it still takes only one packet to reach
all the members of a group.

Finally, if neither multicasting nor broadcasting is available, group communication can still be
implemented by having the sender transmit separate packets to each of the members of the group.
For a group with \( n \) members, \( n \) packets are required, instead of one packet when either
multicasting or broadcasting is used. Although less efficient, this implementation is still workable, especially if most groups are small. The sending of a message from a single sender to a single receiver is sometimes called \textbf{unicasting} (point-to-point transmission), to distinguish it from multicasting and broadcasting.

\section*{2.5.2. Design Issues}

Group communication has many of the same design possibilities as regular message passing, such as buffered versus unbuffered, blocking versus nonblocking, and so forth. However, there are also a large number of additional choices that must be made because sending to a group is inherently different from sending to a single process. Furthermore, groups can be organized in various ways internally. They can also be addressed in novel ways not relevant in point-to-point communication. In this section we will look at some of the most important design issues and point out the various alternatives.

\section*{Closed Groups versus Open Groups}

Systems that support group communication can be divided into two categories depending on who can send to whom. Some systems support \textit{closed groups}, in which only the members of the group can send to the group. Outsiders cannot send messages to the group as a whole, although they may be able to send messages to individual members. In contrast, other systems support \textit{open groups}, which do not have this property. When open groups are used, any process in the system can send to any group. The difference between closed and open groups is shown in Fig. 2-31.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{groupDiagram.png}
\caption{(a) Outsiders may not send to a closed group. (b) Outsiders may send to an open group.}
\end{figure}

The decision as to whether a system supports closed or open groups usually relates to the reason groups are being supported in the first place. Closed groups are typically used for parallel processing. For example, a collection of processes working together to play a game of chess might form a closed group. They have their own goal and do not interact with the outside world.

On the other hand, when the idea of groups is to support replicated servers, it is important that processes that are not members (clients) can send to the group. In addition, the members of the group may also need to use group communication, for example to decide who should carry out a particular request. The distinction between closed and open groups is often made for implementation reasons.
Peer Groups versus Hierarchical Groups

The distinction between closed and open groups relates to who can communicate with the group. Another important distinction has to do with the internal structure of the group. In some groups, all the processes are equal. No one is boss and all decisions are made collectively. In other groups, some kind of hierarchy exists. For example, one process is the coordinator and all the others are workers. In this model, when a request for work is generated, either by an external client or by one of the workers, it is sent to the coordinator. The coordinator then decides which worker is best suited to carry it out, and forwards it there. More complex hierarchies are also possible, of course. These communication patterns are illustrated in Fig. 2-32.

![Peer group](image1.png) ![Hierarchical group](image2.png)

Fig. 2-32. (a) Communication in a peer group. (b) Communication in a simple hierarchical group.

Each of these organizations has its own advantages and disadvantages. The peer group is symmetric and has no single point of failure. If one of the processes crashes, the group simply becomes smaller, but can otherwise continue. A disadvantage is that decision making is more complicated. To decide anything, a vote has to be taken, incurring some delay and overhead.

The hierarchical group has the opposite properties. Loss of the coordinator brings the entire group to a grinding halt, but as long as it is running, it can make decisions without bothering everyone else. For example, a hierarchical group might be appropriate for a parallel chess program. The coordinator takes the current board, generates all the legal moves from it, and farms them out to the workers for evaluation. During this evaluation, new boards are generated and sent back to the coordinator to have them evaluated. When a worker is idle, it asks the coordinator for a new board to work on. In this manner, the coordinator controls the search strategy and prunes the game tree (e.g., using the alpha-beta search method), but leaves the actual evaluation to the workers.

Group Membership

When group communication is present, some method is needed for creating and deleting groups, as well as for allowing processes to join and leave groups. One possible approach is to have a group server to which all these requests can be sent. The group server can then maintain a complete database of all the groups and their exact membership. This method is straightforward, efficient, and easy to implement. Unfortunately, it shares with all centralized techniques a major disadvantage: a single point of failure. If the group server crashes, group management ceases to
exist. Probably most or all groups will have to be reconstructed from scratch, possibly terminating whatever work was going on.

The opposite approach is to manage group membership in a distributed way. In an open group, an outsider can send a message to all group members announcing its presence. In a closed group, something similar is needed (in effect, even closed groups have to be open with respect to joining). To leave a group, a member just sends a goodbye message to everyone.

So far, all of this is straightforward. However, there are two issues associated with group membership that are a bit trickier. First, if a member crashes, it effectively leaves the group. The trouble is, there is no polite announcement of this fact as there is when a process leaves voluntarily. The other members have to discover this experimentally by noticing that the crashed member no longer responds to anything. Once it is certain that the crashed member is really down, it can be removed from the group.

The other knotty issue is that leaving and joining have to be synchronous with messages being sent. In other words, starting at the instant that a process has joined a group, it must receive all messages sent to that group. Similarly, as soon as a process has left a group, it must not receive any more messages from the group, and the other members must not receive any more messages from it. One way of making sure that a join or leave is integrated into the message stream at the right place is to convert this operation into a message sent to the whole group.

One final issue relating to group membership is what to do if so many machines go down that the group can no longer function at all. Some protocol is needed to rebuild the group. Invariably, some process will have to take the initiative to start the ball rolling, but what happens if two or three try at the same time? The protocol will have to be able to withstand this.

**Group Addressing**

In order to send a message to a group, a process must have some way of specifying which group it means. In other words, groups need to be addressed, just as processes do. One way is to give each group a unique address, much like a process address. If the network supports multicast, the group address can be associated with a multicast address, so that every message sent to the group address can be multicast. In this way, the message will be sent to all those machines that need it, and no others.

If the hardware supports broadcast but not multicast, the message can be broadcast. Every kernel will then get it and extract from it the group address. If none of the processes on the machine is a member of the group, the broadcast is simply discarded. Otherwise, it is passed to all group members.

Finally, if neither multicast nor broadcast is supported, the kernel on the sending machine will have to have a list of machines that have processes belonging to the group. The kernel then sends each one a point-to-point message. These three implementation methods are shown in Fig. 2-33. The thing to notice is that in all three cases, a process just sends a message to a group address and it is delivered to all the members. How that happens is up to the operating system. The sender is not aware of the size of the group or whether communication is implemented by multicasting, broadcasting, or unicasting.
A second method of group addressing is to require the sender to provide an explicit list of all destinations (e.g., IP addresses). When this method is used, the parameter in the call to `send` that specifies the destination is a pointer to a list of addresses. This method has the serious drawback that it forces user processes (i.e., the group members) to be aware of precisely who is a member of which group. In other words, it is not transparent. Furthermore, whenever group membership changes, the user processes must update their membership lists. In Fig. 2-33, this administration can easily be done by the kernels to hide it from the user processes.

Group communication also allows a third, and quite novel method of addressing as well, which we will call **predicate addressing**. With this system, each message is sent to all members of the group (or possibly the entire system) using one of the methods described above, but with a new twist. Each message contains a predicate (Boolean expression) to be evaluated. The predicate can involve the receiver's machine number, its local variables, or other factors. If the predicate evaluates to TRUE, the message is accepted. If it evaluates to FALSE, the message is discarded. Using this scheme it is possible, for example, to send a message to only those machines that have at least 4M of free memory and which are willing to take on a new process.
UNIT III

Synchronization in Distributed Systems

In single CPU systems, critical regions, mutual exclusion, and other synchronization problems are solved using methods such as semaphores and monitors. These methods are not well suited to use in distributed systems because they rely (implicitly) on the existence of shared memory.

CLOCK SYNCHRONIZATION

Synchronization in distributed systems is more complicated than in centralized ones because the former have to use distributed algorithms. It is usually not possible (or desirable) to collect all the information about the system in one place, and then let some process examine it and make a decision as is done in the centralized case. In general, distributed algorithms have the following properties:

1. The relevant information is scattered among multiple machines.
2. Processes make decisions based only on local information.
3. A single point of failure in the system should be avoided.
4. No common clock or other precise global time source exists.

The first three points all say that it is unacceptable to collect all the information in a single place for processing. For example, to do resource allocation (assigning I/O devices in a deadlock-free way), it is generally not acceptable to send all the requests to a single manager process, which examines them all and grants or denies requests based on information in its tables. In a large system, such a solution puts a heavy burden on that one process.

Furthermore, having a single point of failure like this makes the system unreliable. Ideally, a distributed system should be more reliable than the individual machines. If one goes down, the rest should be able to continue to function.

In a distributed system, achieving agreement on time is not trivial.

Example unix make command

When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

Logical Clocks

With a single computer and a single clock, it does not matter much if this clock is off by a small amount. Since all processes on the machine use the same clock, they will still be internally consistent. For example, if the file input.c has time 2151 and file input.o has time 2150, make will recompile the source file, even if the clock is off by 2 and the true times are 2153 and 2152, respectively. All that really matters are the relative times.

As soon as multiple CPUs are introduced, each with its own clock, the situation changes. Although the frequency at which a crystal oscillator runs is usually fairly stable, it is impossible to guarantee that the crystals in different computers all run at exactly the same frequency. In practice, when a system has n computers, all n crystals will run at slightly different rates, causing
the (software) clocks gradually to get out of sync and give different values when read out. This difference in time values is called **clock skew**. As a consequence of this clock skew, programs that expect the time associated with a file, object, process, or message to be correct and independent of the machine on which it was generated (i.e., which clock it used) can fail, as we saw in the `make` example above.

Lamport pointed out that clock synchronization need not be absolute. If two processes do not interact, it is not necessary that their clocks be synchronized. Furthermore, he pointed out that what usually matters is not that all processes agree on exactly what time it is, but rather, that they agree on the order in which events occur.

For many purposes, it is sufficient that all machines agree on the same time. It is not essential that this time also agree with the real time as announced on the radio every hour. For running `make`, for example, it is adequate that all machines agree that it is 10:00, even if it is really 10:02. Thus for a certain class of algorithms, it is the internal consistency of the clocks that matters, not whether they are particularly close to the real time. For these algorithms, it is conventional to speak of the clocks as **logical clocks**.

When the additional constraint is present that the clocks must not only be the same, but also must not deviate from the real time by more than a certain amount, the clocks are called **physical clocks**.

To synchronize logical clocks, Lamport defined a relation called **happens-before**. The expression $a \rightarrow b$ is read "a happens before b" and means that all processes agree that first event $a$ occurs, then afterward, event $b$ occurs. The happens-before relation can be observed directly in two situations:

1. If $a$ and $b$ are events in the same process, and $a$ occurs before $b$, then $a \rightarrow b$ is true.
2. If $a$ is the event of a message being sent by one process, and $b$ is the event of the message being received by another process, then $a \rightarrow b$ is also true. A message cannot be received before it is sent, or even at the same time it is sent, since it takes a finite amount of time to arrive.

Happens-before is a transitive relation, so if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$. If two events, $x$ and $y$, happen in different processes that do not exchange messages (not even indirectly via third parties), then $x \rightarrow y$ is not true, but neither is $y \rightarrow x$. These events are said to be **concurrent**, which simply means that nothing can be said (or need be said) about when they happened or which is first.

What we need is a way of measuring time such that for every event, $a$, we can assign it a time value $C(a)$ on which all processes agree. These time values must have the property that if $a \rightarrow b$, then $C(a)<C(b)$. To rephrase the conditions we stated earlier, if $a$ and $b$ are two events within the same process and $a$ occurs before $b$, then $C(a)<C(b)$. Similarly, if $a$ is the sending of a message by one process and $b$ is the reception of that message by another process, then $C(a)$ and $C(b)$ must be assigned in such a way that everyone agrees on the values of $C(a)$ and $C(b)$ with $C(a)<C(b)$. In addition, the clock time, $C$, must always go forward (increasing), never backward (decreasing). Corrections to time can be made by adding a positive value, never by subtracting one.

Now let us look at the algorithm Lamport proposed for assigning times to events. Consider the three processes depicted in following figure. The processes run on different machines, each with its own clock, running at its own speed. As can be seen from the figure, when the clock has ticked 6 times in process 0, it has ticked 8 times in process 1 and 10 times in process 2. Each clock runs at a constant rate, but the rates are different due to differences in the crystals.
At time 6, process 0 sends message $A$ to process 1. How long this message takes to arrive depends on whose clock you believe. In any event, the clock in process 1 reads 16 when it arrives. If the message carries the starting time, 6, in it, process 1 will conclude that it took 10 ticks to make the journey. This value is certainly possible. According to this reasoning, message $B$ from 1 to 2 takes 16 ticks, again a plausible value.

Now comes the fun part. Message $C$ from 2 to 1 leaves at 60 and arrives at 56. Similarly, message $D$ from 1 to 0 leaves at 64 and arrives at 54. These values are clearly impossible. It is this situation that must be prevented.

Lamport's solution follows directly from the happened-before relation. Since $C$ left at 60, it must arrive at 61 or later. Therefore, each message carries the sending time, according to the sender's clock. When a message arrives and the receiver's clock shows a value prior to the time the message was sent, the receiver fast forwards its clock to be one more than the sending time. In Fig. 3-2(b) we see that $C$ now arrives at 61. Similarly, $D$ arrives at 70.

With one small addition, this algorithm meets our requirements for global time. The addition is that between every two events, the clock must tick at least once. If a process sends or receives two messages in quick succession, it must advance its clock by (at least) one tick in between them.

In some situations, an additional requirement is desirable: no two events ever occur at exactly the same time. To achieve this goal, we can attach the number of the process in which the event occurs to the low-order end of the time, separated by a decimal point. Thus if events happen in processes 1 and 2, both with time 40, the former becomes 40.1 and the latter becomes 40.2.

Using this method, we now have a way to assign time to all events in a distributed system subject to the following conditions:

1. If $a$ happens before $b$ in the same process, $C(a) < C(b)$.
2. If $a$ and $b$ represent the sending and receiving of a message, $C(a) < C(b)$.
3. For all events $a$ and $b$, $C(a) 
eq C(b)$.

This algorithm gives us a way to provide a total ordering of all events in the system. Many other distributed algorithms need such an ordering to avoid ambiguities, so the algorithm is widely cited in the literature.

**MUTUAL EXCLUSION**

Systems involving multiple processes are often most easily programmed using critical regions. When a process has to read or update certain shared data structures, it first enters a critical region...
to achieve mutual exclusion and ensure that no other process will use the shared data structures at the same time. In single-processor systems, critical regions are protected using semaphores, monitors, and similar constructs. We will now look at a few examples of how critical regions and mutual exclusion can be implemented in distributed systems.

A Centralized Algorithm

The most straightforward way to achieve mutual exclusion in a distributed system is to simulate how it is done in a one-processor system. One process is elected as the coordinator (e.g., the one running on the machine with the highest network address). Whenever a process wants to enter a critical region, it sends a request message to the coordinator stating which critical region it wants to enter and asking for permission. If no other process is currently in that critical region, the coordinator sends back a reply granting permission, as shown below. When the reply arrives, the requesting process enters the critical region.

![Diagram showing a centralized algorithm for mutual exclusion]

(a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted. (b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply. (c) When process 1 exits the critical region, it tells the coordinator, which then replies to 2.

Now suppose that another process, 2 in Fig.(b), asks for permission to enter the same critical region. The coordinator knows that a different process is already in the critical region, so it cannot grant permission. The exact method used to deny permission is system dependent. In Fig.(b), the coordinator just refrains from replying, thus blocking process 2, which is waiting for a reply. Alternatively, it could send a reply saying "permission denied." Either way, it queues the request from 2 for the time being.

When process 1 exits the critical region, it sends a message to the coordinator releasing its exclusive access, as shown in Fig.(c). The coordinator takes the first item off the queue of deferred requests and sends that process a grant message. If the process was still blocked (i.e., this is the first message to it), it unblocks and enters the critical region. If an explicit message has already been sent denying permission, the process will have to poll for incoming traffic, or block later. Either way, when it sees the grant, it can enter the critical region.

Advantages:
1. It is easy to see that the algorithm guarantees mutual exclusion: the coordinator only lets one process at a time into each critical region.
2. It is also fair, since requests are granted in the order in which they are received.
3. No process ever waits forever (no starvation).
4. The scheme is easy to implement, too, and requires only three messages per use of a critical region (request, grant, release).
5. It can also be used for more general resource allocation rather than just managing critical regions.
Disadvantages:
1. The coordinator is a single point of failure, so if it crashes, the entire system may go down.
2. If processes normally block after making a request, they cannot distinguish a dead coordinator from "permission denied" since in both cases no message comes back.
3. In addition, in a large system, a single coordinator can become a performance bottleneck.

**A Distributed Algorithm**

Having a single point of failure is frequently unacceptable, so researchers have looked for distributed mutual exclusion algorithms. Ricart and Agrawala's algorithm requires that there be a total ordering of all events in the system. That is, for any pair of events, such as messages, it must be unambiguous which one happened first. Lamport's algorithm can be used to provide time-stamps for distributed mutual exclusion.

The algorithm works as follows. When a process wants to enter a critical region, it builds a message containing the name of the critical region it wants to enter, its process number, and the current time. It then sends the message to all other processes, conceptually including itself. The sending of messages is assumed to be reliable; that is, every message is acknowledged. Reliable group communication if available, can be used instead of individual messages.

When a process receives a request message from another process, the action it takes depends on its state with respect to the critical region named in the message. Three cases have to be distinguished:

1. If the receiver is not in the critical region and does not want to enter it, it sends back an OK message to the sender.
2. If the receiver is already in the critical region, it does not reply. Instead, it queues the request.
3. If the receiver wants to enter the critical region but has not yet done so, it compares the timestamp in the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. If the incoming message is lower, the receiver sends back an OK message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

After sending out requests asking permission to enter a critical region, a process sits back and waits until everyone else has given permission. As soon as all the permissions are in, it may enter the critical region. When it exits the critical region, it sends OK messages to all processes on its queue and deletes them all from the queue.

Let us try to understand why the algorithm works. If there is no conflict, it clearly works. However, suppose that two processes try to enter the same critical region simultaneously, as shown in Following fig(a).
(a) Two processes want to enter the same critical region at the same moment. (b) Process 0 has the lowest timestamp, so it wins. (c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region.

Process 0 sends everyone a request with timestamp 8, while at the same time, process 2 sends everyone a request with timestamp 12. Process 1 is not interested in entering the critical region, so it sends OK to both senders. Processes 0 and 2 both see the conflict and compare timestamps. Process 2 sees that it has lost, so it grants permission to 0 by sending OK. Process 0 now queues the request from 2 for later processing and enters the critical region, as shown in Fig. (b). When it is finished, it removes the request from 2 from its queue and sends an OK message to process 2, allowing the latter to enter its critical region, as shown in Fig. The algorithm works because in the case of a conflict, the lowest timestamp wins and everyone agrees on the ordering of the timestamps.

As with the centralized algorithm discussed above, mutual exclusion is guaranteed without deadlock or starvation. The number of messages required per entry is now 2(\(n-1\)), where the total number of processes in the system is \(n\). Best of all, no single point of failure exists.

Unfortunately, the single point of failure has been replaced by \(n\) points of failure. If any process crashes, it will fail to respond to requests. This silence will be interpreted (incorrectly) as denial of permission, thus blocking all subsequent attempts by all processes to enter all critical regions.

Another problem with this algorithm is that either a group communication primitive must be used, or each process must maintain the group membership list itself, including processes entering the group, leaving the group, and crashing. The method works best with small groups of processes that never change their group memberships.

Finally, recall that one of the problems with the centralized algorithm is that making it handle all requests can lead to a bottleneck. In the distributed algorithm, all processes are involved in all decisions concerning entry into critical regions. If one process is unable to handle the load, it is unlikely that forcing everyone to do exactly the same thing in parallel is going to help much.

**A Token Ring Algorithm**

A completely different approach to achieving mutual exclusion in a distributed system is illustrated in following fig. Here we have a bus network, as shown in Fig.(a), with no inherent ordering of the processes. In software, a logical ring is constructed in which each process is assigned a position in the ring, as shown in Fig.(b). The ring positions may be allocated in numerical order of network addresses or some other means. It does not matter what the ordering is. All that matters is that each process knows who is next in line after itself.

When the ring is initialized, process 0 is given a **token**. The token circulates around the ring. It is passed from process \(k\) to process \(k+1\) (modulo the ring size) in point-to-point messages. When a process acquires the token from its neighbor, it checks to see if it is attempting to enter a critical region. If so, the process enters the region, does all the work it needs to, and leaves the region. After it has exited, it passes the token along the ring. It is not permitted to enter a second critical region using the same token.

If a process is handed the token by its neighbor and is not interested in entering a critical region, it just passes it along. As a consequence, when no processes want to enter any critical regions, the token just circulates at high speed around the ring.
(a) An unordered group of processes on a network. (b) A logical ring constructed in software.

The correctness of this algorithm is evident. Only one process has the token at any instant, so only one process can be in a critical region. Since the token circulates among the processes in a well-defined order, starvation cannot occur. Once a process decides it wants to enter a critical region, at worst it will have to wait for every other process to enter and leave one critical region.

As usual, this algorithm has problems too. If the token is ever lost, it must be regenerated. In fact, detecting that it is lost is difficult, since the amount of time between successive appearances of the token on the network is unbounded. The fact that the token has not been spotted for an hour does not mean that it has been lost; somebody may still be using it.

The algorithm also runs into trouble if a process crashes, but recovery is easier than in the other cases. If we require a process receiving the token to acknowledge receipt, a dead process will be detected when its neighbor tries to give it the token and fails. At that point the dead process can be removed from the group, and the token holder can throw the token over the head of the dead process to the next member down the line, or the one after that, if necessary. Of course, doing so requires that everyone maintains the current ring configuration.

**A Comparison of the Three Algorithms**

A brief comparison of the three mutual exclusion algorithms we have looked at is instructive. In the following figure we have listed the algorithms and three key properties: the number of messages required for a process to enter and exit a critical region, the delay before entry can occur (assuming messages are passed sequentially over a LAN), and some problems associated with each algorithm.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Messages per entry/exit</th>
<th>Delay before entry (in message times)</th>
<th>Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>3</td>
<td>2</td>
<td>Coordinator crash</td>
</tr>
<tr>
<td>Distributed</td>
<td>2(n–1)</td>
<td>2(n–1)</td>
<td>Crash of any process</td>
</tr>
<tr>
<td>Token ring</td>
<td>1 to ∞</td>
<td>0 to n–1</td>
<td>Lost token, process crash</td>
</tr>
</tbody>
</table>

A comparison of three mutual exclusion algorithms.
The centralized algorithm is simplest and also most efficient. It requires only three messages to enter and leave a critical region: a request and a grant to enter, and a release to exit. The distributed algorithm requires \( n-1 \) request messages, one to each of the other processes, and an additional \( n-1 \) grant messages, for a total of \( 2(n-1) \). With the token ring algorithm, the number is variable. If every process constantly wants to enter a critical region, then each token pass will result in one entry and exit, for an average of one message per critical region entered. At the other extreme, the token may sometimes circulate for hours without anyone being interested in it. In this case, the number of messages per entry into a critical region is unbounded.

The delay from the moment a process needs to enter a critical region until its actual entry also varies for the three algorithms. When critical regions are short and rarely used, the dominant factor in the delay is the actual mechanism for entering a critical region. When they are long and frequently used, the dominant factor is waiting for everyone else to take their turn.

Finally, all three algorithms suffer badly in the event of crashes. Special measures and additional complexity must be introduced to avoid having a crash bring down the entire system. It is slightly ironic that the distributed algorithms are even more sensitive to crashes than the centralized one. In a fault-tolerant system, none of these would be suitable, but if crashes are very infrequent, they are all acceptable.

**ELECTION ALGORITHMS**

Many distributed algorithms require one process to act as coordinator, initiator, sequencer, or otherwise perform some special role. We have already seen several examples, such as the coordinator in the centralized mutual exclusion algorithm. In general, it does not matter which process takes on this special responsibility, but one of them has to do it.

**The Bully Algorithm**

As a first example, consider the **bully algorithm** devised by Garcia-Molina (1982). When a process notices that the coordinator is no longer responding to requests, it initiates an election. A process, \( P \), holds an election as follows:

1. \( P \) sends an **ELECTION** message to all processes with higher numbers.
2. If no one responds, \( P \) wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over. \( P \)'s job is done.

At any moment, a process can get an **ELECTION** message from one of its lower-numbered colleagues. When such a message arrives, the receiver sends an **OK** message back to the sender to indicate that he is alive and will take over. The receiver then holds an election, unless it is already holding one. Eventually, all processes give up but one, and that one is the new coordinator. It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.

If a process that was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job. Thus the biggest guy in town always wins, hence the name "bully algorithm."

Following figure shows an example of how the bully algorithm works. The group consists of eight processes, numbered from 0 to 7. Previously process 7 was the coordinator, but it has just crashed. Process 4 is the first one to notice this, so it sends **ELECTION** messages to all the processes higher than it, namely 5, 6, and 7, as shown in Fig. . Processes 5 and 6 both respond with **OK**, as shown in Fig. (b). Upon getting the first of these responses, 4 knows that its job is over. It knows that one of these bigwigs will take over and become coordinator. It just sits back and waits to see who the winner will be (although at this point it can make a pretty good guess).
The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

In Fig. (c), both 5 and 6 hold elections, each one only sending messages to those processes higher than itself. In Fig. (d) process 6 tells 5 that it will take over. At this point 6 knows that 7 is dead and that it (6) is the winner. If there is state information to be collected from disk or elsewhere to pick up where the old coordinator left off, 6 must now do what is needed. When it is ready to take over, 6 announces this by sending a COORDINATOR message to all running processes. When 4 gets this message, it can now continue with the operation it was trying to do when it discovered that 7 was dead, but using 6 as the coordinator this time. In this way the failure of 7 is handled and the work can continue.

If process 7 is ever restarted, it will just send all the others a COORDINATOR message and bully them into submission.

**A Ring Algorithm**

Another election algorithm is based on the use of a ring, but without a token. We assume that the processes are physically or logically ordered, so that each process knows who its successor is. When any process notices that the coordinator is not functioning, it builds an ELECTION message containing its own process number and sends the message to its successor. If the successor is down, the sender skips over the successor and goes to the next member along the ring, or the one after that, until a running process is located. At each step, the sender adds its own process number to the list in the message.

Eventually, the message gets back to the process that started it all. That process recognizes this event when it receives an incoming message containing its own process number. At that point, the message type is changed to COORDINATOR and circulated once again, this time to inform everyone else who the coordinator is (the list member with the highest number) and who the members of the new ring are. When this message has circulated once, it is removed and everyone goes back to work.
Election algorithm using a ring.

In above figure, we see what happens if two processes, 2 and 5, discover simultaneously that the previous coordinator, process 7, has crashed. Each of these builds an ELECTION message and starts circulating it. Eventually, both messages will go all the way around, and both 2 and 5 will convert them into COORDINATOR messages, with exactly the same members and in the same order. When both have gone around again, both will be removed. It does no harm to have extra messages circulating; at most it wastes a little bandwidth.

**ATOMIC TRANSACTIONS**

All the synchronization techniques we have studied so far are essentially low level, like semaphores. They require the programmer to be intimately involved with all the details of mutual exclusion, critical region management, deadlock prevention, and crash recovery. What we would really like is a much higher-level abstraction, one that hides these technical issues and allows the programmer to concentrate on the algorithms and how the processes work together in parallel. Such an abstraction exists and is widely used in distributed systems. We will call it an atomic transaction, or simply transaction. The term atomic action is also widely used. In this section we will examine the use, design, and implementation of atomic transactions.

Now look at a modern banking application that updates an online data base in place. The customer calls up the bank using a PC with a modem with the intention of withdrawing money from one account and depositing it in another. The operation is performed in two steps:
1. Withdraw(amount, account1).
2. Deposit(amount, account2).

If the telephone connection is broken after the first one but before the second one, the first account will have been debited but the second one will not have been credited. The money vanishes into thin air.

Being able to group these two operations in an atomic transaction would solve the problem. Either both would be completed, or neither would be completed. The key is rolling back to the initial state if the transaction fails to complete. What we really want is a way to rewind the data base as we could the magnetic tapes. This ability is what the atomic transaction has to offer.

**The Transaction Model**

We will now develop a more precise model of what a transaction is and what its properties are. The system is assumed to consist of some number of independent processes, each of which can fail at random. Communication is normally unreliable in that messages can be lost, but lower levels can use a timeout and retransmission protocol to recover from lost messages. Thus for this
discussion we can assume that communication errors are handled transparently by underlying software.

**Stable Storage**

Storage comes in three categories. First we have ordinary RAM memory, which is wiped out when the power fails or a machine crashes. Next we have disk storage, which survives CPU failures but which can be lost in disk head crashes.

Finally, we have **stable storage**, which is designed to survive anything except major calamities such as floods and earthquakes. Stable storage can be implemented with a pair of ordinary disks, as shown in Fig. (a). Each block on drive 2 is an exact copy of the corresponding block on drive 1. When a block is updated, first the block on drive 1 is updated and verified, then the same block on drive 2 is done.

![Stable storage](image)

(a) Stable storage. (b) Crash after drive 1 is updated. (c) Bad spot.

Suppose that the system crashes after drive 1 is updated but before drive 2 is updated, as shown in Fig. (b). Upon recovery, the disk can be compared block for block. Whenever two corresponding blocks differ, it can be assumed that drive 1 is the correct one (because drive 1 is always updated before drive 2), so the new block is copied from drive 1 to drive 2. When the recovery process is complete, both drives will again be identical.

Another potential problem is the spontaneous decay of a block. Dust particles or general wear and tear can give a previously valid block a sudden checksum error, without cause or warning, as shown in Fig. (c). When such an error is detected, the bad block can be regenerated from the corresponding block on the other drive.

As a consequence of its implementation, stable storage is well suited to applications that require a high degree of fault tolerance, such as atomic transactions. When data are written to stable storage and then read back to check that they have been written correctly, the chance of them subsequently being lost is extremely small.

**Transaction Primitives**

Programming using transactions requires special primitives that must either be supplied by the operating system or by the language runtime system. Examples are:

1. BEGIN_TRANSACTION: Mark the start of a transaction.
2. END_TRANSACTION: Terminate the transaction and try to commit.
3. ABORT_TRANSACTION: Kill the transaction; restore the old values.
4. READ: Read data from a file (or other object).
5. WRITE: Write data to a file (or other object).

The exact list of primitives depends on what kinds of objects are being used in the transaction. In a mail system, there might be primitives to send, receive, and forward mail. In an accounting system, they might be quite different. READ and WRITE are typical examples, however. Ordinary statements, procedure calls, and so on, are also allowed inside a transaction.

BEGIN_TRANSACTION and END_TRANSACTION are used to delimit the scope of a transaction. The operations between them form the body of the transaction. Either all of them are executed or none are executed. These may be system calls, library procedures, or bracketing statements in a language, depending on the implementation.

(a) Transaction to reserve three flights commits. (b) Transaction aborts when third flight is unavailable.

Properties of Transactions

Transactions have four essential properties. Transactions are:

1. Atomic: To the outside world, the transaction happens indivisibly.
2. Consistent: The transaction does not violate system invariants.
3. Isolated: Concurrent transactions do not interfere with each other.
4. Durable: Once a transaction commits, the changes are permanent.

These properties are often referred to by their initial letters, ACID.

The first key property exhibited by all transactions is that they are atomic. This property ensures that each transaction either happens completely, or not at all, and if it happens, it happens in a single indivisible, instantaneous action. While a transaction is in progress, other processes (whether or not they are themselves involved in transactions) cannot see any of the intermediate states.

Suppose, for example, that some file is 10 bytes long when a transaction starts to append to it. If other processes read the file while the transaction is in progress, they see only the original 10 bytes, no matter how many bytes the transaction has appended. If the transaction commits successfully, the file grows instantaneously to its new size at the moment of commitment, with no intermediate states, no matter how many operations it took to get it there.

The second property says that they are consistent. What this means is that if the system has certain invariants that must always hold, if they held before the transaction, they will hold afterward too. For example, in a banking system, a key invariant is the law of conservation of money. After any internal transfer, the amount of money in the bank must be the same as it was before the transfer, but for a brief moment during the transaction, this invariant may be violated. The violation is not visible outside the transaction, however.

The third property says that transactions are isolated or serializable. What it means is that if two or more transactions are running at the same time, to each of them and to other processes, the final result looks as though all transactions ran sequentially in some (system dependent) order.

In following Fig.(a)-(c) we have three transactions that are executed simultaneously by three separate processes. If they were to be run sequentially, the final value of $x$ would be 1, 2, or 3, depending which one ran last ($x$ could be a shared variable, a file, or some other kind of object). In
Fig. (d) we see various orders, called schedules, in which they might be interleaved. Schedule 1 is actually serialized. In other words, the transactions run strictly sequentially, so it meets the serializability condition by definition. Schedule 2 is not serialized, but is still legal because it results in a value for $x$ that could have been achieved by running the transactions strictly sequentially. The third one is illegal since it sets $x$ to 5, something that no sequential order of the transactions could produce. It is up to the system to ensure that individual operations are interleaved correctly. By allowing the system the freedom to choose any ordering of the operations it wants to — provided that it gets the answer right — we eliminate the need for programmers to do their own mutual exclusion, thus simplifying the programming.

<table>
<thead>
<tr>
<th>Schedule 1</th>
<th>Schedule 2</th>
<th>Schedule 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 0$;</td>
<td>$x = 0$;</td>
<td>$x = 0$;</td>
</tr>
<tr>
<td>$x = x + 1$;</td>
<td>$x = x + 1$;</td>
<td>$x = x + 1$;</td>
</tr>
<tr>
<td>$x = 0$;</td>
<td>$x = 0$;</td>
<td>$x = 0$;</td>
</tr>
<tr>
<td>$x = x + 2$;</td>
<td>$x = x + 2$;</td>
<td>$x = x + 2$;</td>
</tr>
<tr>
<td>$x = 0$;</td>
<td>$x = 0$;</td>
<td>$x = 0$;</td>
</tr>
<tr>
<td>$x = x + 3$;</td>
<td>$x = x + 3$;</td>
<td>$x = x + 3$;</td>
</tr>
</tbody>
</table>

(a)-(c) Three transactions. (d) Possible schedules.

The fourth property says that transactions are **durable**. It refers to the fact that once a transaction commits, no matter what happens, the transaction goes forward and the results become permanent. No failure after the commit can undo the results or cause them to be lost.

**Implementation**

Transactions sound like a great idea, but how are they implemented? That is the question we will tackle in this section. It should be clear by now that if each process executing a transaction just updates the objects it uses (files, data base records, etc.) in place, transactions will not be atomic and changes will not vanish magically if the transaction aborts. Furthermore, the results of running multiple transactions will not be serializable either. Clearly, some other implementation method is required. Two methods are commonly used. They will be discussed in turn below.

**Private Workspace**

Conceptually, when a process starts a transaction, it is given a private workspace containing all the files (and other objects) to which it has access. Until the transaction either commits or aborts, all of its reads and writes go to the private workspace, rather than the "real" one, by which we mean the normal file system. This observation leads directly to the first implementation method: actually giving a process a private workspace at the instant it begins a transaction.

The problem with this technique is that the cost of copying everything to a private workspace is prohibitive, but various optimizations make it feasible. The first optimization is based on the realization that when a process reads a file but does not modify it, there is no need for a private copy. It can just use the real one (unless it has been changed since the transaction started). Consequently, when a process starts a transaction, it is sufficient to create a private workspace for it that is empty except for a pointer back to its parent's workspace. When the transaction is at the top level, the parent's workspace is the "real" file system. When the process opens a file for
reading, the back pointers are followed until the file is located in the parent's (or further ancestor's) workspace.

When a file is opened for writing, it can be located in the same way as for reading, except that now it is first copied to the private workspace. However, a second optimization removes most of the copying, even here. Instead of copying the entire file, only the file's index is copied into the private workspace. The index is the block of data associated with each file telling where its disk blocks are. In UNIX, the index is the i-node. Using the private index, the file can be read in the usual way, since the disk addresses it contains are for the original disk blocks. However, when a file block is first modified, a copy of the block is made and the address of the copy inserted into the index, as shown in Fig. 3-18. The block can then be updated without affecting the original. Appended blocks are handled this way too. The new blocks are sometimes called shadow blocks.

![Diagram](image)

(a) The file index and disk blocks for a three-block file. (b) The situation after a transaction has modified block 0 and appended block 3. (c) After committing.

As can be seen from the above figure (b), the process running the transaction sees the modified file, but all other processes continue to see the original file. In a more complex transaction, the private workspace might contain a large number of files instead of just one. If the transaction aborts, the private workspace is simply deleted and all the private blocks that it points to are put back on the free list. If the transaction commits, the private indices are moved into the parent's workspace atomically, as shown in Fig. (c). The blocks that are no longer reachable are put onto the free list.

**Writeahead Log**

The other common method of implementing transactions is the writeahead log, sometimes called an intentions list. With this method, files are actually modified in place, but before any block is changed, a record is written to the writeahead log on stable storage telling which transaction is making the change, which file and block is being changed, and what the old and new values are. Only after the log has been written successfully is the change made to the file.

Following Figure gives an example of how the log works. In following figure (a) we have a simple transaction that uses two shared variables (or other objects), x and y, both initialized to 0. For each of the three statements inside the transaction, a log record is written before executing the statement, giving the old and new values, separated by a slash.
If the transaction succeeds and is committed, a commit record is written to the log, but the data structures do not have to be changed, as they have already been updated. If the transaction aborts, the log can be used to back up to the original state. Starting at the end and going backward, each log record is read and the change described in it undone. This action is called a **rollback**.

The log can also be used for recovering from crashes. Suppose that the process doing the transaction crashes just after having written the last log record of Fig. (d), but before changing $x$. After the failed machine is rebooted, the log is checked to see if any transactions were in progress at the time of the crash. When the last record is read and the current value of $x$ is seen to be 1, it is clear that the crash occurred **before** the update was made, so $x$ is set to 4. If, on the other hand, $x$ is 4 at the time of recovery, it is equally clear that the crash occurred **after** the update, so nothing need be changed. Using the log, it is possible to go forward (do the transaction) or go backward (undo the transaction).

**Two-Phase Commit Protocol**

As we have pointed out repeatedly, the action of committing a transaction must be done atomically, that is, instantaneously and indivisibly. In a distributed system, the commit may require the cooperation of multiple processes on different machines, each of which holds some of the variables, files, and data bases, and other objects changed by the transaction. In this section we will study a protocol for achieving atomic commit in a distributed system.

The protocol we will look at is called the **two-phase commit protocol**. Although it is not the only such protocol, it is probably the most widely used. The basic idea is illustrated in the following fig. One of the processes involved functions as the coordinator. Usually, this is the one executing the transaction. The commit protocol begins when the coordinator writes a log entry saying that it is starting the commit protocol, followed by sending each of the other processes involved (the subordinates) a message telling them to prepare to commit.

![The two-phase commit protocol when it succeeds.](image)
When a subordinate gets the message it checks to see if it is ready to commit, makes a log entry, and sends back its decision. When the coordinator has received all the responses, it knows whether to commit or abort. If all the processes are prepared to commit, the transaction is committed. If one or more are unable to commit (or do not respond), the transaction is aborted. Either way, the coordinator writes a log entry and then sends a message to each subordinate informing it of the decision. It is this write to the log that actually commits the transaction and makes it go forward no matter what happens afterward.

Due to the use of the log on stable storage, this protocol is highly resilient in the face of (multiple) crashes. If the coordinator crashes after having written the initial log record, upon recovery it can just continue where it left off, repeating the initial message if need be. If it crashes after having written the result of the vote to the log, upon recovery it can just reinform all the subordinates of the result. If a subordinate crashes before having replied to the first message, the coordinator will keep sending it messages, until it gives up. If it crashes later, it can see from the log where it was, and thus what it must do.

**Concurrency Control**

When multiple transactions are executing simultaneously in different processes (on different processors), some mechanism is needed to keep them out of each other's way. That mechanism is called a **concurrency control algorithm**.

**Locking**

The oldest and most widely used concurrency control algorithm is **locking**. In the simplest form, when a process needs to read or write a file (or other object) as part of a transaction, it first locks the file. Locking can be done using a single centralized lock manager, or with a local lock manager on each machine for managing local files. In both cases the lock manager maintains a list of locked files, and rejects all attempts to lock files that are already locked by another process. Since well-behaved processes do not attempt to access a file before it has been locked, setting a lock on a file keeps everyone else away from it and thus ensures that it will not change during the lifetime of the transaction. Locks are normally acquired and released by the transaction system and do not require action by the programmer.

This basic scheme is restrictive and can be improved by distinguishing read locks from write locks. If a read lock is set on a file, other read locks are permitted. Read locks are set to make sure that the file does not change, but there is no reason to forbid other transactions from reading the file. In contrast, when a file is locked for writing, no other locks of any kind are permitted. Thus read locks are shared, but write locks must be exclusive.

For simplicity, we have assumed that the unit of locking is the entire file. In practice, it might be a smaller item, such as an individual record or page, or a larger item, such as an entire database. The issue of how large an item to lock is called the **granularity of locking**. The finer the granularity, the more precise the lock can be, and the more parallelism can be achieved (e.g., by not blocking a process that wants to use the end of a file just because some other process is using the beginning). On the other hand, fine-grained locking requires more locks, is more expensive, and is more likely to lead to deadlocks.
Acquiring and releasing locks precisely at the moment they are needed or no longer needed can lead to inconsistency and deadlocks. Instead, most transactions that are implemented by locking use what is called **two-phase locking**. In two-phase locking, which is illustrated in above Fig., the process first acquires all the locks it needs during the **growing phase**, then releases them during the **shrinking phase**. If the process refrains from updating any files until it reaches the shrinking phase, failure to acquire some lock can be dealt with simply by releasing all locks, waiting a little while, and starting all over. Furthermore, it can be proven that if all transactions use two-phase locking, all schedules formed by interleaving them are serializable. This is why two-phase locking is widely used.

In many systems, the shrinking phase does not take place until the transaction has finished running and has either committed or aborted. This policy, called **strict two-phase locking**, has two main advantages. First, a transaction always reads a value written by a committed transaction; therefore, one never has to abort a transaction because its calculations were based on a file it should not have seen. Second, all lock acquisitions and releases can be handled by the system without the transaction being aware of them: locks are acquired whenever a file is to be accessed and released when the transaction has finished. This policy eliminates **cascaded aborts**: having to undo a committed transaction because it saw a file it should not have seen.

Locking, even two-phase locking, can lead to deadlocks. If two processes each try to acquire the same pair of locks but in the opposite order, a deadlock may result. The usual techniques apply here, such as acquiring all locks in some canonical order to prevent hold-and-wait cycles. Also possible is deadlock detection by maintaining an explicit graph of which process has which locks, and checking the graph for cycles. Finally, when it is known in advance that a lock will never be held longer than $T$ sec, a timeout scheme can be used: if a lock remains continuously under the same ownership for longer than $T$ sec, there must be a deadlock.

**Optimistic Concurrency Control**

A second approach to handling multiple transactions at the same time is **optimistic concurrency control**. The idea behind this technique is surprisingly simple: just go ahead and do whatever you want to, without paying attention to what anybody else is doing. If there is a problem, worry about it later. (Many politicians use this algorithm, too.) In practice, conflicts are relatively rare, so most of the time it works all right.

Although conflicts may be rare, they are not impossible, so some way is needed to handle them. What optimistic concurrency control does is keep track of which files have been read and written. At the point of committing, it checks all other transactions to see if any of its files have been changed since the transaction started. If so, the transaction is aborted. If not, it is committed.

Optimistic concurrency control fits best with the implementation based on private workspaces. That way, each transaction changes its files privately, without interference from the others. At the end, the new files are either committed or released.
The big advantages of optimistic concurrency control are that it is deadlock free and allows maximum parallelism because no process ever has to wait for a lock. The disadvantage is that sometimes it may fail, in which case the transaction has to be run all over again. Under conditions of heavy load, the probability of failure may go up substantially, making optimistic concurrency control a poor choice.

**Timestamps**

A completely different approach to concurrency control is to assign each transaction a timestamp at the moment it does BEGIN_TRANSACTION. Using Lamport's algorithm, we can ensure that the timestamps are unique, which is important here. Every file in the system has a read timestamp and a write timestamp associated with it, telling which committed transaction last read and wrote it, respectively. If transactions are short and widely spaced in time, it will normally occur that when a process tries to access a file, the file's read and write timestamps will be lower (older) than the current transaction's timestamp. This ordering means that the transactions are being processed in the proper order, so everything is all right.

When the ordering is incorrect, it means that a transaction that started later than the current one has managed to get in there, access the file, and commit. This situation means that the current transaction is too late, so it is aborted. In the timestamp method, we do not mind if concurrent transactions use the same files, as long as the lower numbered transaction always goes first.

It is easiest to explain the timestamp method by means of an example. Imagine that there are three transactions, alpha, beta, and gamma. Alpha ran a long time ago, and used every file needed by beta and gamma, so all their files have read and write timestamps set to alpha's timestamp. Beta and gamma start concurrently, with beta having a lower timestamp than gamma (but higher than alpha, of course).

In above fig. (c) and (d) beta is out of luck. Gamma has either read (c) or written (d) the file and committed. Beta's transaction is aborted. However, it can apply for a new timestamp and start all over again.

Now look at reads. In Fig. (e), there is no conflict, so the read can happen immediately. In Fig. (f), some interloper has gotten in there and is trying to write the file. The interloper's timestamp is lower than beta's, so beta simply waits until the interloper commits, at which time it can read the new file and continue.

In Fig.(g), gamma has changed the file and already committed. Again beta must abort. In Fig. (h), gamma is in the process of changing the file, although it has not committed yet. Still, beta is too late and must abort.
Timestamping has different properties than locking. When a transaction encounters a larger (later) timestamp, it aborts, whereas under the same circumstances with locking it would either wait or be able to proceed immediately. On the other hand, it is deadlock free, which is a big plus.

All in all, transactions offer many advantages and thus are a promising technique for building reliable distributed systems. Their chief problem is their great implementation complexity, which yields low performance. These problems are being worked on, and perhaps in due course they will be solved.

**DEADLOCKS IN DISTRIBUTED SYSTEMS**

Deadlocks in distributed systems are similar to deadlocks in single-processor systems, only worse. They are harder to avoid, prevent, or even detect, and harder to cure when tracked down because all the relevant information is scattered over many machines. In some systems, such as distributed data base systems, they can be extremely serious, so it is important to understand how they differ from ordinary deadlocks and what can be done about them.

Some people make a distinction between two kinds of distributed deadlocks: communication deadlocks and resource deadlocks. A communication deadlock occurs, for example, when process $A$ is trying to send a message to process $B$, which in turn is trying to send one to process $C$, which is trying to send one to $A$. There are various scenarios in which this situation leads to deadlock, such as no buffers being available. A resource deadlock occurs when processes are fighting over exclusive access to I/O devices, files, locks, or other resources.

Various strategies are used to handle deadlocks. Four of the best-known ones are listed and discussed below.

1. The ostrich algorithm (ignore the problem).
2. Detection (let deadlocks occur, detect them, and try to recover).
3. Prevention (statically make deadlocks structurally impossible).
4. Avoidance (avoid deadlocks by allocating resources carefully).

All four are potentially applicable to distributed systems. The ostrich algorithm is as good and as popular in distributed systems as it is in single-processor systems. In distributed systems used for programming, office automation, process control, and many other applications, no system-wide deadlock mechanism is present, although individual applications, such as distributed data bases, can implement their own if they need one.

Deadlock detection and recovery is also popular, primarily because prevention and avoidance are so difficult. We will discuss several algorithms for deadlock detection below.

Deadlock prevention is also possible, although more difficult than in single-processor systems. However, in the presence of atomic transactions, some new options become available. Two algorithms are discussed below.

Finally, deadlock avoidance is never used in distributed systems. It is not even used in single-processor systems, so why should it be used in the more difficult case of distributed systems? The problem is that the banker’s algorithm and similar algorithms need to know (in advance) how much of each resource every process will eventually need. This information is rarely, if ever, available.

**Distributed Deadlock Detection**

Finding general methods for preventing or avoiding distributed deadlocks appears to be quite difficult, so many researchers have tried to deal with the simpler problem of just detecting deadlocks, rather than trying to inhibit their occurrence.

However, the presence of atomic transactions in some distributed systems makes a major conceptual difference. When a deadlock is detected in a conventional operating system, the way to resolve it is to kill off one or more processes. Doing so invariably leads to one or more unhappy users. When a deadlock is detected in a system based on atomic transactions, it is resolved by aborting one or more transactions. But as we have seen in detail above, transactions have been
designed to withstand being aborted. When a transaction is aborted because it contributes to a
deadlock, the system is first restored to the state it had before the transaction began, at which
point the transaction can start again. With a little bit of luck, it will succeed the second time. Thus
the difference is that the consequences of killing off a process are much less severe when
transactions are used than when they are not used.

Centralized Deadlock Detection

As a first attempt, we can use a centralized deadlock detection algorithm and try to imitate
the nondistributed algorithm. Although each machine maintains the resource graph for its own
processes and resources, a central coordinator maintains the resource graph for the entire system
(the union of all the individual graphs). When the coordinator detects a cycle, it kills off one
process to break the deadlock.

Unlike the centralized case, where all the information is automatically available in the right
place, in a distributed system it has to be sent there explicitly. Each machine maintains the graph
for its own processes and resources. Several possibilities exist for getting it there. First, whenever
an arc is added or deleted from the resource graph, a message can be sent to the coordinator
providing the update. Second, periodically, every process can send a list of arcs added or deleted
since the previous update. This method requires fewer messages than the first one. Third, the
coordinator can ask for information when it needs it.

Unfortunately, none of these methods work well. Consider a system with
processes $A$ and $B$ running on machine 0, and process $C$ running on machine 1. Three resources
exist: $R$, $S$, and $T$. Initially, the situation is as shown in Fig. (a) and (b): $A$ holds $S$ but
wants $R$, which it cannot have because $B$ is using it; $C$ has $T$ and wants $S$, too. The coordinator's
view of the world is shown in Fig. (c). This configuration is safe. As soon as $B$ finishes, $A$ can
get $R$ and finish, releasing $S$ for $C$.

After a while, $B$ releases $R$ and asks for $T$, a perfectly legal and safe swap. Machine 0 sends a
message to the coordinator announcing the release of $R$, and machine 1 sends a message to the
coordinator announcing the fact that $B$ is now waiting for its resource, $T$. Unfortunately, the
message from machine 1 arrives first, leading the coordinator to construct the graph of Fig. (d).
The coordinator incorrectly concludes that a deadlock exists and kills some process. Such a
situation is called a false deadlock. Many deadlock algorithms in distributed systems produce
false deadlocks like this due to incomplete or delayed information.

One possible way out might be to use Lamport's algorithm to provide global time. Since the
message from machine 1 to the coordinator is triggered by the request from machine 0, the
message from machine 1 to the coordinator will indeed have a later timestamp than the message from machine 0 to the coordinator. When the coordinator gets the message from machine 1 that leads it to suspect deadlock, it could send a message to every machine in the system saying: "I just received a message with timestamp $T$ which leads to deadlock. If anyone has a message for me with an earlier timestamp, please send it immediately." When every machine has replied, positively or negatively, the coordinator will see that the arc from $R$ to $B$ has vanished, so the system is still safe. Although this method eliminates the false deadlock, it requires global time and is expensive. Furthermore, other situations exist where eliminating false deadlock is much harder.

**Distributed Deadlock Detection**

the Chandy-Misra-Haas algorithm (Chandy et al., 1983). In this algorithm, processes are allowed to request multiple resources (e.g., locks) at once, instead of one at a time. By allowing multiple requests simultaneously, the growing phase of a transaction can be speeded up considerably. The consequence of this change to the model is that a process may now wait on two or more resources simultaneously.

In the following figure, we present a modified resource graph, where only the processes are shown. Each arc passes through a resource, as usual, but for simplicity the resources have been omitted from the figure. Notice that process 3 on machine 1 is waiting for two resources, one held by process 4 and one held by process 5.

Some of the processes are waiting for local resources, such as process 1, but others, such as process 2, are waiting for resources that are located on a different machine. It is precisely these cross-machine arcs that make looking for cycles difficult. The Chandy-Misra-Haas algorithm is invoked when a process has to wait for some resource, for example, process 0 blocking on process 1. At that point a special probe message is generated and sent to the process (or processes) holding the needed resources. The message consists of three numbers: the process that just blocked, the process sending the message, and the process to whom it is being sent. The initial message from 0 to 1 contains the triple $(0, 0, 1)$.

When the message arrives, the recipient checks to see if it itself is waiting for any processes. If so, the message is updated, keeping the first field but replacing the second field by its own process number and the third one by the number of the process it is waiting for. The message is then sent to the process on which it is blocked. If it is blocked on multiple processes, all of them are sent (different) messages. This algorithm is followed whether the resource is local or remote. In above figure we see the remote messages labeled $(0, 2, 3)$, $(0, 4, 6)$, $(0, 5, 7)$, and $(0, 8, 0)$. If a message goes all the way around and comes back to the original sender, that is, the process listed in the first field, a cycle exists and the system is deadlocked.

There are various ways in which the deadlock can be broken. One way is to have the process that initiated the probe commit suicide. However, this method has problems if several processes invoke the algorithm simultaneously. In above figure, for example, imagine that both 0 and 6 block at the same moment, and both initiate probes. Each would eventually discover the deadlock, and each would kill itself. This is overkill. Getting rid of one of them is enough.
An alternative algorithm is to have each process add its identity to the end of the probe message so that when it returned to the initial sender, the complete cycle would be listed. The sender can then see which process has the highest number, and kill that one or send it a message asking it to kill itself. Either way, if multiple processes discover the same cycle at the same time, they will all choose the same victim.

**Distributed Deadlock Prevention**

Deadlock prevention consists of carefully designing the system so that deadlocks are structurally impossible. Various techniques include allowing processes to hold only one resource at a time, requiring processes to request all their resources initially, and making processes release all resources when asking for a new one. All of these are cumbersome in practice. A method that sometimes works is to order all the resources and require processes to acquire them in strictly increasing order. This approach means that a process can never hold a high resource and ask for a low one, thus making cycles impossible.

However, in a distributed system with global time and atomic transactions, two other practical algorithms are possible. Both are based on the idea of assigning each transaction a global timestamp at the moment it starts. As in many timestamp-based algorithms, in these two it is essential that no two transactions are ever assigned exactly the same timestamp. As we have seen, Lamport’s algorithm guarantees uniqueness (effectively by using process numbers to break ties).

The idea behind the algorithm is that when one process is about to block waiting for a resource that another process is using, a check is made to see which has a larger timestamp (i.e., is younger). We can then allow the wait only if the waiting process has a lower timestamp (is older) than the process waited for. In this manner, following any chain of waiting processes, the timestamps always increase, so cycles are impossible. Alternatively, we can allow processes to wait only if the waiting process has a higher timestamp (is younger) than the process waited for, in which case the timestamps decrease along the chain.

Although both methods prevent deadlocks, it is wiser to give priority to older processes. They have run longer, so the system has a larger investment in them, and they are likely to hold more resources. Also, a young process that is killed off will eventually age until it is the oldest one in the system, so this choice eliminates starvation. As we have pointed out before, killing a transaction is relatively harmless, since by definition it can be restarted safely later.

To make this algorithm clearer, consider the situation of following figure. In (a), an old process wants a resource held by a young process. In (b), a young process wants a resource held by an old process. In one case we should allow the process to wait; in the other we should kill it. Suppose that we label (a) dies and (b) wait. Then we are killing off an old process trying to use a resource held by a young process, which is inefficient. Thus we must label it the other way, as shown in the figure. Under these conditions, the arrows always point in the direction of increasing transaction numbers, making cycles impossible. This algorithm is called wait-die.

Once we are assuming the existence of transactions, we can do something that had previously been forbidden: take resources away from running processes. In effect we are saying that when a conflict arises, instead of killing the process making the request, we can kill the resource owner.
Without transactions, killing a process might have severe consequences, since the process might have modified files, for example. With transactions, these effects will vanish magically when the transaction dies.

Now consider the situation of following figure, where we are going to allow preemption. Given that our system believes in ancestor worship, as we discussed above, we do not want a young whippersnapper preempting a venerable old sage, so Fig. (a) and not Fig. (b) is labeled preempt. We can now safely label Fig. (b) wait. This algorithm is known as **wound-wait**, because one transaction is supposedly wounded (it is actually killed) and the other waits. It is unlikely that this algorithm will make it to the Nomenclature Hall of Fame.

![Diagram showing wound-wait deadlock prevention algorithm](image)

If an old process wants a resource held by a young one, the old process preempts the young one, whose transaction is then killed, as depicted in Fig. The young one probably starts up again immediately, and tries to acquire the resource, leading to Fig. (b), forcing it to wait. Contrast this algorithm with wait-die. There, if an oldtimer wants a resource held by a young squirt, the oldtimer waits politely. However, if the young one wants a resource held by the old one, the young one is killed. It will undoubtedly start up again and be killed again. This cycle may go on many times before the old one releases the resource. Wound-wait does not have this nasty property.
UNIT IV
Processes and Processors in Distributed Systems

In the preceding two chapters, we have looked at two related topics, communication and synchronization in distributed systems. In this chapter we will switch to a different subject: processes. Although processes are also an important concept in uniprocessor systems, in this chapter we will emphasize aspects of process management that are usually not studied in the context of classical operating systems. In particular, we will look at how the existence of multiple processors is dealt with.

In many distributed systems, it is possible to have multiple threads of control within a process. This ability provides some important advantages, but also introduces various problems. We will study these issues first. Then we come to the subject of how the processors and processes are organized and see that several different models are possible. Then we will look at processor allocation and scheduling in distributed systems. Finally, we consider two special kinds of distributed systems, fault-tolerant systems and real-time systems.

4.1. THREADS

In most traditional operating systems, each process has an address space and a single thread of control. In fact, that is almost the definition of a process. Nevertheless, there are frequently situations in which it is desirable to have multiple threads of control sharing one address space but running in quasi-parallel, as though they were separate processes (except for the shared address space). In this section we will discuss these situations and their implications.

4.1.1. Introduction to Threads

Consider, for example, a file server that occasionally has to block waiting for the disk. If the server had multiple threads of control, a second thread could run while the first one was sleeping. The net result would be a higher throughput and better performance. It is not possible to achieve this goal by creating two independent server processes because they must share a common buffer cache, which requires them to be in the same address space. Thus a new mechanism is needed, one that historically was not found in single-processor operating systems.

![Fig. 4-1. (a) three processes with one thread each. (b) One process with three threads.](image-url)

In Fig. 4-1 (a) we see a machine with three processes. Each process has its own program counter, its own stack, its own register set, and its own address space. The processes have nothing to do with each other, except that they may be able to communicate through the system's interprocess communication primitives, such as semaphores, monitors, or messages. In Fig. 4-1(b) we see another machine, with one process. Only this process contains multiple threads of control, usually just called threads, or sometimes lightweight processes. In many respects, threads are like little mini-processes. Each thread runs strictly sequentially and has its own program counter and stack to keep track of where it is. Threads share the CPU just as processes do: first one thread
runs, then another does (timesharing). Only on a multiprocessor do they actually run in parallel. Threads can create child threads and can block waiting for system calls to complete, just like regular processes. While one thread is blocked, another thread in the same process can run, in exactly the same way that when one process blocks, another process in the same machine can run. The analogy: thread is to process as process is to machine, holds in many ways.

Different threads in a process are not quite as independent as different processes, however. All threads have exactly the same address space, which means that they also share the same global variables. Since every thread can access every virtual address, one thread can read, write, or even completely wipe out another thread's stack. There is no protection between threads because (1) it is impossible, and (2) it should not be necessary. Unlike different processes, which may be from different users and which may be hostile to one another, a process is always owned by a single user, who has presumably created multiple threads so that they can cooperate, not fight. In addition to sharing an address space, all the threads share the same set of open files, child processes, timers, and signals, etc. as shown in Fig. 4-2. Thus the organization of Fig. 4-1(a) would be used when the three processes are essentially unrelated, whereas Fig. 4-1(b) would be appropriate when the three threads are actually part of the same job and are actively and closely cooperating with each other.

![Fig. 4-2. Per thread and per process concepts.](image)

Like traditional processes (i.e., processes with only one thread), threads can be in any one of several states: running, blocked, ready, or terminated. A running thread currently has the CPU and is active. A blocked thread is waiting for another thread to unblock it (e.g., on a semaphore). A ready thread is scheduled to run, and will as soon as its turn comes up. Finally, a terminated thread is one that has exited, but which has not yet been collected by its parent (in UNIX terms, the parent thread has not yet done a WAIT).

**4.1.2. Thread Usage**

Threads were invented to allow parallelism to be combined with sequential execution and blocking system calls. Consider our file server example again. One possible organization is shown in Fig. 4-3(a). Here one thread, the **dispatcher**, reads incoming requests for work from the system mailbox. After examining the request, it chooses an idle (i.e., blocked) **worker thread** and hands it the request, possibly by writing a pointer to the message into a special word associated with each thread. The dispatcher then wakes up the sleeping worker (e.g., by doing an UP on the semaphore on which it is sleeping).
When the worker wakes up, it checks to see if the request can be satisfied from the shared block cache, to which all threads have access. If not, it sends a message to the disk to get the needed block (assuming it is a READ) and goes to sleep awaiting completion of the disk operation. The scheduler will now be invoked and another thread will be started, possibly the dispatcher, in order to acquire more work, or possibly another worker that is now ready to run.

Consider how the file server could be written in the absence of threads. One possibility is to have it operate as a single thread. The main loop of the file server gets a request, examines it, and carries it out to completion before getting the next one. While waiting for the disk, the server is idle and does not process any other requests. If the file server is running on a dedicated machine, as is commonly the case, the CPU is simply idle while the file server is waiting for the disk. The net result is that many fewer requests/sec can be processed. Thus threads gain considerable performance, but each thread is programmed sequentially, in the usual way.

So far we have seen two possible designs: a multithreaded file server and a single-threaded file server. Suppose that threads are not available but the system designers find the performance loss due to single threading unacceptable. A third possibility is to run the server as a big finite-state machine. When a request comes in, the one and only thread examines it. If it can be satisfied from the cache, fine, but if not, a message must be sent to the disk.

However, instead of blocking, it records the state of the current request in a table and then goes and gets the next message. The next message may either be a request for new work or a reply from the disk about a previous operation. If it is new work, that work is started. If it is a reply from the disk, the relevant information is fetched from the table and the reply processed. Since it is not permitted to send a message and block waiting for a reply here, RPC cannot be used. The primitives must be nonblocking calls to `send` and `receive`.

In this design, the "sequential process" model that we had in the first two cases is lost. The state of the computation must be explicitly saved and restored in the table for every message sent and received. In effect, we are simulating the threads and their stacks the hard way. The process is being operated as a finite-state machine that gets an event and then reacts to it, depending on what is in it.

It should now be clear what threads have to offer. They make it possible to retain the idea of sequential processes that make blocking system calls (e.g., RPC to talk to the disk) and still achieve parallelism. Blocking system calls make programming easier and parallelism improves performance. The single-threaded server retains the ease of blocking system calls, but gives up performance. The finite-state machine approach achieves high performance through parallelism, but uses nonblocking calls and thus is hard to program. These models are summarized in Fig. 4-4.

![Fig. 4-3. Three organizations of threads in a process. (a) Dispatcher/worker model. (b) Team model. (c) Pipeline model.](image)
<table>
<thead>
<tr>
<th>Model</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>Parallelism, blocking system calls</td>
</tr>
<tr>
<td>Single-thread process</td>
<td>No parallelism, blocking system calls</td>
</tr>
<tr>
<td>Finite-state machine</td>
<td>Parallelism, nonblocking system calls</td>
</tr>
</tbody>
</table>

**Fig. 4-4.** Three ways to construct a server.

The dispatcher structure of Fig. 4-3(a) is not the only way to organize a multithreaded process. The **team** model of Fig. 4-3(b) is also a possibility. Here all the threads are equals, and each gets and processes its own requests. There is no dispatcher. Sometimes work comes in that a thread cannot handle, especially if each thread is specialized to handle a particular kind of work. In this case, a job queue can be maintained, with pending work kept in the job queue. With this organization, a thread should check the job queue before looking in the system mailbox.

Threads can also be organized in the **pipeline** model of Fig. 4-3(c). In this model, the first thread generates some data and passes them on to the next thread for processing. The data continue from thread to thread, with processing going on at each step. Although this is not appropriate for file servers, for other problems, such as the producer-consumer, it may be a good choice. Pipelining is widely used in many areas of computer systems, from the internal structure of RISC CPUs to UNIX command lines.

Threads are frequently also useful for clients. For example, if a client wants a file to be replicated on multiple servers, it can have one thread talk to each server. Another use for client threads is to handle signals, such as interrupts from the keyboard (DEL or BREAK). Instead of letting the signal interrupt the process, one thread is dedicated full time to waiting for signals. Normally, it is blocked, but when a signal comes in, it wakes up and processes the signal. Thus using threads can eliminate the need for user-level interrupts.

Another argument for threads has nothing to do with RPC or communication. Some applications are easier to program using parallel processes, the producer-consumer problem for example. Whether the producer and consumer actually run in parallel is secondary. They are programmed that way because it makes the software design simpler. Since they must share a common buffer, having them in separate processes will not do. Threads fit the bill exactly here.

Finally, although we are not explicitly discussing the subject here, in a multiprocessor system, it is actually possible for the threads in a single address space to run in parallel, on different CPUs. This is, in fact, one of the major ways in which sharing is done on such systems. On the other hand, a properly designed program that uses threads should work equally well on a single CPU that timeshares the threads or on a true multiprocessor, so the software issues are pretty much the same either way.

### 4.1.3. Design Issues for Threads Packages

A set of primitives (e.g., library calls) available to the user relating to threads is called a **threads package**. In this section we will consider some of the issues concerned with the architecture and functionality of threads packages. In the next section we will consider how threads packages can be implemented.

The first issue we will look at is thread management. Two alternatives are possible here, static threads and dynamic threads. With a static design, the choice of how many threads there will be is made when the program is written or when it is compiled. Each thread is allocated a fixed stack. This approach is simple, but inflexible.
A more general approach is to allow threads to be created and destroyed on-the-fly during execution. The thread creation call usually specifies the thread's main program (as a pointer to a procedure) and a stack size, and may specify other parameters as well, for example, a scheduling priority. The call usually returns a thread identifier to be used in subsequent calls involving the thread. In this model, a process starts out with one (implicit) thread, but can create one or more threads as needed, and these can exit when finished.

Threads can be terminated in one of two ways. A thread can exit voluntarily when it finishes its job, or it can be killed from outside. In this respect, threads are like processes. In many situations, such as the file servers of Fig. 4-3, the threads are created immediately after the process starts up and are never killed.

Since threads share a common memory, they can, and usually do, use it for holding data that are shared among multiple threads, such as the buffers in a producer-consumer system. Access to shared data is usually programmed using critical regions, to prevent multiple threads from trying to access the same data at the same time. Critical regions are most easily implemented using semaphores, monitors, and similar constructions. One technique that is commonly used in threads packages is the mutex, which is a kind of watered-down semaphore. A mutex is always in one of two states, unlocked or locked. Two operations are defined on mutexes. The first one, LOCK, attempts to lock the mutex. If the mutex is unlocked, the LOCK succeeds and the mutex becomes locked in a single atomic action. If two threads try to lock the same mutex at exactly the same instant, an event that is possible only on a multiprocessor, on which different threads run on different CPUs, one of them wins and the other loses. A thread that attempts to lock a mutex that is already locked is blocked.

The UNLOCK operation unlocks a mutex. If one or more threads are waiting on the mutex, exactly one of them is released. The rest continue to wait.

Another operation that is sometimes provided is TRYLOCK, which attempts to lock a mutex. If the mutex is unlocked, TRYLOCK returns a status code indicating success. If, however, the mutex is locked, TRYLOCK does not block the thread. Instead, it returns a status code indicating failure.

Mutexes are like binary semaphores (i.e., semaphores that may only have the values 0 or 1). They are not like counting semaphores. Limiting them in this way makes them easier to implement.

Another synchronization feature that is sometimes available in threads packages is the condition variable, which is similar to the condition variable used for synchronization in monitors. Each condition variable is normally associated with a mutex at the time it is created. The difference between mutexes and condition variables is that mutexes are used for short-term locking, mostly for guarding the entry to critical regions. Condition variables are used for long-term waiting until a resource becomes available.

The following situation occurs all the time. A thread locks a mutex to gain entry to a critical region. Once inside the critical region, it examines system tables and discovers that some resource it needs is busy. If it simply locks a second mutex (associated with the resource), the outer mutex will remain locked and the thread holding the resource will not be able to enter the critical region to free it. Deadlock results. Unlocking the outer mutex lets other threads into the critical region, causing chaos, so this solution is not acceptable.

One solution is to use condition variables to acquire the resource, as shown in Fig. 4-5(a). Here, waiting on the condition variable is defined to perform the wait and unlock the mutex atomically. Later, when the thread holding the resource frees it, as shown in Fig. 4-5(b), it calls wakeup, which is defined to wakeup either exactly one thread or all the threads waiting on the specified condition variable. The use of WHILE instead of IF in Fig. 4-5(a) guards against the case that the thread is awakened but that someone else seizes the resource before the thread runs.
The need for the ability to wake up all the threads, rather than just one, is demonstrated in the reader-writer problem. When a writer finishes, it may choose to wake up pending writers or pending readers. If it chooses readers, it should wake them all up, not just one. Providing primitives for waking up exactly one thread and for waking up all the threads provides the needed flexibility.

The code of a thread normally consists of multiple procedures, just like a process. These may have local variables, global variables, and procedure parameters. Local variables and parameters do not cause any trouble, but variables that are global to a thread but not global to the entire program do.

As an example, consider the `errno` variable maintained by UNIX. When a process (or a thread) makes a system call that fails, the error code is put into `errno`. In Fig. 4-6, thread 1 executes the system call `ACCESS` to find out if it has permission to access a certain file. The operating system returns the answer in the global variable `errno`. After control has returned to thread 1, but before it has a chance to read `errno`, the scheduler decides that thread 1 has had enough CPU time for the moment and decides to switch to thread 2. Thread 2 executes an open call that fails, which causes `errno` to be overwritten and thread 1’s access code to be lost forever. When thread 1 starts up later, it will read the wrong value and behave incorrectly.

Various solutions to this problem are possible. One is to prohibit global variables altogether. However worthy this ideal may be, it conflicts with much existing software, such as UNIX. Another is to assign each thread its own private global variables, as shown in Fig. 4-7. In this way, each thread has its own private copy of `errno` and other global variables, so conflicts are avoided. In effect, this decision creates a new scoping level, variables visible to all the procedures of a thread, in addition to the existing scoping levels of variables visible only to one procedure and variables visible everywhere in the program.
Accessing the private global variables is a bit tricky, however, since most programming languages have a way of expressing local variables and global variables, but not intermediate forms. It is possible to allocate a chunk of memory for the globals and pass it to each procedure in the thread, as an extra parameter. While hardly an elegant solution, it works.

Alternatively, new library procedures can be introduced to create, set, and read these thread-wide global variables. The first call might look like this:

```c
create_global("bufptr");
```

It allocates storage for a pointer called *bufptr* on the heap or in a special storage area reserved for the calling thread. No matter where the storage is allocated, only the calling thread has access to the global variable. If another thread creates a global variable with the same name, it gets a different storage location that does not conflict with the existing one.

Two calls are needed to access global variables: one for writing them and the other for reading them. For writing, something like

```c
set_global("bufptr", &buf);
```

will do. It stores the value of a pointer in the storage location previously created by the call to *create_global*. To read a global variable, the call might look like

```c
bufptr = read_global("bufptr");
```

This call returns the address stored in the global variable, so the data value can be accessed.

Our last design issue relating to threads is scheduling. Threads can be scheduled using various scheduling algorithms, including priority, round robin, and others. Threads packages often provide calls to give the user the ability to specify the scheduling algorithm and set the priorities, if any.

### 4.1.4. Implementing a Threads Package

There are two main ways to implement a threads package: in user space and in the kernel. The choice is moderately controversial, and a hybrid implementation is also possible. In this section we will describe these methods, along with their advantages and disadvantages.

## Implementing Threads in User Space

The first method is to put the threads package entirely in user space. The kernel knows nothing about them. As far as the kernel is concerned, it is managing ordinary, single-threaded processes. The first, and most obvious, advantage is that a user-level threads package can be
implemented on an operating system that does not support threads. For example, UNIX originally did not support threads, but various user-space threads packages were written for it.

All of these implementations have the same general structure, which is illustrated in Fig. 4-8(a). The threads run on top of a runtime system, which is a collection of procedures that manage threads. When a thread executes a system call, goes to sleep, performs an operation on a semaphore or mutex, or otherwise does something that may cause it to be suspended, it calls a runtime system procedure. This procedure checks to see if the thread must be suspended. If so, it stores the thread's registers (i.e., its own) in a table, looks for an unblocked thread to run, and reloads the machine registers with the new thread's saved values. As soon as the stack pointer and program counter have been switched, the new thread comes to life again automatically. If the machine has an instruction to store all the registers and another one to load them all, the entire thread switch can be done in a handful of instructions. Doing thread switching like this is at least an order of magnitude faster than trapping to the kernel, and is a strong argument in favor of user-level threads packages.

---

![Fig. 4-8. (a) A user-level threads package. (b) A threads packaged managed by the kernel.](image_url)

User-level threads also have other advantages. They allow each process to have its own customized scheduling algorithm. For some applications, for example, those with a garbage collector thread, not having to worry about a thread being stopped at an inconvenient moment is a plus. They also scale better, since kernel threads invariably require some table space and stack space in the kernel, which can be a problem if there are a very large number of threads.

Despite their better performance, user-level threads packages have some major problems. First among these is the problem of how blocking system calls are implemented. Suppose that a thread reads from an empty pipe or does something else that will block. Letting the thread actually make the system call is unacceptable, since this will stop all the threads. One of the main goals of having threads in the first place was to allow each one to use blocking calls, but to prevent one blocked thread from affecting the others. With blocking system calls, this goal cannot be achieved.

The system calls could all be changed to be nonblocking (e.g., a read on a empty pipe could just fail), but requiring changes to the operating system is unattractive. Besides, one of the arguments for user-level threads was precisely that they could run with existing operating systems. In addition, changing the semantics of READ will require changes to many user programs.

Another alternative is possible in the event that it is possible to tell in advance if a call will block. In some versions of UNIX, a call SELECT exists, which allows the caller to tell whether a pipe is empty, and so on. When this call is present, the library procedure read can be replaced with a
new one that first does a SELECT call and then only does the READ call if it is safe (i.e., will not block). If the read call will block, the call is not made. Instead, another thread is run. The next time the runtime system gets control, it can check again to see if the READ is now safe. This approach requires rewriting parts of the system call library, is inefficient and inelegant, but there is little choice. The code placed around the system call to do the checking is called a jacket.

Somewhat analogous to the problem of blocking system calls is the problem of page faults. If a thread causes a page fault, the kernel, not even knowing about the existence of threads, naturally blocks the entire process until the needed page has been fetched, even though other threads might be runnable.

Another problem with user-level thread packages is that if a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU. Within a single process, there are no clock interrupts, making round-robin scheduling impossible. Unless a thread enters the runtime system of its own free will, the scheduler will never get a chance.

An area in which the absence of clock interrupts is crucial is synchronization. It is common in distributed applications for one thread to initiate an activity to which another thread must respond and then just sit in a tight loop testing whether the response has happened. This situation is called a spin lock or busy waiting. This approach is especially attractive when the response is expected quickly and the cost of using semaphores is high. If threads are rescheduled automatically every few milliseconds based on clock interrupts, this approach works fine. However, if threads run until they block, this approach is a recipe for deadlock.

One possible solution to the problem of threads running forever is to have the runtime system request a clock signal (interrupt) once a second to give it control, but this too is crude and messy to program. Periodic clock interrupts at a higher frequency are not always possible, and even if they are, the total overhead may be substantial. Furthermore, a thread might also need a clock interrupt, interfering with the runtime system’s use of the clock.

Another, and probably most devastating argument against user-level threads is that programmers generally want threads in applications where the threads block often, as, for example, in a multithreaded file server. These threads are constantly making system calls. Once a trap has occurred to the kernel to carry out the system call, it is hardly any more work for the kernel to switch threads if the old one has blocked, and having the kernel do this eliminates the need for constantly checking to see if system calls are safe. For applications that are essentially entirely CPU bound and rarely block, what is the point of having threads at all? No one would seriously propose to compute the first \( n \) prime numbers or play chess using threads because there is nothing to be gained by doing it that way.

### Implementing Threads in the Kernel

Now let us consider having the kernel know about and manage the threads. No runtime system is needed, as shown in Fig. 4-8(b). Instead, when a thread wants to create a new thread or destroy an existing thread, it makes a kernel call, which then does the creation or destruction.

To manage all the threads, the kernel has one table per process with one entry per thread. Each entry holds the thread’s registers, state, priority, and other information. The information is the same as with user-level threads, but it is now in the kernel instead of in user space (inside the runtime system). This information is also the same information that traditional kernels maintain about each of their single-threaded processes, that is, the process state.

All calls that might block a thread, such as interthread synchronization using semaphores, are implemented as system calls, at considerably greater cost than a call to a runtime system procedure. When a thread blocks, the kernel, at its option, can run either another thread from the
same process (if one is ready), or a thread from a different process. With user-level threads, the runtime system keeps running threads from its own process until the kernel takes the CPU away from it (or there are no ready threads left to run).

Due to the relatively greater cost of creating and destroying threads in the kernel, some systems take an environmentally correct approach and recycle their threads. When a thread is destroyed, it is marked as not runnable, but its kernel data structures are not otherwise affected. Later, when a new thread must be created, an old thread is reactivated, saving some overhead. Thread recycling is also possible for user-level threads, but since the thread management overhead is much smaller, there is less incentive to do this.

Kernel threads do not require any new, nonblocking system calls, nor do they lead to deadlocks when spin locks are used. In addition, if one thread in a process causes a page fault, the kernel can easily run another thread while waiting for the required page to be brought in from the disk (or network). Their main disadvantage is that the cost of a system call is substantial, so if thread operations (creation, deletion, synchronization, etc.) are common, much more overhead will be incurred.

In addition to the various problems specific to user threads and those specific to kernel threads, there are some other problems that occur with both of them. For example, many library procedures are not reentrant. For example, sending a message over the network may well be programmed to assemble the message in a fixed buffer first, then to trap to the kernel to send it. What happens if one thread has assembled its message in the buffer, then a clock interrupt forces a switch to a second thread that immediately overwrites the buffer with its own message? Similarly, after a system call completes, a thread switch may occur before the previous thread has had a chance to read out the error status (errno, as discussed above). Also, memory allocation procedures, such as the UNIX malloc, fiddle with crucial tables without bothering to set up and use protected critical regions, because they were written for single-threaded environments where that was not necessary. Fixing all these problems properly effectively means rewriting the entire library.

A different solution is to provide each procedure with a jacket that locks a global semaphore or mutex when the procedure is started. In this way, only one thread may be active in the library at once. Effectively, the entire library becomes a big monitor.

Signals also present difficulties. Suppose that one thread wants to catch a particular signal (say, the user hitting the DEL key), and another thread wants this signal to terminate the process. This situation can arise if one or more threads run standard library procedures and others are user-written. Clearly, these wishes are incompatible. In general, signals are difficult enough to manage in a single-threaded environment. Going to a multithreaded environment does not make them any easier to handle. Signals are typically a per-process concept, not a per-thread concept, especially if the kernel is not even aware of the existence of the threads.

### Scheduler Activations

Various researchers have attempted to combine the advantage of user threads (good performance) with the advantage of kernel threads (not having to use a lot of tricks to make things work). Below we will describe one such approach devised by Anderson et al. (1991), called scheduler activations. Related work is discussed by Edler et al. (1988) and Scott et al. (1990).

The goals of the scheduler activation work are to mimic the functionality of kernel threads, but with the better performance and greater flexibility usually associated with threads packages implemented in user space. In particular, user threads should not have to make special nonblocking system calls or check in advance if it is safe to make certain system calls.
Nevertheless, when a thread blocks on a system call or on a page fault, it should be possible to run other threads within the same process, if any are ready.

Efficiency is achieved by avoiding unnecessary transitions between user and kernel space. If a thread blocks on a local semaphore, for example, there is no reason to involve the kernel. The user-space runtime system can block the synchronizing thread and schedule a new one by itself.

When scheduler activations are used, the kernel assigns a certain number of virtual processors to each process and lets the (user-space) runtime system allocate threads to processors. This mechanism can also be used on a multiprocessor where the virtual processors may be real CPUs. The number of virtual processors allocated to a process is initially one, but the process can ask for more and can also return processors it no longer needs. The kernel can take back virtual processors already allocated to assign them to other, more needy, processes.

The basic idea that makes this scheme work is that when the kernel knows that a thread has blocked (e.g., by its having executed a blocking system call or caused a page fault), the kernel notifies the process’ runtime system, passing as parameters on the stack the number of the thread in question and a description of the event that occurred. The notification happens by having the kernel activate the runtime system at a known starting address, roughly analogous to a signal in UNIX. This mechanism is called an upcall.

Once activated like this, the runtime system can reschedule its threads, typically by marking the current thread as blocked and taking another thread from the ready list, setting up its registers, and restarting it. Later, when the kernel learns that the original thread can run again (e.g., the pipe it was trying to read from now contains data, or the page it faulted over has been brought in from disk), it makes another upcall to the runtime system to inform it of this event. The runtime system, at its own discretion, can either restart the blocked thread immediately, or put it on the ready list to be run later.

When a hardware interrupt occurs while a user thread is running, the interrupted CPU switches into kernel mode. If the interrupt is caused by an event not of interest to the interrupted process, such as completion of another process’ I/O, when the interrupt handler has finished, it puts the interrupted thread back in the state it was in before the interrupt. If, however, the process is interested in the interrupt, such as the arrival of a page needed by one of the process’ threads, the interrupted thread is not restarted. Instead, the interrupted thread is suspended and the runtime system started on that virtual CPU, with the state of the interrupted thread on the stack. It is then up to the runtime system to decide which thread to schedule on that CPU: the interrupted one, the newly ready one, or some third choice.

Although scheduler activations solve the problem of how to pass control to an unblocked thread in a process one of whose threads has just blocked, it creates a new problem. The new problem is that an interrupted thread might have been executing a semaphore operation at the time it was suspended, in which case it would probably be holding a lock on the ready list. If the runtime system started by the upcall then tries to acquire this lock itself, in order to put a newly ready thread on the list, it will fail to acquire the lock and a deadlock will ensue. The problem can be solved by keeping track of when threads are or are not in critical regions, but the solution is complicated and hardly elegant.

Another objection to scheduler activations is the fundamental reliance on upcalls, a concept that violates the structure inherent in any layered system. Normally, layer \( n \) offers certain services that layer \( n+1 \) can call on, but layer \( n \) may not call procedures in layer \( n+1 \).

4.2. SYSTEM MODELS

Processes run on processors. In a traditional system, there is only one processor, so the question of how the processor should be used does not come up. In a distributed system, with multiple processors, it is a major design issue. The processors in a distributed system can be
organized in several ways. In this section we will look at two of the principal ones, the workstation model and the processor pool model, and a hybrid form encompassing features of each one. These models are rooted in fundamentally different philosophies of what a distributed system is all about.

4.2.1. The Workstation Model

The workstation model is straightforward: the system consists of workstations (high-end personal computers) scattered throughout a building or campus and connected by a high-speed LAN, as shown in Fig. 4-10. Some of the workstations may be in offices, and thus implicitly dedicated to a single user, whereas others may be in public areas and have several different users during the course of a day. In both cases, at any instant of time, a workstation either has a single user logged into it, and thus has an "owner" (however temporary), or it is idle.

![Figure 4-10](image)

Fig. 4-10. A network of personal workstations, each with a local file system.

In some systems the workstations have local disks and in others they do not. The latter are universally called **diskless workstations**, but the former are variously known as **diskful workstations**, or **disky workstations**, or even stranger names. If the workstations are diskless, the file system must be implemented by one or more remote file servers. Requests to read and write files are sent to a file server, which performs the work and sends back the replies.

Diskless workstations are popular at universities and companies for several reasons, not the least of which is price. Having a large number of workstations equipped with small, slow disks is typically much more expensive than having one or two file servers equipped with huge, fast disks and accessed over the LAN.

A second reason that diskless workstations are popular is their ease of maintenance. When a new release of some program, say a compiler, comes out, the system administrators can easily install it on a small number of file servers in the machine room. Installing it on dozens or hundreds of machines all over a building or campus is another matter entirely. Backup and hardware maintenance is also simpler with one centrally located 5-gigabyte disk than with fifty 100-megabyte disks scattered over the building.

Another point against disks is that they have fans and make noise. Many people find this noise objectionable and do not want it in their office.

Finally, diskless workstations provide symmetry and flexibility. A user can walk up to any workstation in the system and log in. Since all his files are on the file server, one diskless workstation is as good as another. In contrast, when all the files are stored on local disks, using someone else's workstation means that you have easy access to his files, but getting to your own requires extra effort, and is certainly different from using your own workstation.

When the workstations have private disks, these disks can be used in one of at least four ways:

1. Paging and temporary files.
2. Paging, temporary files, and system binaries.
3. Paging, temporary files, system binaries, and file caching.
4. Complete local file system.

The first design is based on the observation that while it may be convenient to keep all the user files on the central file servers (to simplify backup and maintenance, etc.) disks are also needed for paging (or swapping) and for temporary files. In this model, the local disks are used only for paging and files that are temporary, unshared, and can be discarded at the end of the login session. For example, most compilers consist of multiple passes, each of which creates a temporary file read by the next pass. When the file has been read once, it is discarded. Local disks are ideal for storing such files.

The second model is a variant of the first one in which the local disks also hold the binary (executable) programs, such as the compilers, text editors, and electronic mail handlers. When one of these programs is invoked, it is fetched from the local disk instead of from a file server, further reducing the network load. Since these programs rarely change, they can be installed on all the local disks and kept there for long periods of time. When a new release of some system program is available, it is essentially broadcast to all machines. However, if that machine happens to be down when the program is sent, it will miss the program and continue to run the old version. Thus some administration is needed to keep track of who has which version of which program.

A third approach to using local disks is to use them as explicit caches (in addition to using them for paging, temporaries, and binaries). In this mode of operation, users can download files from the file servers to their own disks, read and write them locally, and then upload the modified ones at the end of the login session. The goal of this architecture is to keep long-term storage centralized, but reduce network load by keeping files local while they are being used. A disadvantage is keeping the caches consistent. What happens if two users download the same file and then each modifies it in different ways? This problem is not easy to solve, and we will discuss it in detail later in the book.

Fourth, each machine can have its own self-contained file system, with the possibility of mounting or otherwise accessing other machines' file systems. The idea here is that each machine is basically self-contained and that contact with the outside world is limited. This organization provides a uniform and guaranteed response time for the user and puts little load on the network. The disadvantage is that sharing is more difficult, and the resulting system is much closer to a network operating system than to a true transparent distributed operating system.

The one diskless and four diskful models we have discussed are summarized in Fig. 4-11. The progression from top to bottom in the figure is from complete dependence on the file servers to complete independence from them.

The advantages of the workstation model are manifold and clear. The model is certainly easy to understand. Users have a fixed amount of dedicated computing power, and thus guaranteed response time. Sophisticated graphics programs can be very fast, since they can have direct access to the screen. Each user has a large degree of autonomy and can allocate his workstation's resources as he sees fit. Local disks add to this independence, and make it possible to continue working to a lesser or greater degree even in the face of file server crashes.

However, the model also has two problems. First, as processor chips continue to get cheaper, it will soon become economically feasible to give each user first 10 and later 100 CPUs. Having 100 workstations in your office makes it hard to see out the window. Second, much of the time users are not using their workstations, which are idle, while other users may need extra computing capacity and cannot get it. From a system-wide perspective, allocating resources in such a way that some users have resources they do not need while other users need these resources badly is inefficient.

<table>
<thead>
<tr>
<th>Dependence on</th>
<th>Disk usage</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
</table>

Prepared By Department Of CSE.  Subject: Distributed Systems
<table>
<thead>
<tr>
<th>file servers ↑</th>
<th>(Diskless)</th>
<th>Low cost, easy hardware and software maintenance, symmetry and flexibility</th>
<th>Heavy network usage; file servers may become bottlenecks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paging, scratch files</td>
<td>Reduces network load over diskless case</td>
<td>Higher cost due to large number of disks needed</td>
<td></td>
</tr>
<tr>
<td>Paging, scratch files, binaries</td>
<td>Reduces network load even more</td>
<td>Higher cost; additional complexity of updating the binaries</td>
<td></td>
</tr>
<tr>
<td>Paging, scratch files, binaries, file caching</td>
<td>Still lower network load; reduces load on file servers as well</td>
<td>Higher cost; cache consistency problems</td>
<td></td>
</tr>
<tr>
<td>Full local file system</td>
<td>Hardly any network load; eliminates need for file servers</td>
<td>Loss of transparency</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 4-11. Disk usage on workstations.

The first problem can be addressed by making each workstation a personal multiprocessor. For example, each window on the screen can have a dedicated CPU to run its programs. Preliminary evidence from some early personal multiprocessors such as the DEC Firefly, suggest, however, that the mean number of CPUs utilized is rarely more than one, since users rarely have more than one active process at once. Again, this is an inefficient use of resources, but as CPUs get cheaper and cheaper as the technology improves, wasting them will become less of a sin.

4.2.2. Using Idle Workstations

The second problem, idle workstations, has been the subject of considerable research, primarily because many universities have a substantial number of personal workstations, some of which are idle (an idle workstation is the devil's playground?). Measurements show that even at peak periods in the middle of the day, often as many as 30 percent of the workstations are idle at any given moment. In the evening, even more are idle. A variety of schemes have been proposed for using idle or otherwise underutilized workstations (Litzkow et al., 1988; Nichols, 1987; and Theimer et al., 1985). We will describe the general principles behind this work in this section.

The earliest attempt to allow idle workstations to be utilized was the `rsh` program that comes with Berkeley UNIX. This program is called by

```
rsh machine command
```

in which the first argument names a machine and the second names a command to run on it. What `rsh` does is run the specified command on the specified machine. Although widely used, this program has several serious flaws. First, the user must tell which machine to use, putting the full burden of keeping track of idle machines on the user. Second, the program executes in the environment of the remote machine, which is usually different from the local environment. Finally, if someone should log into an idle machine on which a remote process is running, the process continues to run and the newly logged-in user either has to accept the lower performance or find another machine.

The research on idle workstations has centered on solving these problems. The key issues are:

1. How is an idle workstation found?
2. How can a remote process be run transparently?
3. What happens if the machine's owner comes back?
Let us consider these three issues, one at a time.

How is an idle workstation found? To start with, what is an idle workstation? At first glance, it might appear that a workstation with no one logged in at the console is an idle workstation, but with modern computer systems things are not always that simple. In many systems, even with no one logged in there may be dozens of processes running, such as clock daemons, mail daemons, news daemons, and all manner of other daemons. On the other hand, a user who logs in when arriving at his desk in the morning, but otherwise does not touch the computer for hours, hardly puts any additional load on it. Different systems make different decisions as to what "idle" means, but typically, if no one has touched the keyboard or mouse for several minutes and no user-initiated processes are running, the workstation can be said to be idle. Consequently, there may be substantial differences in load between one idle workstation and another, due, for example, to the volume of mail coming into the first one but not the second.

The algorithms used to locate idle workstations can be divided into two categories: server driven and client driven. In the former, when a workstation goes idle, and thus becomes a potential compute server, it announces its availability. It can do this by entering its name, network address, and properties in a registry file (or data base), for example. Later, when a user wants to execute a command on an idle workstation, he types something like

\[
\text{remote command}
\]

and the \textit{remote} program looks in the registry to find a suitable idle workstation. For reliability reasons, it is also possible to have multiple copies of the registry.

An alternative way for the newly idle workstation to announce the fact that it has become unemployed is to put a broadcast message onto the network. All other workstations then record this fact. In effect, each machine maintains its own private copy of the registry. The advantage of doing it this way is less overhead in finding an idle workstation and greater redundancy. The disadvantage is requiring all machines to do the work of maintaining the registry.

Whether there is one registry or many, there is a potential danger of race conditions occurring. If two users invoke the \textit{remote} command simultaneously, and both of them discover that the same machine is idle, they may both try to start up processes there at the same time. To detect and avoid this situation, the \textit{remote} program can check with the idle workstation, which, if still free, removes itself from the registry and gives the go-ahead sign. At this point, the caller can send over its environment and start the remote process, as shown in Fig. 4-12.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig4-12.png}
\caption{A registry-based algorithm for finding and using idle workstations.}
\end{figure}
The other way to locate idle workstations is to use a client-driven approach. When \textit{remote} is invoked, it broadcasts a request saying what program it wants to run, how much memory it needs, whether or not floating point is needed, and so on. These details are not needed if all the workstations are identical, but if the system is heterogeneous and not every program can run on every workstation, they are essential. When the replies come back, \textit{remote} picks one and sets it up. One nice twist is to have "idle" workstations delay their responses slightly, with the delay being proportional to the current load. In this way, the reply from the least heavily loaded machine will come back first and be selected.

Finding a workstation is only the first step. Now the process has to be run there. Moving the code is easy. The trick is to set up the remote process so that it sees the same environment it would have locally, on the \textbf{home workstation}, and thus carries out the same computation it would have locally.

To start with, it needs the same view of the file system, the same working directory, and the same environment variables (shell variables), if any. After these have been set up, the program can begin running. The trouble starts when the first system call, say a READ, is executed. What should the kernel do? The answer depends very much on the system architecture. If the system is diskless, with all the files located on file servers, the kernel can just send the request to the appropriate file server, the same way the home machine would have done had the process been running there. On the other hand, if the system has local disks, each with a complete file system, the request has to be forwarded back to the home machine for execution.

Some system calls must be forwarded back to the home machine no matter what, even if all the machines are diskless. For example, reads from the keyboard and writes to the screen can never be carried out on the remote machine. However, other system calls must be done remotely under all conditions. For example, the UNIX system calls \texttt{SBRK} (adjust the size of the data segment), \texttt{NICE} (set CPU scheduling priority), and \texttt{PROFIL} (enable profiling of the program counter) cannot be executed on the home machine. In addition, all system calls that query the state of the machine have to be done on the machine on which the process is actually running. These include asking for the machine's name and network address, asking how much free memory it has, and so on.

System calls involving time are a problem because the clocks on different machines may not be synchronized. In Chap. 3, we saw how hard it is to achieve synchronization. Using the time on the remote machine may cause programs that depend on time, like \texttt{make}, to give incorrect results. Forwarding all time-related calls back to the home machine, however, introduces delay, which also causes problems with time.

To complicate matters further, certain special cases of calls which normally might have to be forwarded back, such as creating and writing to a temporary file, can be done much more efficiently on the remote machine. In addition, mouse tracking and signal propagation have to be thought out carefully as well. Programs that write directly to hardware devices, such as the screen's frame buffer, diskette, or magnetic tape, cannot be run remotely at all. All in all, making programs run on remote machines as though they were running on their home machines is possible, but it is a complex and tricky business.

The final question on our original list is what to do if the machine's owner comes back (i.e., somebody logs in or a previously inactive user touches the keyboard or mouse). The easiest thing is to do nothing, but this tends to defeat the idea of "personal" workstations. If other people can run programs on your workstation at the same time that you are trying to use it, there goes your guaranteed response.

Another possibility is to kill off the intruding process. The simplest way is to do this abruptly and without warning. The disadvantage of this strategy is that all work will be lost and the file system may be left in a chaotic state. A better way is to give the process fair warning, by sending it
a signal to allow it to detect impending doom, and shut down gracefully (write edit buffers to the disk, close files, and so on). If it has not exited within a few seconds, it is then terminated. Of course, the program must be written to expect and handle this signal, something most existing programs definitely are not.

A completely different approach is to migrate the process to another machine, either back to the home machine or to yet another idle workstation. Migration is rarely done in practice because the actual mechanism is complicated. The hard part is not moving the user code and data, but finding and gathering up all the kernel data structures relating to the process that is leaving. For example, it may have open files, running timers, queued incoming messages, and other bits and pieces of information scattered around the kernel. These must all be carefully removed from the source machine and successfully reinstalled on the destination machine. There are no theoretical problems here, but the practical engineering difficulties are substantial. For more information, see (Artsy and Finkel, 1989; Doughs and Ousterhout, 1991; and Zayas, 1987).

In both cases, when the process is gone, it should leave the machine in the same state in which it found it, to avoid disturbing the owner. Among other items, this requirement means that not only must the process go, but also all its children and their children. In addition, mailboxes, network connections, and other system-wide data structures must be deleted, and some provision must be made to ignore RPC replies and other messages that arrive for the process after it is gone. If there is a local disk, temporary files must be deleted, and if possible, any files that had to be removed from its cache restored.

4.2.3. The Processor Pool Model

Although using idle workstations adds a little computing power to the system, it does not address a more fundamental issue: What happens when it is feasible to provide 10 or 100 times as many CPUs as there are active users? One solution, as we saw, is to give everyone a personal multiprocessor. However this is a somewhat inefficient design.

An alternative approach is to construct a processor pool, a rack full of cpus in the machine room, which can be dynamically allocated to users on demand. The processor pool approach is illustrated in Fig. 4-13. Instead of giving users personal workstations, in this model they are given high-performance graphics terminals, such as X terminals (although small workstations can also be used as terminals). This idea is based on the observation that what many users really want is a high-quality graphical interface and good performance. Conceptually, it is much closer to traditional timesharing than to the personal computer model, although it is built with modern technology (low-cost microprocessors).

The motivation for the processor pool idea comes from taking the diskless workstation idea a step further. If the file system can be centralized in a small number of file servers to gain
economies of scale, it should be possible to do the same thing for compute servers. By putting all the CPUs in a big rack in the machine room, power supply and other packaging costs can be reduced, giving more computing power for a given amount of money. Furthermore, it permits the use of cheaper X terminals (or even ordinary ASCII terminals), and decouples the number of users from the number of workstations. The model also allows for easy incremental growth. If the computing load increases by 10 percent, you can just buy 10 percent more processors and put them in the pool.

In effect, we are converting all the computing power into "idle workstations" that can be accessed dynamically. Users can be assigned as many CPUs as they need for short periods, after which they are returned to the pool so that other users can have them. There is no concept of ownership here: all the processors belong equally to everyone.

The biggest argument for centralizing the computing power in a processor pool comes from queueing theory. A queueing system is a situation in which users generate random requests for work from a server. When the server is busy, the users queue for service and are processed in turn. Common examples of queueing systems are bakeries, airport check-in counters, supermarket check-out counters, and numerous others. The bare basics are depicted in Fig. 4-14.

Queueing systems are useful because it is possible to model them analytically. Let us call the total input rate $\lambda$ requests per second, from all the users combined. Let us call the rate at which the server can process requests $\mu$. For stable operation, we must have $\mu > \lambda$. If the server can handle 100 requests/sec, but the users continuously generate 110 requests/sec, the queue will grow without bound. (Small intervals in which the input rate exceeds the service rate are acceptable, provided that the mean input rate is lower than the service rate and there is enough buffer space.)

It can be proven (Kleinrock, 1974) that the mean time between issuing a request and getting a complete response, $T$, is related to $\lambda$ and $\mu$ by the formula

$$T = \frac{1}{\mu - \lambda}$$

As an example, consider a file server that is capable of handling as many as 50 requests/sec but which only gets 40 requests/sec. The mean response time will be $1/10$ sec or 100 msec. Note that when $\lambda$ goes to 0 (no load), the response time of the file server does not go to 0, but to $1/50$ sec or 20 msec. The reason is obvious once it is pointed out. If the file server can process only 50 requests/sec, it must take 20 msec to process a single request, even in the absence of any competition, so the response time, which includes the processing time, can never go below 20 msec.

Suppose that we have $n$ personal multiprocessors, each with some number of CPUs, and each one forms a separate queueing system with request arrival rate $\lambda$ and CPU processing rate $\mu$. The mean response time, $T$, will be as given above. Now consider what happens if we scoop up all the
CPUs and place them in a single processor pool. Instead of having \( n \) small queueing systems running in parallel, we now have one large one, with an input rate \( n\lambda \) and a service rate \( n\mu \). Let us call the mean response time of this combined system \( T_1 \). From the formula above we find

\[
T_1 = \frac{1}{n\mu - n\lambda} = T/n
\]

This surprising result says that by replacing \( n \) small resources by one big one that is \( n \) times more powerful, we can reduce the average response time \( n \)-fold. This result is extremely general and applies to a large variety of systems. It is one of the main reasons that airlines prefer to fly a 300-seat 747 once every 5 hours to a 10-seat business jet every 10 minutes. The effect arises because dividing the processing power into small servers (e.g., personal workstations), each with one user, is a poor match to a workload of randomly arriving requests. Much of the time, a few servers are busy, even overloaded, but most are idle. It is this wasted time that is eliminated in the processor pool model, and the reason why it gives better overall performance. The concept of using idle workstations is a weak attempt at recapturing the wasted cycles, but it is complicated and has many problems, as we have seen.

In fact, this queueing theory result is one of the main arguments against having distributed systems at all. Given a choice between one centralized 1000-MIPS CPU and 100 private, dedicated, 10-MIPS CPUs, the mean response time of the former will be 100 times better, because no cycles are ever wasted. The machine goes idle only when no user has any work to do. This fact argues in favor of concentrating the computing power as much as possible.

However, mean response time is not everything. There are also arguments in favor of distributed computing, such as cost. If a single 1000-MIPS CPU is much more expensive than 100 10-MIPS CPUs, the price/performance ratio of the latter may be much better. It may not even be possible to build such a large machine at any price. Reliability and fault tolerance are also factors.

Also, personal workstations have a uniform response, independent of what other people are doing (except when the network or file servers are jammed). For some users, a low variance in response time may be perceived as more important than the mean response time itself. Consider, for example, editing on a private workstation on which asking for the next page to be displayed always takes 500 msec. Now consider editing on a large, centralized, shared computer on which asking for the next page takes 5 msec 95 percent of the time and 5 sec one time in 20. Even though the mean here is twice as good as on the workstation, the users may consider the performance intolerable. On the other hand, to the user with a huge simulation to run, the big computer may win hands down.

So far we have tacitly assumed that a pool of \( n \) processors is effectively the same thing as a single processor that is \( n \) times as fast as a single processor. In reality, this assumption is justified only if all requests can be split up in such a way as to allow them to run on all the processors in parallel. If a job can be split into, say, only 5 parts, then the processor pool model has an effective service time only 5 times better than that of a single processor, not \( n \) times better.

Still, the processor pool model is a much cleaner way of getting extra computing power than looking around for idle workstations and sneaking over there while nobody is looking. By starting out with the assumption that no processor belongs to anyone, we get a design based on the concept of requesting machines from the pool, using them, and putting them back when done. There is also no need to forward anything back to a "home" machine because there are none.

There is also no danger of the owner coming back, because there are no owners. In the end, it all comes down to the nature of the workload. If all people are doing is simple editing and occasionally sending an electronic mail message or two, having a personal workstation is probably enough. If, on the other hand, the users are engaged in a large software development project, frequently running make on large directories, or are trying to invert massive sparse matrices, or do major simulations or run big artificial intelligence or VLSI routing programs, constantly hunting for
substantial numbers of idle workstations will be no fun at all. In all these situations, the processor pool idea is fundamentally much simpler and more attractive.

4.2.4. A Hybrid Model

A possible compromise is to provide each user with a personal workstation and to have a processor pool in addition. Although this solution is more expensive than either a pure workstation model or a pure processor pool model, it combines the advantages of both of the others.

Interactive work can be done on workstations, giving guaranteed response. Idle workstations, however, are not utilized, making for a simpler system design. They are just left unused. Instead, all noninteractive processes run on the processor pool, as does all heavy computing in general. This model provides fast interactive response, an efficient use of resources, and a simple design.

4.3. PROCESSOR ALLOCATION

By definition, a distributed system consists of multiple processors. These may be organized as a collection of personal workstations, a public processor pool, or some hybrid form. In all cases, some algorithm is needed for deciding which process should be run on which machine. For the workstation model, the question is when to run a process locally and when to look for an idle workstation. For the processor pool model, a decision must be made for every new process. In this section we will study the algorithms used to determine which process is assigned to which processor. We will follow tradition and refer to this subject as "processor allocation" rather than "process allocation," although a good case can be made for the latter.

4.3.1. Allocation Models

Before looking at specific algorithms, or even at design principles, it is worthwhile saying something about the underlying model, assumptions, and goals of the work on processor allocation. Nearly all work in this area assumes that all the machines are identical, or at least code-compatible, differing at most by speed. An occasional paper assumes that the system consists of several disjoint processor pools, each of which is homogeneous. These assumptions are usually valid, and make the problem much simpler, but leave unanswered for the time being such questions as whether a command to start up the text formatter should be started up on a 486, SPARC, or MIPS CPU, assuming that binaries for all of them are available.

Almost all published models assume that the system is fully interconnected, that is, every processor can communicate with every other processor. We will assume this as well. This assumption does not mean that every machine has a wire to every other machine, just that transport connections can be established between every pair. That messages may have to be routed hop by hop over a sequence of machines is of interest only to the lower layers. Some networks support broadcasting or multicasting, and some algorithms use these facilities.

New work is generated when a running process decides to fork or otherwise create a subprocess. In some cases the forking process is the command interpreter (shell) that is starting up a new job in response to a command from the user. In others, a user process itself creates one or more children, for example, in order to gain performance by having all the children run in parallel.

Processor allocation strategies can be divided into two broad classes. In the first, which we shall call nonmigratory, when a process is created, a decision is made about where to put it. once placed on a machine, the process stays there until it terminates. It may not move, no matter how badly overloaded its machine becomes and no matter how many other machines are idle. In contrast, with migratory allocation algorithms, a process can be moved even if it has already started execution. while migratory strategies allow better load balancing, they are more complex and have a major impact on system design.

Implicit in an algorithm that assigns processes to processors is that we are trying to optimize something. If this were not the case, we could just make the assignments at random or in numerical order. Precisely what it is that is being optimized, however, varies from one system to another. One possible goal is to maximize CPU utilization, that is, maximize the number of cpu
cycles actually executed on behalf of user jobs per hour of real time. Maximizing CPU utilization is another way of saying that CPU idle time is to be avoided at all costs. When in doubt, make sure that every CPU has something to do.

Another worthy objective is minimizing mean response time. Consider, for example, the two processors and two processes of Fig. 4-15. Processor 1 runs at 10 MIPS; processor 2 runs at 100 MIPS, but has a waiting list of backlogged processes that will take 5 sec to finish off. Process A has 100 million instructions and process B has 300 million. The response times for each process on each processor (including the wait time) are shown in the figure. If we assign A to processor 1 and B to processor 2, the mean response time will be \((10 + 8)/2\) or 9 sec. If we assign them the other way around, the mean response time will be \((30 + 6)/2\) or 18 sec. Clearly, the former is a better assignment in terms of minimizing mean response time.

![Fig. 4-15. Response times of two processes on two processors.](image)

A variation of minimizing the response time is minimizing the response ratio. The response ratio is defined as the amount of time it takes to run a process on some machine, divided by how long it would take on some unloaded benchmark processor. For many users, response ratio is a more useful metric than response time since it takes into account the fact that big jobs are supposed to take longer than small ones. To see this point, which is better, a 1-sec job that takes 5 sec or a 1-min job that takes 70 sec? Using response time, the former is better, but using response ratio, the latter is much better because \(5/1 \gg 70/60\).

### 4.3.2. Design Issues for Processor Allocation Algorithms

A large number of processor allocation algorithms have been proposed over the years. In this section we will look at some of the key choices involved in these algorithms and point out the various trade-offs. The major decisions the designers must make can be summed up in five issues:

1. Deterministic versus heuristic algorithms.
2. Centralized versus distributed algorithms.
3. Optimal versus suboptimal algorithms.
4. Local versus global algorithms.
5. Sender-initiated versus receiver-initiated algorithms.

Other decisions also come up, but these are the main ones that have been studied extensively in the literature. Let us look at each of these in turn.

Deterministic algorithms are appropriate when everything about process behavior is known in advance. Imagine that you have a complete list of all processes, their computing requirements, their file requirements, their communication requirements, and so on. Armed with this information, it is possible to make a perfect assignment. In theory, one could try all possible assignments and take the best one.
In few, if any, systems, is total knowledge available in advance, but sometimes a reasonable approximation is obtainable. For example, in banking, insurance, or airline reservations, today's work is just like yesterday's. The airlines have a pretty good idea of how many people want to fly from New York to Chicago on a Monday morning in early Spring, so the nature of the workload can be accurately characterized, at least statistically, making it possible to consider deterministic allocation algorithms.

At the other extreme are systems where the load is completely unpredictable. Requests for work depend on who's doing what, and can change dramatically from hour to hour, or even from minute to minute. Processor allocation in such systems cannot be done in a deterministic, mathematical way, but of necessity uses ad hoc techniques called **heuristics**.

The second design issue is centralized versus distributed. This theme has occurred repeatedly throughout the book. Collecting all the information in one place allows a better decision to be made, but is less robust and can put a heavy load on the central machine. Decentralized algorithms are usually preferable, but some centralized algorithms have been proposed for lack of suitable decentralized alternatives.

The third issue is related to the first two: Are we trying to find the best allocation, or merely an acceptable one? Optimal solutions can be obtained in both centralized and decentralized systems, but are invariably more expensive than suboptimal ones. They involve collecting more information and processing it more thoroughly. In practice, most actual distributed systems settle for heuristic, distributed, suboptimal solutions because it is hard to obtain optimal ones.

The fourth issue relates to what is often called **transfer policy**. When a process is about to be created, a decision has to be made whether or not it can be run on the machine where it is being generated. If that machine is too busy, the new process must be transferred somewhere else. The choice here is whether or not to base the transfer decision entirely on local information. One school of thought advocates a simple (local) algorithm: if the machine's load is below some threshold, keep the new process; otherwise, try to get rid of it. Another school says that this heuristic is too crude. Better to collect (global) information about the load elsewhere before deciding whether or not the local machine is too busy for another process. Each school has its points. Local algorithms are simple, but may be far from optimal, whereas global ones may only give a slightly better result at much higher cost.

The last issue in our list deals with **location policy**. Once the transfer policy has decided to get rid of a process, the location policy has to figure out where to send it. Clearly, the location policy cannot be local. It needs information about the load elsewhere to make an intelligent decision. This information can be disseminated in two ways, however. In one method, the senders start the information exchange. In another, it is the receivers that take the initiative.

As a simple example, look at Fig. 4-16(a). Here an overloaded machine sends out requests for help to other machines, in hopes of offloading its new process on some other machine. The sender takes the initiative in locating more CPU cycles in this example. In contrast, in Fig. 4-16(b), a machine that is idle or underloaded announces to other machines that it has little to do and is prepared to take on extra work. Its goal is to locate a machine that is willing to give it some work to do.
For both the sender-initiated and receiver-initiated cases, various algorithms have different strategies for whom to probe, how long to continue probing, and what to do with the results. Nevertheless, the difference between the two approaches should be clear by now.

**4.3.3. Implementation Issues for Processor Allocation Algorithms**

The points raised in the preceding section are all clear-cut theoretical issues about which one can have endless wonderful debates. In this section we will look at some other issues that are more related to the nitty-gritty details of implementing processor allocation algorithms than to the great principles behind them.

To start with, virtually all the algorithms assume that machines know their own load, so they can tell if they are underloaded or overloaded, and can tell other machines about their state. Measuring load is not as simple as it first appears. One approach is simply to count the number of processes on each machine and use that number as the load. However, as we have pointed out before, even on an idle system there may be many processes running, including mail and news daemons, window managers, and other processes. Thus the process count says almost nothing about the current load.

The next step is to count only processes that are running or ready. After all, every running or runnable process puts some load on the machine, even if it is a background process. However, many of these daemons wake up periodically, check to see if anything interesting has happened, and if not, go back to sleep. Most put only a small load on the system.

A more direct measurement, although it is more work to capture, is the fraction of time the CPU is busy. Clearly, a machine with a 20 percent CPU utilization is more heavily loaded than one with a 10 percent CPU utilization, whether it is running user or daemon programs. One way to measure the CPU utilization is to set up a timer and let it interrupt the machine periodically. At each interrupt, the state of the CPU is observed. In this way, the fraction of time spent in the idle loop can be observed.

A problem with timer interrupts is that when the kernel is executing critical code, it will often disable all interrupts, including the timer interrupt. Thus if the timer goes off while the kernel is active, the interrupt will be delayed until the kernel finishes. If the kernel was in the process of blocking the last active processes, the timer will not go off until the kernel has finished — and entered the idle loop. This effect will tend to underestimate the true CPU usage.

Another implementation issue is how overhead is dealt with. Many theoretical processor allocation algorithms ignore the overhead of collecting measurements and moving processes around. If an algorithm discovers that by moving a newly created process to a distant machine it
can improve system performance by 10 percent, it may be better to do nothing, since the cost of moving the process may eat up all the gain. A proper algorithm should take into account the CPU time, memory usage, and network bandwidth consumed by the processor allocation algorithm itself. Few do, mostly because it is not easy.

Our next implementation consideration is complexity. Virtually all researchers measure the quality of their algorithms by looking at analytical, simulation, or experimental measures of CPU utilization, network usage, and response time. Seldom is the complexity of the software considered, despite the obvious implications for system performance, correctness, and robustness. It rarely happens that someone publishes a new algorithm, demonstrates how good its performance is, and then concludes that the algorithm is not worth using because its performance is only slightly better than existing algorithms but is much more complicated to implement (or slower to run).

In this vein, a study by Eager et al. (1986) sheds light on the subject of pursuing complex, optimal algorithms. They studied three algorithms. In all cases, each machine measures its own load and decides for itself whether it is underloaded. Whenever a new process is created, the creating machine checks to see if it is overloaded. If so, it seeks out a remote machine on which to start the new process. The three algorithms differ in how the candidate machine is located.

Algorithm 1 picks a machine at random and just sends the new process there. If the receiving machine itself is overloaded, it, too, picks a random machine and sends the process off. This process is repeated until either somebody is willing to take it, or a hop counter is exceeded, in which case no more forwarding is permitted.

Algorithm 2 picks a machine at random and sends it a probe asking if it is underloaded or overloaded. If the machine admits to being underloaded, it gets the new process; otherwise, a new probe is tried. This loop is repeated until a suitable machine is found or the probe limit is exceeded, in which case it stays where it is created.

Algorithm 3 probes $k$ machines to determine their exact loads. The process is then sent to the machine with the smallest load.

Intuitively, if we ignore all the overhead of the probes and process transfers, one would expect algorithm 3 to have the best performance, and indeed it does. But the gain in performance of algorithm 3 over algorithm 2 is small, even though the complexity and amount of additional work required are larger. Eager et al. concluded that if using a simple algorithm gives you most of the gain of a much more expensive and complicated one, it is better to use the simple one.

Our final point here is that stability is also an issue that crops up. Different machines run their algorithms asynchronously from one another, so the system is rarely in equilibrium. It is possible to get into situations where neither $A$ nor $B$ has quite up-to-date information, and each thinks the other has a lighter load, resulting in some poor process being shuttled back and forth repeatedly. The problem is that most algorithms that exchange information can be shown to be correct after all the information has been exchanged and everything has settled down, but little can be said about their operation while tables are still being updated. It is in these nonequilibrium situations that problems often arise.

4.3.4. Example Processor Allocation Algorithms

To provide insight into how processor allocation can really be accomplished, in this section we will discuss several different algorithms. These have been selected to cover a broad range of possibilities, but there are others as well.

A Graph-Theoretic Deterministic Algorithm

A widely-studied class of algorithm is for systems consisting of processes with known CPU and memory requirements, and a known matrix giving the average amount of traffic between each pair
of processes. If the number of CPUs, \( k \), is smaller than the number of processes, several processes will have to be assigned to each CPU. The idea is to perform this assignment such as to minimize network traffic.

The system can be represented as a weighted graph, with each node being a process and each arc representing the flow of messages between two processes. Mathematically, the problem then reduces to finding a way to partition (i.e., cut) the graph into \( k \) disjoint subgraphs, subject to certain constraints (e.g., total CPU and memory requirements below some limits for each subgraph). For each solution that meets the constraints, arcs that are entirely within a single subgraph represent intramachine communication and can be ignored. Arcs that go from one subgraph to another represent network traffic. The goal is then to find the partitioning that minimizes the network traffic while meeting all the constraints. Figure 4-17 shows two ways of partitioning the same graph, yielding two different network loads.

![Figure 4-17](image)

Fig. 4-17. Two ways of allocating nine processes to three processors.

In Fig. 4-17(a), we have partitioned the graph with processes \( A, E, \) and \( G \) on one processor, processes \( B, F, \) and \( H \) on a second, and processes \( C, D, \) and \( I \) on the third. The total network traffic is the sum of the arcs intersected by the dotted cut lines, or 30 units. In Fig. 4-17(b) we have a different partitioning that has only 28 units of network traffic. Assuming that it meets all the memory and CPU constraints, this is a better choice because it requires less communication.

Intuitively, what we are doing is looking for clusters that are tightly coupled (high intracluster traffic flow) but which interact little with other clusters (low intercluster traffic flow). Some of the many papers discussing the problem are (Chow and Abraham, 1982; Stone and Bokhari, 1978; and Lo, 1984).

### A Centralized Algorithm

Graph-theoretic algorithms of the kind we have just discussed are of limited applicability since they require complete information in advance, so let us turn to a heuristic algorithm that does not require any advance information. This algorithm, called **up-down** (Mutka and Livny, 1987), is centralized in the sense that a coordinator maintains a **usage table** with one entry per personal workstation (i.e., per user), initially zero. When significant events happen, messages are sent to the coordinator to update the table. Allocation decisions are based on the table. These decisions are made when scheduling events happen: a processor is being requested, a processor has become free, or the clock has ticked.

The unusual thing about this algorithm, and the reason that it is centralized, is that instead of trying to maximize CPU utilization, it is concerned with giving each workstation owner a fair share of the computing power. Whereas other algorithms will happily let one user take over all the
machines if he promises to keep them all busy (i.e., achieve a high CPU utilization), this algorithm is designed to prevent precisely that.

When a process is to be created, and the machine it is created on decides that the process should be run elsewhere, it asks the usage table coordinator to allocate it a processor. If there is one available and no one else wants it, the request is granted. If no processors are free, the request is temporarily denied and a note is made of the request.

When a workstation owner is running processes on other people's machines, it accumulates penalty points, a fixed number per second, as shown in Fig. 4-18. These points are added to its usage table entry. When it has unsatisfied requests pending, penalty points are subtracted from its usage table entry. When no requests are pending and no processors are being used, the usage table entry is moved a certain number of points closer to zero, until it gets there. In this way, the score goes up and down, hence the name of the algorithm.

Usage table entries can be positive, zero, or negative. A positive score indicates that the workstation is a net user of system resources, whereas a negative score means that it needs resources. A zero score is neutral.

The heuristic used for processor allocation can now be given. When a processor becomes free, the pending request whose owner has the lowest score wins. As a consequence, a user who is occupying no processors and who has had a request pending for a long time will always beat someone who is using many processors. This property is the intention of the algorithm, to allocate capacity fairly.

In practice this means that if one user has a fairly continuous load on the system, and another user comes along and wants to start a process, the light user will be favored over the heavy one. Simulation studies (Mutka and Livny, 1987) show that the algorithm works as expected under a variety of load conditions.

![Fig. 4-18. Operation of the up-down algorithm.](image)

**A Hierarchical Algorithm**
Centralized algorithms, such as up-down, do not scale well to large systems. The central node soon becomes a bottleneck, not to mention a single point of failure. These problems can be attacked by using a hierarchical algorithm instead of a centralized one. Hierarchical algorithms retain much of the simplicity of centralized ones, but scale better.

One approach that has been proposed for keeping tabs on a collection of processors is to organize them in a logical hierarchy independent of the physical structure of the network, as in MICROS (Wittie and van Tilborg, 1980). This approach organizes the machines like people in corporate, military, academic, and other real-world hierarchies. Some of the machines are workers and others are managers.

For each group of \( k \) workers, one manager machine (the "department head") is assigned the task of keeping track of who is busy and who is idle. If the system is large, there will be an unwieldy number of department heads, so some machines will function as "deans," each riding herd on some number of department heads. If there are many deans, they too can be organized hierarchically, with a "big cheese" keeping tabs on a collection of deans. This hierarchy can be extended ad infinitum, with the number of levels needed growing logarithmically with the number of workers. Since each processor need only maintain communication with one superior and a few subordinates, the information stream is manageable.

An obvious question is: What happens when a department head, or worse yet, a big cheese, stops functioning (crashes)? One answer is to promote one of the direct subordinates of the faulty manager to fill in for the boss. The choice of which can be made by the subordinates themselves, by the deceased's peers, or in a more autocratic system, by the sick manager's boss.

To avoid having a single (vulnerable) manager at the top of the tree, one can truncate the tree at the top and have a committee as the ultimate authority, as shown in Fig. 4-19. When a member of the ruling committee malfunctions, the remaining members promote someone one level down as replacement.

![Fig. 4-19. A processor hierarchy can be modeled as an organizational hierarchy.](image)

While this scheme is not really distributed, it is feasible, and in practice works well. In particular, the system is self-repairing and can survive occasional crashes of both workers and managers without any long-term effects.

In MICROS, the processors are monoprogrammed, so if a job requiring \( S \) processes suddenly appears, the system must allocate 5 processors for it. Jobs can be created at any level of the hierarchy. The strategy used is for each manager to keep track of approximately how many workers below it are available (possibly several levels below it). If it thinks that a sufficient number are available, it reserves some number \( R \) of them, where \( R > S \), because the estimate of available workers may not be exact and some machines may be down.

If the manager receiving the request thinks that it has too few processors available, it passes the request upward in the tree to its boss. If the boss cannot handle it either, the request continues propagating upward until it reaches a level that has enough available workers at its disposal. At that point, the manager splits the request into parts and parcels them out among the managers below it, which then do the same thing until the wave of allocation requests hits bottom. At the
bottom level, the processors are marked as "busy" and the actual number of processors allocated is reported back up the tree.

To make this strategy work well, \( R \) must be large enough that the probability is high that enough workers will be found to handle the entire job. Otherwise the request will have to move up one level in the tree and start all over, wasting considerable time and computing power. On the other hand, if \( R \) is too large, too many processors will be allocated, wasting computing capacity until word gets back to the top and they can be released.

The whole situation is greatly complicated by the fact that requests for processors can be generated randomly anywhere in the system, so at any instant, multiple requests are likely to be in various stages of the allocation algorithm, potentially giving rise to out-of-date estimates of available workers, race conditions, deadlocks, and more. In Van Tilborg and Wittie (1981) a mathematical analysis of the problem is given and various other aspects not described here are covered in detail.

**A Sender-Initiated Distributed Heuristic Algorithm**

The algorithms described above are all centralized or semicentralized. Distributed algorithms also exist. Typical of these are the ones described by Eager et al. (1986). As mentioned above, in the most cost-effective algorithm they studied, when a process is created, the machine on which it originates sends probe messages to a randomly-chosen machine, asking if its load is below some threshold value. If so, the process is sent there. If not, another machine is chosen for probing. Probing does not go on forever. If no suitable host is found within \( N \) probes, the algorithm terminates and the process runs on the originating machine.

An analytical queueing model of this algorithm has been constructed and investigated. Using this model, it was established that the algorithm behaves well and is stable under a wide range of parameters, including different threshold values, transfer costs, and probe limits.

Nevertheless, it should be observed that under conditions of heavy load, all machines will constantly send probes to other machines in a futile attempt to find one that is willing to accept more work. Few processes will be off-loaded, but considerable overhead may be incurred in the attempt to do so.

**A Receiver-Initiated Distributed Heuristic Algorithm**

A complementary algorithm to the one given above, which is initiated by an overloaded sender, is one initiated by an underloaded receiver. With this algorithm, whenever a process finishes, the system checks to see if it has enough work. If not, it picks some machine at random and asks it for work. If that machine has nothing to offer, a second, and then a third machine is asked. If no work is found with \( N \) probes, the receiver temporarily stops asking, does any work it has queued up, and tries again when the next process finishes. If no work is available, the machine goes idle. After some fixed time interval, it begins probing again.

An advantage of this algorithm is that it does not put extra load on the system at critical times. The sender-initiated algorithm makes large numbers of probes precisely when the system can least tolerate it — when it is heavily loaded. With the receiver-initiated algorithm, when the system is heavily loaded, the chance of a machine having insufficient work is small, but when this does happen, it will be easy to find work to take over. Of course, when there is little work to do, the receiver-initiated algorithm, creates considerable probe traffic as all the unemployed machines
desperately hunt for work. However, it is far better to have the overhead go up when the system is underloaded than when it is overloaded.

It is also possible to combine both of these algorithms and have machines try to get rid of work when they have too much, and try to acquire work when they do not have enough. Furthermore, machines can perhaps improve on random polling by keeping a history of past probes to determine if any machines are chronically underloaded or overloaded. One of these can be tried first, depending on whether the initiator is trying to get rid of work or acquire it.

### A Bidding Algorithm

Another class of algorithms tries to turn the computer system into a miniature economy, with buyers and sellers of services and prices set by supply and demand (Ferguson et al., 1988). The key players in the economy are the processes, which must buy CPU time to get their work done, and processors, which auction their cycles off to the highest bidder.

Each processor advertises its approximate price by putting it in a publicly readable file. This price is not guaranteed, but gives an indication of what the service is worth (actually, it is the price that the last customer paid). Different processors may have different prices, depending on their speed, memory size, presence of floating-point hardware, and other features. An indication of the service provided, such as expected response time, can also be published.

When a process wants to start up a child process, it goes around and checks out who is currently offering the service that it needs. It then determines the set of processors whose services it can afford. From this set, it computes the best candidate, where "best" may mean cheapest, fastest, or best price/performance, depending on the application. It then generates a bid and sends the bid to its first choice. The bid may be higher or lower than the advertised price.

Processors collect all the bids sent to them, and make a choice, presumably by picking the highest one. The winners and losers are informed, and the winning process is executed. The published price of the server is then updated to reflect the new going rate.

Although Ferguson et al. do not go into the details, such an economic model raises all kinds of interesting questions, among them the following. Where do processes get money to bid? Do they get regular salaries? Does everyone get the same monthly salary, or do deans get more than professors, who in turn get more than students? If new users are introduced into the system without a corresponding increase in resources, do prices get bid up (inflation)? Can processors form cartels to gouge users? Are users' unions allowed? Is disk space also chargeable? How about laser printer output? The list goes on and on.

### 4.4. SCHEDULING IN DISTRIBUTED SYSTEMS

There is not really a lot to say about scheduling in a distributed system. Normally, each processor does its own local scheduling (assuming that it has multiple processes running on it), without regard to what the other processors are doing. Usually, this approach works fine. However, when a group of related, heavily interacting processes are all running on different processors, independent scheduling is not always the most efficient way.
The basic difficulty can be illustrated by an example in which processes $A$ and $B$ run on one processor and processes $C$ and $D$ run on another. Each processor is timeshared in, say, 100-msec time slices, with $A$ and $C$ running in the even slices and $B$ and $D$ running in the odd ones, as shown in Fig. 4-20(a). Suppose that $A$ sends many messages or makes many remote procedure calls to $D$. During time slice 0, $A$ starts up and immediately calls $D$, which unfortunate is not running because it is now $C$'s turn. After 100 msec, process switching takes place, and $D$ gets $A$'s message, carries out the work, and quickly replies. Because $B$ is now running, it will be another 100 msec before $A$ gets the reply and can proceed. The net result is one message exchange every 200 msec. What is needed is a way to ensure that processes that communicate frequently run simultaneously.

Although it is difficult to determine dynamically the interprocess communication patterns, in many cases, a group of related processes will be started off together. For example, it is usually a good bet that the filters in a UNIX pipeline will communicate with each other more than they will with other, previously started processes. Let us assume that processes are created in groups and that intragroup communication is much more prevalent than intergroup communication. Let us assume further that a sufficiently large number of processors is available to handle the largest group, and that each processor is multiprogrammed with $N$ process slots ($N$-way multiprogramming).

Ousterhout (1982) proposed several algorithms based on a concept he calls co-scheduling, which takes interprocess communication patterns into account while scheduling to ensure that all members of a group run at the same time. The first algorithm uses a conceptual matrix in which each column is the process table for one processor, as shown in Fig. 4-20(b). Thus, column 4 consists of all the processes that run on processor 4. Row 3 is the collection of all processes that are in slot 3 of some processor, starting with the process in slot 3 of processor 0, then the process in slot 3 of processor 1, and so on. The gist of his idea is to have each processor use a round-robin scheduling algorithm with all processors first running the process in slot 0 for a fixed period, then all processors running the process in slot 1 for a fixed period, and so on. A broadcast message could be used to tell each processor when to do process switching, to keep the time slices synchronized.

By putting all the members of a process group in the same slot number, but on different processors, one has the advantage of $N$–fold parallelism, with a guarantee that all the processes will be run at the same time, to maximize communication throughput. Thus in Fig. 4-20(b), four processes that must communicate should be put into slot 3, on processors 1, 2, 3, and 4 for optimum performance. This scheduling technique can be combined with the hierarchical model of process management used in MICROS by having each department head maintain the matrix for its workers, assigning processes to slots in the matrix and broadcasting time signals.
Ousterhout also described several variations to this basic method to improve performance. One of these breaks the matrix into rows and concatenates the rows to form one long row. With \( k \) processors, any \( k \) consecutive slots belong to different processors. To allocate a new process group to slots, one lays a window \( k \) slots wide over the long row such that the leftmost slot is empty but the slot just outside the left edge of the window is full. If sufficient empty slots are present in the window, the processes are assigned to the empty slots; otherwise, the window is slid to the right and the algorithm repeated. Scheduling is done by starting the window at the left edge and moving rightward by about one window's worth per time slice, taking care not to split groups over windows. Ousterhout's paper discusses these and other methods in more detail and gives some performance results.

4.5. FAULT TOLERANCE

A system is said to fail when it does not meet its specification. In some cases, such as a supermarket's distributed ordering system, a failure may result in some store running out of canned beans. In other cases, such in a distributed air traffic control system, a failure may be catastrophic. As computers and distributed systems become widely used in safety-critical missions, the need to prevent failures becomes correspondingly greater. In this section we will examine some issues concerning system failures and how they can be avoided. Additional introductory material can be found in (Cristian, 1991; and Nelson, 1990). Gantenbein (1992) has compiled a bibliography on the subject.

4.5.1. Component Faults

Computer systems can fail due to a fault in some component, such as a processor, memory, I/O device, cable, or software. A fault is a malfunction, possibly caused by a design error, a manufacturing error, a programming error, physical damage, deterioration in the course of time, harsh environmental conditions (it snowed on the computer), unexpected inputs, operator error, rodents eating part of it, and many other causes. Not all faults lead (immediately) to system failures, but some do.

Faults are generally classified as transient, intermittent, or permanent. Transient faults occur once and then disappear. If the operation is repeated, the fault goes away. A bird flying through the beam of a microwave transmitter may cause lost bits on some network (not to mention a roasted bird). If the transmission times out and is retried, it will probably work the second time.

An intermittent fault occurs, then vanishes of its own accord, then reappears, and so on. A loose contact on a connector will often cause an intermittent fault. Intermittent faults cause a great deal of aggravation because they are difficult to diagnose. Typically, whenever the fault doctor shows up, the system works perfectly.

A permanent fault is one that continues to exist until the faulty component is repaired. Burnt-out chips, software bugs, and disk head crashes often cause permanent faults.

The goal of designing and building fault-tolerant systems is to ensure that the system as a whole continues to function correctly, even in the presence of faults. This aim is quite different from simply engineering the individual components to be highly reliable, but allowing (even expecting) the system to fail if one of the components does so.

Faults and failures can occur at all levels: transistors, chips, boards, processors, operating systems, user programs, and so on. Traditional work in the area of fault tolerance has been concerned mostly with the statistical analysis of electronic component faults. Very briefly, if some component has a probability \( p \) of malfunctioning in a given second of time, the probability of it not failing for \( k \) consecutive seconds and then failing is \( p(1-p)^k \). The expected time to failure is then given by the formula

\[
\text{mean time to failure} = \sum_{i=1}^{\infty} kp(1-p)^{i-1}
\]
Using the well-known equation for an infinite sum starting at $k-1$: $\Sigma a^k = a/(1-a)$, substituting $a=1-p$, differentiating both sides of the resulting equation with respect to $p$, and multiplying through by $-p$ we see that

$$\text{mean time to failure} = 1/p$$

For example, if the probability of a crash is $10^{-6}$ per second, the mean time to failure is $10^6$ sec or about 11.6 days.

### 4.5.2. System Failures

In a critical distributed system, often we are interested in making the system be able to survive component (in particular, processor) faults, rather than just making these unlikely. System reliability is especially important in a distributed system due to the large number of components present, hence the greater chance of one of them being faulty.

For the rest of this section, we will discuss processor faults or crashes, but this should be understood to mean equally well process faults or crashes (e.g., due to software bugs). Two types of processor faults can be distinguished:

1. Fail-silent faults.
2. Byzantine faults.

With fail-silent faults, a faulty processor just stops and does not respond to subsequent input or produce further output, except perhaps to announce that it is no longer functioning. These are also called fail-stop faults. With Byzantine faults, a faulty processor continues to run, issuing wrong answers to questions, and possibly working together maliciously with other faulty processors to give the impression that they are all working correctly when they are not. Undetected software bugs often exhibit Byzantine faults. Clearly, dealing with Byzantine faults is going to be much more difficult than dealing with fail-silent ones.

The term "Byzantine" refers to the Byzantine Empire, a time (330-1453) and place (the Balkans and modern Turkey) in which endless conspiracies, intrigue, and untruthfulness were alleged to be common in ruling circles. Byzantine faults were first analyzed by Pease et al. (1980) and Lamport et al. (1982). Some researchers also consider combinations of these faults with communication line faults, but since standard protocols can recover from line errors in predictable ways, we will examine only processor faults.

### 4.5.3. Synchronous versus Asynchronous Systems

As we have just seen, component faults can be transient, intermittent, or permanent, and system failures can be fail-silent or Byzantine. A third orthogonal axis deals with performance in a certain abstract sense. Suppose that we have a system in which if one processor sends a message to another, it is guaranteed to get a reply within a time $T$ known in advance. Failure to get a reply means that the receiving system has crashed. The time $T$ includes sufficient time to deal with lost messages (by sending them up to $n$ times).

In the context of research on fault tolerance, a system that has the property of always responding to a message within a known finite bound if it is working is said to be synchronous. A system not having this property is said to be asynchronous. While this terminology is unfortunately, since it conflicts with more traditional uses of the terms, it is widely used among workers in fault tolerance.

It should be intuitively clear that asynchronous systems are going to be harder to deal with than synchronous ones. If a processor can send a message and know that the absence of a reply within $T$ sec means that the intended recipient has failed, it can take corrective action. If there is no upper limit to how long the response might take, even determining whether there has been a failure is going to be a problem.

### 4.5.4. Use of Redundancy
The general approach to fault tolerance is to use redundancy. Three kinds are possible: information redundancy, time redundancy, and physical redundancy. With information redundancy, extra bits are added to allow recovery from garbled bits. For example, a Hamming code can be added to transmitted data to recover from noise on the transmission line.

With time redundancy, an action is performed, and then, if need be, it is performed again. Using the atomic transactions described in Chap. 3 is an example of this approach. If a transaction aborts, it can be redone with no harm. Time redundancy is especially helpful when the faults are transient or intermittent.

With physical redundancy, extra equipment is added to make it possible for the system as a whole to tolerate the loss or malfunctioning of some components. For example, extra processors can be added to the system so that if a few of them crash, the system can still function correctly.

There are two ways to organize these extra processors: active replication and primary backup. Consider the case of a server. When active replication is used, all the processors are used all the time as servers (in parallel) in order to hide faults completely. In contrast, the primary backup scheme just uses one processor as a server, replacing it with a backup if it fails.

We will discuss these two strategies below. For both of them, the issues are:
1. The degree of replication required.
2. The average and worst-case performance in the absence of faults.
3. The average and worst-case performance when a fault occurs.

Theoretical analyses of many fault-tolerant systems can be done in these terms. For more information, see (Schneider, 1990; and Budhiraja et al., 1993).

4.5.5. Fault Tolerance Using Active Replication

Active replication is a well-known technique for providing fault tolerance using physical redundancy. It is used in biology (mammals have two eyes, two ears, two lungs, etc.), aircraft (747s have four engines but can fly on three), and sports (multiple referees in case one misses an event). Some authors refer to active replication as the state machine approach.

It has also been used for fault tolerance in electronic circuits for years. Consider, for example, the circuit of Fig. 4-21(a). Here signals pass through devices A, B, and C, in sequence. If one of them is faulty, the final result will probably be wrong.

In Fig. 4-21(b), each device is replicated three times. Following each stage in the circuit is a triplicated voter. Each voter is a circuit that has three inputs and one output. If two or three of the inputs are the same, the output is equal to that input. If all three inputs are different, the output is undefined. This kind of design is known as TMR (Triple Modular Redundancy).

![Fig. 4-21. Triple modular redundancy.](image)
Suppose element $A_2$ fails. Each of the voters, $V_1$, $V_2$, and $V_3$ gets two good (identical) inputs and one rogue input, and each of them outputs the correct value to the second stage. In essence, the effect of $A_2$ failing is completed masked, so that the inputs to $B_1$, $B_2$, and $B_3$ are exactly the same as they would have been had no fault occurred.

Now consider what happens if $B_3$ and $C_1$ are also faulty, in addition to $A_2$. These effects are also masked, so the three final outputs are still correct.

At first it may not be obvious why three voters are needed at each stage. After all, one voter could also detect and pass through the majority view. However, a voter is also a component and can also be faulty. Suppose, for example, that $V_1$ malfunctions. The input to $B_1$ will then be wrong, but as long as everything else works, $B_2$ and $B_3$ will produce the same output and $V_4$, $V_5$, and $V_6$ will all produce the correct result into stage three. A fault in $V_1$ is effectively no different than a fault in $B_1$. In both cases $B_1$ produces incorrect output, but in both cases it is voted down later.

Although not all fault-tolerant distributed operating systems use TMR, the technique is very general, and should give a clear feeling for what a fault-tolerant system is, as opposed to a system whose individual components are highly reliable but whose organization is not fault tolerant. Of course, TMR can be applied recursively, for example, to make a chip highly reliable by using TMR inside it, unknown to the designers who use the chip.

Getting back to fault tolerance in general and active replication in particular, in many systems, servers act like big finite-state machines: they accept requests and produce replies. Read requests do not change the state of the server, but write requests do. If each client request is sent to each server, and they all are received and processed in the same order, then after processing each one, all nonfaulty servers will be in exactly the same state and will give the same replies. The client or voter can combine all the results to mask faults.

An important issue is how much replication is needed. The answer depends on the amount of fault tolerance desired. A system is said to be $k$ fault tolerant if it can survive faults in $k$ components and still meet its specifications. If the components, say processors, fail silently, then having $k+1$ of them is enough to provide $k$ fault tolerance. If $k$ of them simply stop, then the answer from the other one can be used.

On the other hand, if the processors exhibit Byzantine failures, continuing to run when sick and sending out erroneous or random replies, a minimum of $2k+1$ processors are needed to achieve $k$ fault tolerance. In the worst case, the $k$ failing processors could accidentally (or even intentionally) generate the same reply. However, the remaining $k+1$ will also produce the same answer, so the client or voter can just believe the majority.

Of course, in theory it is fine to say that a system is $k$ fault tolerant and just let the $k+1$ identical replies outvote the $k$ identical replies, but in practice it is hard to imagine circumstances in which one can say with certainty that $k$ processors can fail but $k+1$ processors cannot fail. Thus even in a fault-tolerant system some kind of statistical analysis may be needed.

An implicit precondition for this finite state machine model to be relevant is that all requests arrive at all servers in the same order, sometimes called the atomic broadcast problem. Actually, this condition can be relaxed slightly, since reads do not matter and some writes may commute, but the general problem remains. One way to make sure that all requests are processed in the same order at all servers is to number them globally. Various protocols have been devised to accomplish this goal. For example, all requests could first be sent to a global number server to get a serial number, but then provision would have to be made for the failure of this server (e.g., by making it internally fault tolerant).

Another possibility is to use Lamport’s logical clocks, as described in Chap. 3. If each message sent to a server is tagged with a timestamp, and servers process all requests in timestamp order, all requests will be processed in the same order at all servers. The trouble with this method is that when a server receives a request, it does not know whether any earlier requests are currently
under way. In fact, most timestamp solutions suffer from this problem. In short, active replication is not a trivial matter. Schneider (1990) discusses the problems and solutions in some detail.

### 4.5.6. Fault Tolerance Using Primary Backup

The essential idea of the primary-backup method is that at any one instant, one server is the primary and does all the work. If the primary fails, the backup takes over. Ideally, the cutover should take place in a clean way and be noticed only by the client operating system, not by the application programs. Like active replication, this scheme is widely used in the world. Some examples are government (the Vice President), aviation (co-pilots), automobiles (spare tires), and diesel-powered electrical generators in hospital operating rooms.

Primary-backup fault tolerance has two major advantages over active replication. First, it is simpler during normal operation since messages go to just one server (the primary) and not to a whole group. The problems associated with ordering these messages also disappear. Second, in practice it requires fewer machines, because at any instant one primary and one backup is needed (although when a backup is put into service as a primary, a new backup is needed instantly). On the downside, it works poorly in the presence of Byzantine failures in which the primary erroneously claims to be working perfectly. Also, recovery from a primary failure can be complex and time consuming.

As an example of the primary-backup solution, consider the simple protocol of Fig. 4-22 in which a write operation is depicted. The client sends a message to the primary, which does the work and then sends an update message to the backup. When the backup gets the message, it does the work and then sends an acknowledgement back to the primary. When the acknowledgement arrives, the primary sends the reply to the client.

![Fig. 4-22. A simple primary-backup protocol on a write operation.](image)

Now let us consider the effect of a primary crash at various moments during an RPC. If the primary crashes before doing the work (step 2), no harm is done. The client will time out and retry. If it tries often enough, it will eventually get the backup and the work will be done exactly once. If the primary crashes after doing the work but before sending the update, when the backup takes over and the request comes in again, the work will be done a second time. If the work has side effects, this could be a problem. If the primary crashes after step 4 but before step 6, the work may end up being done three times, once by the primary, once by the backup as a result of step 3, and once after the backup becomes the primary. If requests carry identifiers, it may be possible to ensure that the work is done only twice, but getting it done exactly once is difficult to impossible.

One theoretical and practical problem with the primary-backup approach is when to cut over from the primary to the backup. In the protocol above, the backup could send: "Are you alive?" messages periodically to the primary. If the primary fails to respond within a certain time, the backup would take over.

However, what happens if the primary has not crashed, but is merely slow (i.e., we have an asynchronous system)? There is no way to distinguish between a slow primary and one that has gone down. Yet there is a need to make sure that when the backup takes over, the primary really stops trying to act like the primary. Ideally the backup and primary should have a protocol to discuss this, but it is hard to negotiate with the dead. The best solution is a hardware mechanism...
in which the backup can forcibly stop or reboot the primary. Note that all primary-backup schemes require agreement, which is tricky to achieve, whereas active replication does not always require an agreement protocol (e.g., TMR).

A variant of the approach of Fig. 4-22 uses a dual-ported disk shared between the primary and secondary. In this configuration, when the primary gets a request, it writes the request to disk before doing any work and also writes the results to disk. No messages to or from the backup are needed. If the primary crashes, the backup can see the state of the world by reading the disk. The disadvantage of this scheme is that there is only one disk, so if that fails, everything is lost. Of course, at the cost of extra equipment and performance, the disk could also be replicated and all writes could be done to both disks.

### 4.5.7. Agreement in Faulty Systems

In many distributed systems there is a need to have processes agree on something. Examples are electing a coordinator, deciding whether to commit a transaction or not, dividing up tasks among workers, synchronization, and so on. When the communication and processors are all perfect, reaching such agreement is often straightforward, but when they are not, problems arise. In this section we will look at some of the problems and their solutions (or lack thereof).

The general goal of distributed agreement algorithms is to have all the non-faulty processors reach consensus on some issue, and do that within a finite number of steps. Different cases are possible depending on system parameters, including:

1. Are messages delivered reliably all the time?
2. Can processes crash, and if so, fail-silent or Byzantine?
3. Is the system synchronous or asynchronous?

Before considering the case of faulty processors, let us look at the "easy" case of perfect processors but communication lines that can lose messages. There is a famous problem, known as the **two-army problem**, which illustrates the difficulty of getting even two perfect processors to reach agreement about 1 bit of information. The red army, with 5000 troops, is encamped in a valley. Two blue armies, each 3000 strong, are encamped on the surrounding hillsides overlooking the valley. If the two blue armies can coordinate their attacks on the red army, they will be victorious. However, if either one attacks by itself it will be slaughtered. The goal of the blue armies is to reach agreement about attacking. The catch is that they can only communicate using an unreliable channel: sending a messenger who is subject to capture by the red army.

Suppose that the commander of blue army 1, General Alexander, sends a message to the commander of blue army 2, General Bonaparte, reading: "I have a plan — let's attack at dawn tomorrow." The messenger gets through and Bonaparte sends him back with a note saying: "Splendid idea, Alex. See you at dawn tomorrow." The messenger gets back to his base safely, delivers his messages, and Alexander tells his troops to prepare for battle at dawn.

However, later that day, Alexander realizes that Bonaparte does not know if the messenger got back safely and not knowing this, may not dare to attack. Consequently, Alexander tells the messenger to go tell Bonaparte that his (Bonaparte's) message arrived and that the battle is set. Once again the messenger gets through and delivers the acknowledgement. But now Bonaparte worries that Alexander does not know if the acknowledgement got through. He reasons that if Bonaparte thinks that the messenger was captured, he will not be sure about his (Alexander's) plans, and may not risk the attack, so he sends the messenger back again.

Even if the messenger makes it through every time, it is easy to show that Alexander and Bonaparte will never reach agreement, no matter how many acknowledgements they send. Assume that there is some protocol that terminates in a finite number of steps. Remove any extra steps at the end to get the minimum protocol that works. Some message is now the last one and it is
essential to the agreement (because this is the minimum protocol). If this message fails to arrive, the war is off.

However, the sender of the last message does not know if the last message arrived. If it did not, the protocol did not complete and the other general will not attack. Thus the sender of the last message cannot know if the war is scheduled or not, and hence cannot safely commit his troops. Since the receiver of the last message knows the sender cannot be sure, he will not risk certain death either, and there is no agreement. Even with nonfaulty processors (generals), agreement between even two processes is not possible in the face of unreliable communication.

Now let us assume that the communication is perfect but the processors are not. The classical problem here also occurs in a military setting and is called the Byzantine generals problem. In this problem the red army is still encamped in the valley, but n blue generals all head armies on the nearby hills. Communication is done pairwise by telephone and is perfect, but m of the generals are traitors (faulty) and are actively trying to prevent the loyal generals from reaching agreement by feeding them incorrect and contradictory information (to model malfunctioning processors). The question is now whether the loyal generals can still reach agreement.

For the sake of generality, we will define agreement in a slightly different way here. Each general is assumed to know how many troops he has. The goal of the problem is for the generals to exchange troop strengths, so that at the end of the algorithm, each general has a vector of length n corresponding to all the armies. If general i is loyal, then element i is his troop strength; otherwise, it is undefined.

A recursive algorithm was devised by Lamport et al. (1982) that solves this problem under certain conditions. In Fig. 4-23 we illustrate the working of the algorithm for the case of n=4 and m=1. For these parameters, the algorithm operates in four steps. In step one, every general sends a (reliable) message to every other general announcing his truth strength. Loyal generals tell the truth; traitors may tell every other general a different lie. In Fig. 4-23(a) we see that general 1 reports 1K troops, general 2 reports 2K troops, general 3 lies to everyone, giving x, y, and z, respectively, and general 4 reports 4K troops. In step 2, the results of the announcements of step 1 are collected together in the form of the vectors of Fig. 4-23(b).

Step 3 consists of every general passing his vector from Fig. 4-23(b) to every other general. Here, too, general 3 lies through his teeth, inventing 12 new values, a through l. The results of step 3 are shown in Fig. 4-23(c). Finally, in step 4, each general examines the i\(^{th}\) element of each of the newly received vectors. If any value has a majority, that value is put into the result vector. If
no value has a majority, the corresponding element of the result vector is marked UNKNOWN. From Fig. 4-23(c) we see that generals 1, 2, and 4 all come to agreement on (1, 2, UNKNOWN, 4) which is the correct result. The traitor was not able to gum up the works.

Now let us revisit this problem for \( m=3 \) and \( n=1 \), that is, only two loyal generals and one traitor, as illustrated in Fig. 4-24. Here we see that in Fig. 4-24(c) neither of the loyal generals sees a majority for element 1, element 2, or element 3, so all of them are marked UNKNOWN. The algorithm has failed to produce agreement.

![Fig. 4-24. The same as Fig. 4-23, except now with 2 loyal generals and one traitor.](image)

In their paper, Lamport et al. (1982) proved that in a system with \( m \) faulty processors, agreement can be achieved only if \( 2m+1 \) correctly functioning processors are present, for a total of \( 3m+1 \). Put in slightly different terms, agreement is possible only if more than two-thirds of the processors are working properly.

Worse yet, Fischer et al. (1985) proved that in a distributed system with asynchronous processors and unbounded transmission delays, no agreement is possible if even one processor is faulty (even if that one processor fails silently). The problem with asynchronous systems is that arbitrarily slow processors are indistinguishable from dead ones. Many other theoretical results are known about when agreement is possible and when it is not. Surveys of these results are given by Barborak et al. (1993) and Turek and Shasha (1992).

### 4.6. REAL-TIME DISTRIBUTED SYSTEMS

Fault-tolerant systems are not the only kind of specialized distributed systems. The real-time systems form another category. Sometimes these two are combined to give fault-tolerant real-time systems. In this section we will examine various aspects of real-time distributed systems. For additional material, see for example, (Burns and Wellings, 1990; Klein et al., 1994; and Shin, 1991).

#### 4.6.1. What Is a Real-Time System?

For most programs, correctness depends only on the logical sequence of instructions executed, not when they are executed. If a C program correctly computes the double-precision floating-point square root function on a 200-MHz engineering workstation, it will also compute the function correctly on a 4.77-MHz 8088-based personal computer, only slower.

In contrast, real-time programs (and systems) interact with the external world in a way that involves time. When a stimulus appears, the system must respond to it in a certain way and before a certain deadline. If it delivers the correct answer, but after the deadline, the system is regarded as having failed. When the answer is produced is as important as which answer is produced.
Consider a simple example. An audio compact disk player consists of a CPU that takes the bits arriving from the disk and processes them to generate music. Suppose that the CPU is just barely fast enough to do the job. Now imagine that a competitor decides to build a cheaper player using a CPU running at one-third the speed. If it buffers all the incoming bits and plays them back at one-third the expected speed, people will wince at the sound, and if it only plays every third note, the audience will not be wildly ecstatic either. Unlike the earlier square root example, time is inherently part of the specification of correctness here.

Many other applications involving the external world are also inherently real time. Examples include computers embedded in television sets and video recorders, computers controlling aircraft ailerons and other parts (so called fly-by-wire), automobile subsystems controlled by computers (drive-by-wire?), military computers controlling guided antitank missiles (shoot-by-wire?), computerized air traffic control systems, scientific experiments ranging from particle accelerators to psychology lab mice with electrodes in their brains, automated factories, telephone switches, robots, medical intensive care units, CAT scanners, automatic stock trading systems, and numerous others.

Many real-time applications and systems are highly structured, much more so than general-purpose distributed systems. Typically, an external device (possibly a clock) generates a stimulus for the computer, which must then perform certain actions before a deadline. When the required work has been completed, the system becomes idle until the next stimulus arrives.

Frequently, the stimuli are periodic, with a stimulus occurring regularly every AT seconds, such as a computer in a TV set or VCR getting a new frame every 1/60 of a second. Sometimes stimuli are aperiodic, meaning that they are recurrent, but not regular, as in the arrival of an aircraft in an air traffic controller’s air space. Finally, some stimuli are sporadic (unexpected), such as a device overheating.

Even in a largely periodic system, a complication is that there may be many types of events, such as video input, audio input, and motor drive management, each with its own period and required actions. Figure 4-25 depicts a situation with three periodic event streams, A, B, and C, plus one sporadic event, X.

**Fig. 4-25.** Superposition of three event streams plus one sporadic event.

Despite the fact that the CPU may have to deal with multiple event streams, it is not acceptable for it to say: It is true that I missed event B, but it is not my fault — I was still working on A when B happened. While it is not hard to manage two or three input streams with priority interrupts, as applications get larger and more complex (e.g., automated factory assembly lines with thousands of robots), it will become more and more difficult for one machine to meet all the deadlines and other real-time constraints.

Consequently, some designers are experimenting with the idea of putting a dedicated microprocessor in front of each real-time device to accept output from it whenever it has something to say, and give it input at whatever speed it requires. Of course, this does not make the real-time
character go away, but instead gives rise to a distributed real-time system, with its own unique characteristics and challenges (e.g., real-time communication).

Distributed real-time systems can often be structured as illustrated in Fig. 4-26. Here we see a collection of computers connected by a network. Some of these are connected to external devices that produce or accept data or expect to be controlled in real time. The computers may be tiny microcontrollers built into the devices, or stand-alone machines. In both cases they usually have sensors for receiving signals from the devices and/or actuators for sending signals to them. The sensors and actuators may be digital or analog.

![Fig. 4-26. A distributed real-time computer system.](image)

Real-time systems are generally split into two types depending on how serious their deadlines are and the consequences of missing one. These are:

1. **Soft real-time systems.**
2. **Hard real-time systems.**

**Soft real-time** means that missing an occasional deadline is all right. For example, a telephone switch might be permitted to lose or misroute one call in $10^5$ under overload conditions and still be within specification. In contrast, even a single missed deadline in a **hard real-time** system is unacceptable, as this might lead to loss of life or an environmental catastrophe. In practice, there are also intermediate systems where missing a deadline means you have to kill off the current activity, but the consequence is not fatal. For example, if a soda bottle on a conveyor belt has passed by the nozzle, there is no point in continuing to squirt soda at it, but the results are not fatal. Also, in some real-time systems, some subsystems are hard real time whereas others are soft real time.

Real-time systems have been around for decades, so there is a considerable amount of folk wisdom accumulated about them, most of it wrong. Stankovic (1988) has pointed out some of these myths, the worst of which are summarized here.

**Myth 1: Real-time systems are about writing device drivers in assembly code.**

This was perhaps true in the 1970s for real-time systems consisting of a few instruments attached to a minicomputer, but current real-time systems are too complicated to trust to assembly language and writing the device drivers is the least of a real-time system designer’s worries.

**Myth 2: Real-time computing is fast computing.**

Not necessarily. A computer-controlled telescope may have to track stars or galaxies in real time, but the apparent rotation of the heavens is only 15 degrees of arc per hour of time, not especially fast. Here accuracy is what counts.

**Myth 3: Fast computers will make real-time system obsolete.**

No. They just encourage people to build real-time systems that were previously beyond the state-of-the-art. Cardiologists would love to have an MRI scanner that shows a beating heart inside an exercising patient in real time. When they get that, they will ask for it in three dimensions, in full
color, and with the possibility of zooming in and out. Furthermore, making systems faster by using multiple processors introduces new communication, synchronization, and scheduling problems that have to be solved.

4.6.2. Design Issues

Real-time distributed systems have some unique design issues. In this section we will examine some of the most important ones.

Clock Synchronization

The first issue is the maintenance of time itself. With multiple computers, each having its own local clock, keeping the clocks in synchrony is a key issue. We examined this point in Chap. 3, so we will not repeat that discussion here.

Event-Triggered versus Time-Triggered Systems

In an event-triggered real-time system, when a significant event in the outside world happens, it is detected by some sensor, which then causes the attached CPU to get an interrupt. Event-triggered systems are thus interrupt driven. Most real-time systems work this way. For soft real-time systems with lots of computing power to spare, this approach is simple, works well, and is still widely used. Even for more complex systems, it works well if the compiler can analyze the program and know all there is to know about the system behavior once an event happens, even if it cannot tell when the event will happen.

The main problem with event-triggered systems is that they can fail under conditions of heavy load, that is, when many events are happening at once. Consider, for example, what happens when a pipe ruptures in a computer-controlled nuclear reactor. Temperature alarms, pressure alarms, radioactivity alarms, and other alarms will all go off at once, causing massive interrupts. This event shower may overwhelm the computing system and bring it down, potentially causing problems far more serious than the rupture of a single pipe.

An alternative design that does not suffer from this problem is the time-triggered real-time system. In this kind of system, a clock interrupt occurs every AT milliseconds. At each clock tick (selected) sensors are sampled and (certain) actuators are driven. No interrupts occur other than clock ticks.

In the ruptured pipe example given above, the system would become aware of the problem at the first clock tick after the event, but the interrupt load would not change on account of the problem, so the system would not become overloaded. Being able to operate normally in times of crisis increases the chances of dealing successfully with the crisis.

It goes without saying that AT must be chosen with extreme care. If it is too small, the system will get many clock interrupts and waste too much time fielding them. If it is too large, serious events may not be noticed until it is too late. Also, the decision about which sensors to check on every clock tick, and which to check on every other clock tick, and so on, is critical. Finally, some events may be shorter than a clock tick, so they must be saved to avoid losing them. They can be preserved electrically by latch circuits or by microprocessors embedded in the external devices.

As an example of the difference between these two approaches, consider the design of an elevator controller in a 100-story building. Suppose that the elevator is sitting peacefully on the 60th floor waiting for customers. Then someone pushes the call button on the first floor. Just 100 msec later, someone else pushes the call button on the 100th floor. In an event-triggered system, the first call generates an interrupt, which causes the elevator to take off downward. The second
call comes in after the decision to go down has already been made, so it is noted for future reference, but the elevator continues on down.

Now consider a time-triggered elevator controller that samples every 500 msec. If both calls fall within one sampling period, the controller will have to make a decision, for example, using the nearest-customer-first rule, in which case it will go up.

In summary, event-triggered designs give faster response at low load but more overhead and chance of failure at high load. Time-trigger designs have the opposite properties and are furthermore only suitable in a relatively static environment in which a great deal is known about system behavior in advance. Which one is better depends on the application. In any event, we note that there is much lively controversy over this subject in real-time circles.

Predictability

One of the most important properties of any real-time system is that its behavior be predictable. Ideally, it should be clear at design time that the system can meet all of its deadlines, even at peak load. Statistical analyses of behavior assuming independent events are often misleading because there may be unsuspected correlations between events, as between the temperature, pressure, and radioactivity alarms in the ruptured pipe example above.

Most distributed system designers are used to thinking in terms of independent users accessing shared files at random or numerous travel agents accessing a shared airline data base at unpredictable times. Fortunately, this kind of chance behavior rarely holds in a real-time system. More often, it is known that when event $E$ is detected, process $X$ should be run, followed by processes $Y$ and $Z$, in either order or in parallel. Furthermore, it is often known (or should be known) what the worst-case behavior of these processes is. For example, if it is known that $X$ needs 50 msec, $Y$ and $Z$ need 60 msec each, and process startup takes 5 msec, then it can be guaranteed in advance that the system can flawlessly handle five periodic type $E$ events per second in the absence of any other work. This kind of reasoning and modeling leads to a deterministic rather than a stochastic system.

Fault Tolerance

Many real-time systems control safety-critical devices in vehicles, hospitals, and power plants, so fault tolerance is frequently an issue. Active replication is sometimes used, but only if it can be done without extensive (and thus time-consuming) protocols to get everyone to agree on everything all the time. Primary-backup schemes are less popular because deadlines may be missed during cutover after the primary fails. A hybrid approach is to follow the leader, in which one machine makes all the decisions, but the others just do what it says to do without discussion, ready to take over at a moment's notice.

In a safety-critical system, it is especially important that the system be able to handle the worst-case scenario. It is not enough to say that the probability of three components failing at once is so low that it can be ignored. Failures are not always independent. For example, during a sudden electric power failure, everyone grabs the telephone, possibly causing the phone system to overload, even though it has its own independent power generation system. Furthermore, the peak load on the system often occurs precisely at the moment when the maximum number of components have failed because much of the traffic is related to reporting the failures. Consequently, fault-tolerant real-time systems must be able to cope with the maximum number of faults and the maximum load at the same time.
Some real-time systems have the property that they can be stopped cold when a serious failure occurs. For instance, when a railroad signaling system unexpectedly blacks out, it may be possible for the control system to tell every train to stop immediately. If the system design always spaces trains far enough apart and all trains start braking more-or-less simultaneously, it will be possible to avert disaster and the system can recover gradually when the power comes back on. A system that can halt operation like this without danger is said to be **fail-safe**.

**Language Support**

While many real-time systems and applications are programmed in general-purpose languages such as C, specialized real-time languages can potentially be of great assistance. For example, in such a language, it should be easy to express the work as a collection of short tasks (e.g., lightweight processes or threads) that can be scheduled independently, subject to user-defined precedence and mutual exclusion constraints.

The language should be designed so that the maximum execution time of every task can be computed at compile time. This requirement means that the language cannot support general **while** loops. Iteration must be done using **for** loops with constant parameters. Recursion cannot be tolerated either (it is beginning to look like FORTRAN has a use after all). Even these restrictions may not be enough to make it possible to calculate the execution time of each task in advance since cache misses, page faults, and cycle stealing by DMA channels all affect performance, but they are a start.

Real-time languages need a way to deal with time itself. To start with, a special variable, *clock*, should be available, containing the current time in ticks. However, one has to be careful about the unit that time is expressed in. The finer the resolution, the faster *clock* will overflow. If it is a 32-bit integer, for example, the range for various resolutions is shown in Fig. 4-27. Ideally, the clock should be 64 bits wide and have a 1 nsec resolution.

<table>
<thead>
<tr>
<th>Clock resolution</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 nsec</td>
<td>4 seconds</td>
</tr>
<tr>
<td>1 µsec</td>
<td>72 minutes</td>
</tr>
<tr>
<td>1 msec</td>
<td>50 days</td>
</tr>
<tr>
<td>1 sec</td>
<td>136 years</td>
</tr>
</tbody>
</table>

Fig. 4-27. Range of a 32-bit clock before overflowing for various resolutions.

The language should have a way to express minimum and maximum delays. In Ada®, for example, there is a delay statement that specifies a minimum value that a process must be suspended. However, the actual delay may be more by an unbounded amount. There is no way to give an upper bound or a time interval in which the delay is required to fall.

There should also be a way to express what to do if an expected event does not occur within a certain interval. For example, if a process blocks on a semaphore for more than a certain time, it should be possible to time out and be released. Similarly, if a message is sent, but no reply is forthcoming fast enough, the sender should be able to specify that it is to be deblocked after *k* msec.

Finally, since periodic events play such a big role in real-time systems, it would be useful to have a statement of the form
every (25 msec) { ... }

that causes the statements within the curly brackets to be executed every 25 msec. Better yet, if a task contains several such statements, the compiler should be able to compute what percentage of the CPU time is required by each one, and from these data compute the minimum number of machines needed to run the entire program and how to assign processes to machines.

4.6.3. Real-Time Communication

Communication in real-time distributed systems is different from communication in other distributed systems. While high performance is always welcome, predictability and determinism are the real keys to success. In this section we will look at some real-time communication issues, for both LANs and WANs. Finally, we will examine one example system in some detail to show how it differs from conventional (i.e., non-real-time) distributed systems. Alternative approaches are described in (Malcolm and Zhao, 1994; and Ramanathan and Shin, 1992)

Achieving predictability in a distributed system means that communication between processors must also be predictable. LAN protocols that are inherently stochastic, such as Ethernet, are unacceptable because they do not provide a known upper bound on transmission time. A machine wanting to send a packet on an Ethernet may collide with one or more other machines. All machines then wait a random time and then try again, but these transmissions may also collide, and so on. Consequently, it is not possible to give a worst-case bound on packet transmission in advance.

As a contrast to Ethernet, consider a token ring LAN. Whenever a processor has a packet to send, it waits for the circulating token to pass by, then it captures the token, sends its packet, and puts the token back on the ring so that the next machine downstream gets the opportunity to seize it. Assuming that each of the \( k \) machines on the ring is allowed to send at most one \( n \)-byte packet per token capture, it can be guaranteed that an urgent packet arriving anywhere in the system can always be transmitted within \( kn \) byte times. This is the kind of upper bound that a real-time distributed system needs.

Token rings can also handle traffic consisting of multiple priority classes. The goal here is to ensure that if a high-priority packet is waiting for transmission, it will be sent before any low-priority packets that its neighbors may have. For example, it is possible to add a reservation field to each packet, which can be increased by any processor as the packet goes by. When the packet has gone all the way around, the reservation field indicates the priority class of the next packet. When the current sender is finished transmitting, it regenerates a token bearing this priority class. Only processors with a pending packet of this class may capture it, and then only to send one packet. Of course, this scheme means that the upper bound of \( kn \) byte times now applies only to packets of the highest priority class.

An alternative to a token ring is the TDMA (Time Division Multiple Access) protocol shown in Fig. 4-28. Here traffic is organized in fixed-size frames, each of which contains \( n \) slots. Each slot is assigned to one processor, which may use it to transmit a packet when its time comes. In this way collisions are avoided, the delay is bounded, and each processor gets a guaranteed fraction of the bandwidth, depending on how many slots per frame it has been assigned.
Real-time distributed systems operating over wide-area networks have the same need for predictability as those confined to a room or building. The communication in these systems is invariably connection oriented. Often, there is the ability to establish **real-time connections** between distant machines. When such a connection is established, the quality of service is negotiated in advance between the network users and the network provider. This quality may involve a guaranteed maximum delay, maximum jitter (variance of packet delivery times), minimum bandwidth, and other parameters. To make good on its guarantees, the network may have to reserve memory buffers, table entries, CPU cycles, link capacity, and other resources for this connection throughout its lifetime. The user is likely to be charged for these resources, whether or not they are used, since they are not available to other connections.

A potential problem with wide-area real-time distributed systems is their relatively high packet loss rates. Standard protocols deal with packet loss by setting a timer when each packet is transmitted. If the timer goes off before the acknowledgement is received, the packet is sent again. In real-time systems, this kind of unbounded transmission delay is rarely acceptable.

One easy solution is for the sender *always* to transmit each packet two (or more) times, preferably over independent connections if that option is available. Although this scheme wastes at least half the bandwidth, if one packet in, say, 10.5 is lost, only one time in 10.10 will both copies be lost. If a packet takes a millisecond, this works out to one lost packet every four months. With three transmissions, one packet is lost every 30,000 years. The net effect of multiple transmissions of every packet right from the start is a low and bounded delay virtually all the time.

### The Time-Triggered Protocol

On account of the constraints on real-time distributed systems, their protocols are often quite unusual. In this section we will examine one such protocol, **TTP (Time-Triggered Protocol)** (Kopetz and Grunsteidl, 1994), which is as different from the Ethernet protocol as a Victorian drawing room is from a Wild West saloon. TTP is used in the MARS real-time system (Kopetz et al., 1989) and is intertwined with it in many ways, so we will refer to properties of MARS where necessary.

A node in MARS consists of at least one CPU, but often two or three work together to present the image of a single fault-tolerant, fail-silent node to the outside world. The nodes in MARS are connected by two reliable and independent TDMA broadcast networks. All packets are sent on both networks in parallel. The expected loss rate is one packet every 30 million years.

MARS is a time-triggered system, so clock synchronization is critical. Time is discrete, with clock ticks generally occurring every microsecond. TTP assumes that all the clocks are synchronized with a precision on the order of tens of microseconds. This precision is possible because the protocol itself provides continuous clock synchronization and has been designed to allow it to be done in hardware to extremely high precision.

All nodes in MARS are aware of the programs being run on all the other nodes. In particular, all nodes know when a packet is to be sent by another node and can detect its presence or absence easily. Since packets are assumed not to be lost (see above), the absence of a packet at a moment when one is expected means that the sending node has crashed.

For example, suppose that some exceptional event is detected and a packet is broadcast to tell everyone else about it. Node 6 is expected to make some computation and then broadcast a reply after 2 msec in slot 15 of the TDMA frame. If the message is not forthcoming in the expected slot, the other nodes assume that node 6 has gone down, and take whatever steps are necessary...
to recover from its failure. This tight bound and instant consensus eliminate the need for time-
consuming agreement protocols and allow the system to be both fault tolerant and operate in real
time.

Every node maintains the global state of the system. These states are required to be identical
everywhere. It is a serious (and detectable) error if someone is out of step with the rest. The global
state consists of three components:

1. The current mode.
2. The global time.
3. A bit map giving the current system membership.

The mode is defined by the application and has to do with which phase the system is in. For
example, in a space application, the countdown, launch, flight, and landing might all be separate
modes. Each mode has its own set of processes and the order in which they run, list of
participating nodes, TDMA slot assignments, message names and formats, and legal successor
modes.

The second field in the global state is the global time. Its granularity is application defined,
but in any event must be coarse enough that all nodes agree on it. The third field keeps track of
which nodes are up and which are down.

Unlike the OSI and Internet protocol suites, the TTP protocol consists of a single layer that
handles end-to-end data transport, clock synchronization, and membership management. A typical
packet format is illustrated in Fig. 4-29. It consists of a start-of-packet field, a control field, a data
field, and a CRC field.

![Fig. 4-29. A typical TTP packet.](image)

The control field contains a bit used to initialize the system (more about which later), a
subfield for changing the current mode, and a subfield for acknowledging the packets sent by the
preceding node (according to the current membership list). The purpose of this field is to let the
previous node know that it is functioning correctly and its packets are getting onto the network as
they should be. If an expected acknowledgement is lacking, all nodes mark the expected sender as
down and expunge it from the membership bit maps in their current state. The rejected node is
expected to go along with being excommunicated without protest.

The data field contains whatever data are required. The CRC field is quite unusual, as it
provides a checksum over not only the packet contents, but over the sender's global state as well.
This means that if a sender has an incorrect global state, the CRC of any packets it sends will not
agree with the values the receivers compute using their states. The next sender will not
acknowledge the packet, and all nodes, including the one with the bad state, mark it as down in
their membership bit maps.
Periodically, a packet with the initialization bit is broadcast. This packet also contains the current global state. Any node that is marked as not being a member, but which is supposed to be a member in this mode, can now join as a passive member. If a node is supposed to be a member, it has a TDMA slot assigned, so there is no problem of when to respond (in its own TDMA slot). Once its packet has been acknowledged, all the other nodes mark it as being active (operational) again.

A final interesting aspect of the protocol is the way it handles clock synchronization. Because each node knows the time when TDMA frames start and the position of its slot within the frame, it knows exactly when to begin its packet. This scheme avoids collisions. However, it also contains valuable timing information. If a packet begins $n$ microseconds before or after it is supposed to, each other node can detect this tardiness and use it as an estimate of the skew between its clock and the sender's clock. By monitoring the starting position of every packet, a node might learn, for example, that every other node appears to be starting its transmissions 10 microseconds too late. In this case it can reasonably conclude that its own clock is actually 10 microseconds fast and make the necessary correction. By keeping a running average of the earliness or lateness of all other packets, each node can adjust its clock continuously to keep it in sync with the others without running any special clock management protocol.

In summary, the unusual properties of TTP are the detection of lost packets by the receivers, not the senders, the automatic membership protocol, the CRC on the packet plus global state, and the way that clock synchronization is done.

### 4.6.4. Real-Time Scheduling

Real-time systems are frequently programmed as a collection of short tasks (processes or threads), each with a well-defined function and a well-bounded execution time. The response to a given stimulus may require multiple tasks to be run, generally with constraints on their execution order. In addition, a decision has to be made about which tasks to run on which processors. In this section we will deal with some of the issues concerning task scheduling in real-time systems.

Real-time scheduling algorithms can be characterized by the following parameters:

1. Hard real time versus soft real time.
2. Preemptive versus nonpreemptive scheduling.
3. Dynamic versus static.

Hard real-time algorithms must guarantee that all deadlines are met. Soft realtime algorithms can live with a best efforts approach. The most important case is hard real time.

Preemptive scheduling allows a task to be suspended temporarily when a higher-priority task arrives, resuming it later when no higher-priority tasks are available to run. Nonpreemptive scheduling runs each task to completion. Once a task is started, it continues to hold its processor until it is done. Both kinds of scheduling strategies are used.

Dynamic algorithms make their scheduling decisions during execution. When an event is detected, a dynamic preemptive algorithm decides on the spot whether to run the (first) task associated with the event or to continue running the current task. A dynamic nonpreemptive algorithm just notes that another task is runnable. When the current task finishes, a choice among the now-ready tasks is made.

With static algorithms, in contrast, the scheduling decisions, whether preemptive or not, are made in advance, before execution. When an event occurs, the runtime scheduler just looks in a table to see what to do.

Finally, scheduling can be centralized, with one machine collecting all the information and making all the decisions, or it can be decentralized, with each processor making its own decisions. In the centralized case, the assignment of tasks to processors can be made at the same time. In
the decentralized case, assigning tasks to processors is distinct from deciding which of the tasks assigned to a given processor to run next.

A key question that all real-time system designers face is whether or not it is even possible to meet all the constraints. If a system has one processor and it gets 60 interrupts/sec, each requiring 50 msec of work, the designers have a Big Problem on their hands.

Suppose that a periodic real-time distributed system has \( m \) tasks and \( N \) processors to run them on. Let \( C_i \) be the CPU time needed by task \( i \) and let \( P_i \) be its period, that is, the time between consecutive interrupts. To be feasible, the utilization of the system, \( \mu \), must be related to \( N \) by the equation

\[
\mu = \frac{\sum_{i=1}^{m} C_i}{\sum_{i=1}^{m} P_i} \leq N
\]

For example, if a task is started every 20 msec and runs for 10 msec each time, it uses up 0.5 CPUs. Five such tasks would need three CPUs to do the job. A set of tasks that meets the foregoing requirement is said to be schedulable. Note that the equation above really gives a lower bound on the number of CPUs needed, since it ignores task switching time, message transport, and other sources of overhead, and assumes that optimal scheduling is possible.

In the following two sections we will look at dynamic and static scheduling, respectively, of sets of periodic tasks. For additional information, see (Ramamirtham et al., 1990; and Schwan and Zhou, 1992).

## Dynamic Scheduling

Let us look first at a few of the better-known dynamic scheduling algorithms — algorithms that decide during program execution which task to run next. The classic algorithm is the rate monotonic algorithm (Liu and Layland, 1973). It was designed for preemptively scheduling periodic tasks with no ordering or mutual exclusion constraints on a single processor. It works like this. In advance, each task is assigned a priority equal to its execution frequency. For example, a task run every 20 msec is assigned priority 50 and a task run every 100 msec is assigned priority 10. At run time, the scheduler always selects the highest priority task to run, preempting the current task if need be. Liu and Layland proved that this algorithm is optimal. They also proved that any set of tasks meeting the utilization condition

\[
\mu = \frac{\sum_{i=1}^{m} C_i}{\sum_{i=1}^{m} P_i} \leq m(2^{1/m} - 1)
\]

is schedulable using the rate monotonic algorithm. The right-hand side converges to \( \ln 2 \) (about 0.693) as \( m \to \infty \). In practice, this limit is overly pessimistic; a set of tasks with \( \mu \) as high as 0.88 can usually be scheduled.

A second popular preemptive dynamic algorithm is earliest deadline first. Whenever an event is detected, the scheduler adds it to the list of waiting tasks. This list is always kept sorted by deadline, closest deadline first. (For a periodic task, the deadline is the next occurrence.) The scheduler then just chooses the first task on the list, the one closest to its deadline. Like the rate monotonic algorithm, it produces optimal results, even for task sets with \( \mu = 1 \).

A third preemptive dynamic algorithm first computes for each task the amount of time it has to spare, called the laxity (slack). For a task that must finish in 200 msec but has another 150 msec to run, the laxity is 50 msec. This algorithm, called least laxity, chooses the task with the least laxity, that is, the one with the least breathing room.

None of the algorithms above are provably optimal in a distributed system, but they can be used as heuristics. Also, none of them takes into account order or mutex constraints, even on a
uniprocessor, which makes them less useful in practice than they are in theory. Consequently, many practical systems use static scheduling when enough information is available. Not only can static algorithms take side constraints into account, but they have very low overhead at run time.

**Static Scheduling**

Static scheduling is done before the system starts operating. The input consists of a list of all the tasks and the times that each must run. The goal is to find an assignment of tasks to processors and for each processor, a static schedule giving the order in which the tasks are to be run. In theory, the scheduling algorithm can run an exhaustive search to find the optimal solution, but the search time is exponential in the number of tasks (Ullman, 1976), so heuristics of the type described above are generally used. Rather than simply give some additional heuristics here, we will go into one example in detail, to show the interplay of scheduling and communication in a real-time distributed system with nonpreemptive static scheduling (Kopetz et al., 1989).

Let us assume that every time a certain event is detected, task 1 is started on processor \( A \), as shown in Fig. 4-30. This task, in turn, starts up additional tasks, both locally and on a second processor, \( B \). For simplicity, we will assume that the assignment of tasks to processors is dictated by external considerations (which task needs access to which I/O device) and is not a parameter here. All tasks take 1 unit of CPU time.

Task 1 starts up tasks 2 and 3 on its machine, as well as task 7 on processor \( B \). Each of these three tasks starts up another task, and so on, as illustrated. The arrows indicate messages being sent between tasks. In this simple example, it is perhaps easiest to think of \( X \rightarrow Y \) as meaning that \( Y \) cannot start until a message from \( X \) has arrived. Some tasks, such as 8, require two messages before they may start. The cycle is completed when task 10 has run and generated the expected response to the initial stimulus.

![Fig. 4-30. Ten real-time tasks to be executed on two processors.](image)

After task 1 has completed, tasks 2 and 3 are both runnable. The scheduler has a choice of which one to schedule next. Suppose that it decides to schedule task 2 next. It then has a choice between tasks 3 and 4 as the successor to task 2. If it chooses task 3, it then has a choice between tasks 4 and 5 next. However, if it chooses 4 instead of 3, it must run 3 following 4, because 6 is not enabled yet, and will not be until both 5 and 9 have run.
Meanwhile, activity is also occurring in parallel on processor B. As soon as task 1 has initiated it, task 7 can start on B, at the same time as either 2 or 3. When both 5 and 7 have finished, task 8 can be started, and so on. Note that task 6 requires input from 4, 5, and 9 to start, and produces output for 10.

![Diagram showing two schedules](image)

**Fig. 4-31.** Two possible schedules for the tasks of Fig. 4-30.

Two potential schedules are given in Fig. 4-31(a) and (b). Messages between tasks on different processors are depicted as arrows here; messages between tasks on the same machine are handled internally and are not shown. Of the two schedules illustrated, the one in Fig. 4-31(b) is a better choice because it allows task 5 to run early, thus making it possible for task 8 to start earlier. If task 5 is delayed significantly, as in Fig. 4-31(a), then tasks 8 and 9 are delayed, which also means that 6 and eventually 10 are delayed, too.

It is important to realize that with static scheduling, the decision of whether to use one of these schedules, or one of several alternatives is made by the scheduler in advance, before the system starts running. It analyzes the graph of Fig. 4-30, also using as input information about the running times of all the tasks, and then applies some heuristic to find a good schedule. Once a schedule has been selected, the choices made are incorporated into tables so that at run time a simple dispatcher can carry out the schedule with little overhead.

Now let us consider the problem of scheduling the same tasks again, but this time taking communication into account. We will use TDMA communication, with eight slots per TDMA frame. In this example, a TDMA slot is equal to one-fourth of a task execution time. We will arbitrarily assign slot 1 to processor A and slot 5 to processor B. The assignment of TDMA slots to processors is up to the static scheduler and may differ between program phases.

In Fig. 4-32 we show both schedules of Fig. 4-31, but now taking the use of TDMA slots into account. A task may not send a message until a slot owned by its processor comes up. Thus, task 5 may not send to task 8 until the first slot of the next TDMA frame occurs in rotation, requiring a delay in starting task 8 in Fig. 4-32(a) that was not present before.
The important thing to notice about this example is that the runtime behavior is completely deterministic, and known even before the program starts executing. As long as communication and processor errors do not occur, the system will always meet its real-time deadlines. Processor failures can be masked by having each node consist of two or more CPUs actively tracking each other. Some extra time may have to be statically allocated to each task interval to allow for recovery, if need be. Lost or garbled packets can be handled by having every one sent twice initially, either on disjoint networks or on one network by making the TDMA slots two packets wide.

It should be clear by now that real-time systems do not try to squeeze the last drop of performance out of the hardware, but rather use extra resources to make sure that the real-time constraints are met under all conditions. However, the relatively low use of the communication bandwidth in our example is not typical. It is a consequence of this example using only two processors with modest communication requirements. Practical real-time systems have many processors and extensive communication.

A Comparison of Dynamic versus Static Scheduling

The choice of dynamic or static scheduling is an important one and has far-reaching consequences for the system. Static scheduling is a good fit with a time-triggered design, and dynamic scheduling is a good fit for an event-triggered design. Static scheduling must be carefully planned in advance, with considerable effort going into choosing the various parameters. Dynamic scheduling does not require as much advance work, since scheduling decisions are made on-the-fly, during execution.

Dynamic scheduling can potentially make better use of resources than static scheduling. In the latter, the system must frequently be overdimensioned to have so much capacity that it can handle even the most unlikely cases. However, in a hard real-time system, wasting resources is often the price that must be paid to guarantee that all deadlines will be met.

On the other hand, given enough computing power, an optimal or nearly optimal schedule can be derived in advance for a static system. For an application such as reactor control, it may well be worth investing months of CPU time to find the best schedule. A dynamic system cannot afford the luxury of a complex scheduling calculation during execution, so to be safe, may have to be heavily overdimensioned as well, and even then, there is no guarantee that it will meet its specifications. Instead, extensive testing is required.

As a final thought, it should be pointed out that our discussion has simplified matters considerably. For example, tasks may need access to shared variables, so these have to be reserved in advance. Often there are scheduling constraints, which we have ignored. Finally, some systems do advance planning during runtime, making them hybrids between static and dynamic.
UNIT V

Distributed File Systems

A key component of any distributed system is the file system. As in single processor systems, in distributed systems the job of the file system is to store programs and data and make them available as needed. Many aspects of distributed file systems are similar to conventional file systems, so we will not repeat that material here. Instead, we will concentrate on those aspects of distributed file systems that are different from centralized ones.

To start with, in a distributed system, it is important to distinguish between the concepts of the file service and the file server. The file service is the specification of what the file system offers to its clients. It describes the primitives available, what parameters they take, and what actions they perform. To the clients, the file service defines precisely what service they can count on, but says nothing about how it is implemented. In effect, the file service specifies the file system’s interface to the clients.

A file server, in contrast, is a process that runs on some machine and helps implement the file service. A system may have one file server or several. In particular, they should not know how many file servers there are and what the location or function of each one is. All they know is that when they call the procedures specified in the file service, the required work is performed somehow, and the required results are returned. In fact, the clients should not even know that the file service is distributed. Ideally, it should look the same as a normal single-processor file system.

Since a file server is normally just a user process (or sometimes a kernel process) running on some machine, a system may contain multiple file servers, each offering a different file service. For example, a distributed system may have two servers that offer UNIX file service and MS-DOS file service, respectively, with each user process using the one appropriate for it. In that way, it is possible to have a terminal with multiple windows, with UNIX programs running in some windows and MS-DOS programs running in other windows, with no conflicts. Whether the servers offer specific file services, such as UNIX or MS-DOS, or more general file services is up to the system designers. The type and number of file services available may even change as the system evolves.

5.1. DISTRIBUTED FILE SYSTEM DESIGN

A distributed file system typically has two reasonably distinct components: the true file service and the directory service. The former is concerned with the operations on individual files, such as reading, writing, and appending, whereas the latter is concerned with creating and managing directories, adding and deleting files from directories, and so on. In this section we will discuss the true file service interface; in the next one we will discuss the directory service interface.

5.1.1. The File Service Interface

For any file service, whether for a single processor or for a distributed system, the most fundamental issue is: What is a file? In many systems, such as UNIX and MS-DOS, a file is an uninterpreted sequence of bytes. The meaning and structure of the information in the files is entirely up to the application programs; the operating system is not interested.

On mainframes, however, many types of files exist, each with different properties. A file can be structured as a sequence of records, for example, with operating system calls to read or write a particular record. The record can usually be specified by giving either its record number (i.e., position within the file) or the value of some field. In the latter case, the operating system either maintains the file as a B-tree or other suitable data structure, or uses hash tables to locate records quickly. Since most distributed systems are intended for UNIX or MS-DOS environments, most file servers support the notion of a file as a sequence of bytes rather than as a sequence of keyed records.
A file can have attributes, which are pieces of information about the file but which are not part of the file itself. Typical attributes are the owner, size, creation date, and access permissions. The file service usually provides primitives to read and write some of the attributes. For example, it may be possible to change the access permissions but not the size (other than by appending data to the file). In a few advanced systems, it may be possible to create and manipulate user-defined attributes in addition to the standard ones.

Another important aspect of the file model is whether files can be modified after they have been created. Normally, they can be, but in some distributed systems, the only file operations are CREATE and READ. Once a file has been created, it cannot be changed. Such a file is said to be immutable. Having files be immutable makes it much easier to support file caching and replication because it eliminates all the problems associated with having to update all copies of a file whenever it changes.

Protection in distributed systems uses essentially the same techniques as in single-processor systems: capabilities and access control lists. With capabilities, each user has a kind of ticket, called a capability, for each object to which it has access. The capability specifies which kinds of accesses are permitted (e.g., reading is allowed but writing is not).

All access control list schemes associate with each file a list of users who may access the file and how. The UNIX scheme, with bits for controlling reading, writing, and executing each file separately for the owner, the owner's group, and everyone else is a simplified access control list.

File services can be split into two types, depending on whether they support an upload/download model or a remote access model. In the upload/download model, shown in fig. 5-1(a), the file service provides only two major operations: read file and write file. The former operation transfers an entire file from one of the file servers to the requesting client. The latter operation transfers an entire file the other way, from client to server. Thus the conceptual model is moving whole files in either direction. The files can be stored in memory or on a local disk, as needed.

![Fig. 5-1. (a) The upload/download model. (b) The remote access model.](image-url)

The advantage of the upload/download model is its conceptual simplicity. Application programs fetch the files they need, then use them locally. Any modified files or newly created files are written back when the program finishes. No complicated file service interface has to be mastered to use this model. Furthermore, whole file transfer is highly efficient. However, enough storage must be available on the client to store all the files required. Furthermore, if only a fraction of a file is needed, moving the entire file is wasteful.
The other kind of file service is the **remote access model**, as illustrated in Fig. 5-1(b). In this model, the file service provides a large number of operations for opening and closing files, reading and writing parts of files, moving around within files (LSEEK), examining and changing file attributes, and so on. Whereas in the upload/download model, the file service merely provides physical storage and transfer, here the file system runs on the servers, not on the clients. It has the advantage of not requiring much space on the clients, as well as eliminating the need to pull in entire files when only small pieces are needed.

### 5.1.2. The Directory Server Interface

The other part of the file service is the directory service, which provides operations for creating and deleting directories, naming and renaming files, and moving them from one directory to another. The nature of the directory service does not depend on whether individual files are transferred in their entirety or accessed remotely.

The directory service defines some alphabet and syntax for composing file (and directory) names. File names can typically be from 1 to some maximum number of letters, numbers, and certain special characters. Some systems divide file names into two parts, usually separated by a period, such as `prog.c` for a C program or `man.txt` for a text file. The second part of the name, called the **file extension**, identifies the file type. Other systems use an explicit attribute for this purpose, instead of tacking an extension onto the name.

All distributed systems allow directories to contain subdirectories, to make it possible for users to group related files together. Accordingly, operations are provided for creating and deleting directories as well as entering, removing, and looking up files in them. Normally, each subdirectory contains all the files for one project, such as a large program or document (e.g., a book). When the (sub)directory is listed, only the relevant files are shown; unrelated files are in other (sub)directories and do not clutter the listing. Subdirectories can contain their own subdirectories, and so on, leading to a tree of directories, often called a **hierarchical file system**. Figure 5-2(a) illustrates a tree with five directories.

In some systems, it is possible to create links or pointers to an arbitrary directory. These can be put in any directory, making it possible to build not only trees, but arbitrary directory graphs, which are more powerful. The distinction between trees and graphs is especially important in a distributed system.

![Fig. 5-2](image)

**Fig. 5-2.** (a) A directory tree contained on one machine. (b) A directory graph on two machines.
The nature of the difficulty can be seen by looking at the directory graph of Fig. 5-2(b). In this figure, directory \( D \) has a link to directory \( B \). The problem occurs when the link from \( A \) to \( B \) is removed. In a tree-structured hierarchy, a link to a directory can be removed only when the directory pointed to is empty. In a graph, it is allowed to remove a link to a directory as long as at least one other link exists. By keeping a reference count, shown in the upper right-hand corner of each directory in Fig. 5-2(b), it can be determined when the link being removed is the last one.

After the link from \( A \) to \( B \) is removed, \( B \)'s reference count is reduced from 2 to 1, which on paper is fine. However, \( B \) is now Unreachable from the root of the file system (\( A \)). The three directories, \( B, D, \) and \( E \), and all their files are effectively orphans.

This problem exists in centralized systems as well, but it is more serious in distributed ones. If everything is on one machine, it is possible, albeit somewhat expensive, to discover orphaned directories, because all the information is in one place. All file activity can be stopped and the graph traversed starting at the root, marking all reachable directories. At the end of this process, all unmarked directories are known to be Unreachable. In a distributed system, multiple machines are involved and all activity cannot be stopped, so getting a "snapshot" is difficult, if not impossible.

A key issue in the design of any distributed file system is whether or not all machines (and processes) should have exactly the same view of the directory hierarchy. As an example of what we mean by this remark, consider Fig. 5-3. In Fig. 5-3(a) we show two file servers, each holding three directories and some files. In Fig. 5-3(b) we have a system in which all clients (and other machines) have the same view of the distributed file system. If the path /\( D/E/x \) is valid on one machine, it is valid on all of them.
Fig. 5-3. (a) Two file servers. The squares are directories and the circles are files. (b) A system in which all clients have the same view of the file system. (c) A system in which different clients may have different views of the file system.

In contrast, in Fig. 5-3(c), different machines can have different views of the file system. To repeat the preceding example, the path /D/E/x might well be valid on client 1 but not on client 2. In systems that manage multiple file servers by remote mounting, Fig. 5-3(c) is the norm. It is flexible and straightforward to implement, but it has the disadvantage of not making the entire system behave like a single old-fashioned timesharing system. In a timesharing system, the file system looks the same to any process [i.e., the model of Fig. 5-3(b)]. This property makes a system easier to program and understand.

A closely related question is whether or not there is a global root directory, which all machines recognize as the root. One way to have a global root directory is to have this root contain one entry for each server and nothing else. Under these circumstances, paths take the form /server/path, which has its own disadvantages, but at least is the same everywhere in the system.

Naming Transparency

The principal problem with this form of naming is that it is not fully transparent. Two forms of transparency are relevant in this context and are worth distinguishing. The first one, location transparency, means that the path name gives no hint as to where the file (or other object) is
located. A path like /server1/dir1/dir2/x tells everyone that x is located on server 1, but it does not tell where that server is located. The server is free to move anywhere it wants to in the network without the path name having to be changed. Thus this system has location transparency.

However, suppose that file x is extremely large and space is tight on server 1. Furthermore, suppose that there is plenty of room on server 2. The system might well like to move x to server 2 automatically. Unfortunately, when the first component of all path names is the server, the system cannot move the file to the other server automatically, even if dir1 and dir2 exist on both servers. The problem is that moving the file automatically changes its path name from /server1/dir1/dir2/x to /server2/dir1/dir2/x. Programs that have the former string built into them will cease to work if the path changes. A system in which files can be moved without their names changing is said to have location independence. A distributed system that embeds machine or server names in path names clearly is not location independent. One based on remote mounting is not either, since it is not possible to move a file from one file group (the unit of mounting) to another and still be able to use the old path name. Location independence is not easy to achieve, but it is a desirable property to have in a distributed system.

To summarize what we have said earlier, there are three common approaches to file and directory naming in a distributed system:

1. Machine + path naming, such as /machine/path or machine:path.
2. Mounting remote file systems onto the local file hierarchy.
3. A single name space that looks the same on all machines.

The first two are easy to implement, especially as a way to connect up existing systems that were not designed for distributed use. The latter is difficult and requires careful design, but it is needed if the goal of making the distributed system act like a single computer is to be achieved.

**Two-Level Naming**

Most distributed systems use some form of two-level naming. Files (and other objects) have symbolic names such as prog.c, for use by people, but they can also have internal, binary names for use by the system itself. What directories in fact really do is provide a mapping between these two naming levels. It is convenient for people and programs to use symbolic (ASCII) names, but for use within the system itself, these names are too long and cumbersome. Thus when a user opens a file or otherwise references a symbolic name, the system immediately looks up the symbolic name in the appropriate directory to get the binary name that will be used to locate the file. Sometimes the binary names are visible to the users and sometimes they are not.

The nature of the binary name varies considerably from system to system. In a system consisting of multiple file servers, each of which is self-contained (i.e., does not hold any references to directories or files on other file servers), the binary name can just be a local i-node number, as in UNIX.

A more general naming scheme is to have the binary name indicate both a server and a specific file on that server. This approach allows a directory on one server to hold a file on a different server. An alternative way to do the same thing that is sometimes preferred is to use a symbolic link. A symbolic link is a directory entry that maps onto a (server, file name) string, which can be looked up on the server named to find the binary name. The symbolic link itself is just a path name.

Yet another idea is to use capabilities as the binary names. In this method, looking up an ASCII name yields a capability, which can take one of many forms. For example, it can contain a physical or logical machine number or network address of the appropriate server, as well as a number indicating which specific file is required. A physical address can be used to send a message.
to the server without further interpretation. A logical address can be located either by broadcasting or by looking it up on a name server.

One last twist that is sometimes present in a distributed system but rarely in a centralized one is the possibility of looking up an ASCII name and getting not one but several binary names (i-nodes, capabilities, or something else). These would typically represent the original file and all its backups. Armed with multiple binary names, it is then possible to try to locate one of the corresponding files, and if that one is unavailable for any reason, to try one of the others. This method provides a degree of fault tolerance through redundancy.

5.1.3. Semantics of File Sharing

When two or more users share the same file, it is necessary to define the semantics of reading and writing precisely to avoid problems. In single-processor systems that permit processes to share files, such as UNIX, the semantics normally state that when a READ operation follows a WRITE operation, the READ returns the value just written, as shown in Fig. 5-4(a). Similarly, when two writes happen in quick succession, followed by a READ, the value read is the value stored by the last write. In effect, the system enforces an absolute time ordering on all operations and always returns the most recent value. We will refer to this model as UNIX semantics. This model is easy to understand and straightforward to implement.

![Diagram](https://example.com/diagram.png)

**Fig. 5-4.** (a) On a single processor, when a READ follows a WRITE, the value returned by the read is the value just written. (b) In a distributed system with caching, obsolete values may be returned.
In a distributed system, UNIX semantics can be achieved easily as long as there is only one file server and clients do not cache files. All reads and writes go directly to the file server, which processes them strictly sequentially. This approach gives UNIX semantics (except for the minor problem that network delays may cause a read that occurred a microsecond after a write to arrive at the server first and thus get the old value).

In practice, however, the performance of a distributed system in which all file requests must go to a single server is frequently poor. This problem is often solved by allowing clients to maintain local copies of heavily used files in their private caches. Although we will discuss the details of file caching below, for the moment it is sufficient to point out that if a client locally modifies a cached file and shortly thereafter another client reads the file from the server, the second client will get an obsolete file, as illustrated in Fig. 5-4(b).

One way out of this difficulty is to propagate all changes to cached files back to the server immediately. Although conceptually simple, this approach is inefficient. An alternative solution is to relax the semantics of file sharing. Instead of requiring a read to see the effects of all previous writes, one can have a new rule that says: "Changes to an open file are initially visible only to the process (or possibly machine) that modified the file. Only when the file is closed are the changes made visible to other processes (or machines)." The adoption of such a rule does not change what happens in Fig. 5-4(b), but it does redefine the actual behavior (B getting the original value of the file) as being the correct one. When A closes the file, it sends a copy to the server, so that subsequent reads get the new value, as required. This rule is widely implemented and is known as session semantics.

Using session semantics raises the question of what happens if two or more clients are simultaneously caching and modifying the same file. One solution is to say that as each file is closed in turn, its value is sent back to the server, so the final result depends on who closes last. A less pleasant, but slightly easier to implement alternative is to say that the final result is one of the candidates, but leave the choice of which one unspecified.

One final difficulty with using caching and session semantics is that it violates another aspect of the UNIX semantics in addition to not having all reads return the value most recently written. In UNIX, associated with each open file is a pointer that indicates the current position in the file. Reads take data starting at this position and writes deposit data there. This pointer is shared between the process that opened the file and all its children. With session semantics, when the children run on different machines, this sharing cannot be achieved.

To see what the consequences of having to abandon shared file pointers are, consider a command like

```
run >out
```

where run is a shell script that executes two programs, a and b, one after another. If both programs produce output, it is expected that the output produced by b will directly follow the output from a within out. The way this is achieved is that when b starts up, it inherits the file pointer from a, which is shared by the shell and both processes. In this way, the first byte that b writes directly follows the last byte written by a. With session semantics and no shared file pointers, a completely different mechanism is needed to make shell scripts and similar constructions that use shared file pointers work. Since no general-purpose solution to this problem is known, each system must deal with it in an ad hoc way.

A completely different approach to the semantics of file sharing in a distributed system is to make all files immutable. There is thus no way to open a file for writing. In effect, the only operations on files are create and read.

What is possible is to create an entirely new file and enter it into the directory system under the name of a previous existing file, which now becomes inaccessible (at least under that name). Thus although it becomes impossible to modify the file x, it remains possible to replace x by a new
file atomically. In other words, although files cannot be updated, directories can be. Once we have decided that files cannot be changed at all, the problem of how to deal with two processes, one of which is writing on a file and the other of which is reading it, simply disappears.

What does remain is the problem of what happens when two processes try to replace the same file at the same time. As with session semantics, the best solution here seems to be to allow one of the new files to replace the old one, either the last one or nondeterministically.

A somewhat stickier problem is what to do if a file is replaced while another process is busy reading it. One solution is to somehow arrange for the reader to continue using the old file, even if it is no longer in any directory, analogous to the way UNIX allows a process that has a file open to continue using it, even after it has been deleted from all directories. Another solution is to detect that the file has changed and make subsequent attempts to read from it fail.

A fourth way to deal with shared files in a distributed system is to use atomic transactions, as we discussed in detail in Chap. 3. To summarize briefly, to access a file or a group of files, a process first executes some type of BEGIN TRANSACTION primitive to signal that what follows must be executed indivisibly. Then come system calls to read and write one or more files. When the work has been completed, an END TRANSACTION primitive is executed. The key property of this method is that the system guarantees that all the calls contained within the transaction will be carried out in order, without any interference from other, concurrent transactions. If two or more transactions start up at the same time, the system ensures that the final result is the same as if they were all run in some (undefined) sequential order.

The classical example of where transactions make programming much easier is in a banking system. Imagine that a certain bank account contains 100 dollars, and that two processes are each trying to add 50 dollars to it. In an unconstrained system, each process might simultaneously read the file containing the current balance (100), individually compute the new balance (150), and successively overwrite the file with this new value. The final result could either be 150 or 200, depending on the precise timing of the reading and writing. By grouping all the operations into a transaction, interleaving cannot occur and the final result will always be 200.

In Fig. 5-5 we summarize the four approaches we have discussed for dealing with shared files in a distributed system.

<table>
<thead>
<tr>
<th>Method</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX semantics</td>
<td>Every operation on a file is instantly visible to all processes</td>
</tr>
<tr>
<td>Session semantics</td>
<td>No changes are visible to other processes until the file is closed</td>
</tr>
<tr>
<td>Immutable files</td>
<td>No updates are possible; simplifies sharing and replication</td>
</tr>
<tr>
<td>Transactions</td>
<td>All changes have the all-or-nothing property</td>
</tr>
</tbody>
</table>

**Fig. 5-5.** Four ways of dealing with the shared files in a distributed system.

**5.2. DISTRIBUTED FILE SYSTEM IMPLEMENTATION**

In the preceding section, we have described various aspects of distributed file systems from the user's perspective, that is, how they appear to the user. In this section we will see how these systems are implemented. We will start out by presenting some experimental information about file usage. Then we will go on to look at system structure, the implementation of caching, replication in
distributed systems, and concurrency control. We will conclude with a short discussion of some lessons that have been learned from experience.

5.2.1. File Usage

Before implementing any system, distributed or otherwise, it is useful to have a good idea of how it will be used, to make sure that the most commonly executed operations will be efficient. To this end, Satyanarayanan (1981) made a study of file usage patterns. We will present his major results below.

However, first, a few words of warning about these and similar measurements are in order. Some of the measurements are static, meaning that they represent a snapshot of the system at a certain instant. Static measurements are made by examining the disk to see what is on it. These measurements include the distribution of file sizes, the distribution of file types, and the amount of storage occupied by files of various types and sizes. Other measurements are dynamic, made by modifying the file system to record all operations to a log for subsequent analysis. These data yield information about the relative frequency of various operations, the number of files open at any moment, and the amount of sharing that takes place. By combining the static and dynamic measurements, even though they are fundamentally different, we can get a better picture of how the file system is used.

One problem that always occurs with measurements of any existing system is knowing how typical the observed user population is. Satyanarayanan's measurements were made at a university. Do they also apply to industrial research labs? To office automation projects? To banking systems? No one really knows for sure until these systems, too, are instrumented and measured.

Another problem inherent in making measurements is watching out for artifacts of the system being measured. As a simple example, when looking at the distribution of file names in an MS-DOS system, one could quickly conclude that file names are never more than eight characters (plus an optional three-character extension). However, it would be a mistake to draw the conclusion that eight characters are therefore enough, since nobody ever uses more than eight characters. Since MS-DOS does not allow more than eight characters in a file name, it is impossible to tell what users would do if they were not constrained to eight-character file names.

Finally, Satyanarayanan's measurements were made on more-or-less traditional UNIX systems. Whether or not they can be transferred or extrapolated to distributed systems is not really known.

This being said, the most important conclusions are listed in Fig. 5-6. From these observations, one can draw certain conclusions. To start with, most files are under 10K, which agrees with the results of Mullender and Tanenbaum (1984) made under different circumstances. This observation suggests that it may be feasible to transfer entire files rather than disk blocks between server and client. Since whole file transfer is typically simpler and more efficient, this idea is worth considering. Of course, some files are large, so provision has to be made for them too. Still, a good guideline is to optimize for the normal case and treat the abnormal case specially.

| Most files are small (less than 10 K) |
| Reading is much more common than writing |
| Reads and writes are sequential; random access is rare |
| Most files have a short lifetime |
File sharing is unusual

The average process uses only a few files

Distinct file classes with different properties exist

Fig. 5-6. Observed file system properties.

An interesting observation is that most files have short lifetimes. A common pattern is to create a file, read it (probably once), and then delete it. A typical usage might be a compiler that creates temporary files for transmitting information between its passes. The implication here is that it is probably a good idea to create the file on the client and keep it there until it is deleted. Doing so may eliminate a considerable amount of unnecessary client-server traffic.

The fact that few files are shared argues for client caching. As we have seen already, caching makes the semantics more complicated, but if files are rarely shared, it may well be best to do client caching and accept the consequences of session semantics in return for the better performance.

Finally, the clear existence of distinct file classes suggests that perhaps different mechanisms should be used to handle the different classes. System binaries need to be widespread but hardly ever change, so they should probably be widely replicated, even if this means that an occasional update is complex. Compiler and other temporary files are short, unshared, and disappear quickly, so they should be kept locally wherever possible. Electronic mailboxes are frequently updated but rarely shared, so replication is not likely to gain anything. Ordinary data files may be shared, so they may need still other handling.

5.2.2. System Structure

In this section we will look at some of the ways that file servers and directory servers are organized internally, with special attention to alternative approaches. Let us start with a very simple question: Are clients and servers different? Surprisingly enough, there is no agreement on this matter.

In some systems, there is no distinction between clients and servers. All machines run the same basic software, so any machine wanting to offer file service to the public is free to do so. Offering file service is just a matter of exporting the names of selected directories so that other machines can access them.

In other systems, the file server and directory server are just user programs, so a system can be configured to run client and server software on the same machines or not, as it wishes. Finally, at the other extreme, are systems in which clients and servers are fundamentally different machines, in terms of either hardware or software. The servers may even run a different version of the operating system from the clients. While separation of function may seem a bit cleaner, there is no fundamental reason to prefer one approach over the others.

A second implementation issue on which systems differ is how the file and directory service is structured. One organization is to combine the two into a single server that handles all the directory and file calls itself. Another possibility, however, is to keep them separate. In the latter case, opening a file requires going to the directory server to map its symbolic name onto its binary name (e.g., machine + i-node) and then going to the file server with the binary name to read or write the file.

Arguing in favor of the split is that the two functions are really unrelated, so keeping them separate is more flexible. For example, one could implement an MS-DOS directory server and a
UNIX directory server, both of which use the same file server for physical storage. Separation of
function is also likely to produce simpler software. Weighing against this is that having two servers
requires more communication.

Let us consider the case of separate directory and file servers for the moment. In the normal
case, the client sends a symbolic name to the directory server, which then returns the binary name
that the file server understands. However, it is possible for a directory hierarchy to be partitioned
among multiple servers, as illustrated in Fig. 5-7. Suppose, for example, that we have a system in
which the current directory, on server 1, contains an entry, $a$, for another directory on server 2.
Similarly, this directory contains an entry, $b$, for a directory on server 3. This third directory
contains an entry for a file $c$, along with its binary name.

To look up $a/b/c$, the client sends a message to server 1, which manages its current directory.
The server finds $a$, but sees that the binary name refers to another server. It now has a choice. It
can either tell the client which server holds $b$ and have the client look up $b/c$ there itself, as shown
in Fig. 5-7(a), or it can forward the remainder of the request to server 2 itself and not reply at all,
as shown in Fig. 5-7(b). The former scheme requires the client to be aware of which server holds
which directory, and requires more messages. The latter method is more efficient, but cannot be
handled using normal RPC since the process to which the client sends the message is not the one
that sends the reply.

Looking up path names all the time, especially if multiple directory servers are involved, can
be expensive. Some systems attempt to improve their performance by maintaining a cache of hints,
that is, recently looked up names and the results of these lookups. When a file is opened, the
cache is checked to see if the path name is there. If so, the directory-by-directory lookup is skipped
and the binary address is taken from the cache. If not, it is looked up.

![Fig. 5-7. (a) Iterative lookup of $a/b/c$. (b) Automatic lookup.](image-url)
For name caching to work, it is essential that when an obsolete binary name is used inadvertently, the client is somehow informed so it can fall back on the directory-by-directory lookup to find the file and update the cache. Furthermore, to make hint caching worthwhile in the first place, the hints have to be right most of the time. When these conditions are fulfilled, caching hints can be a powerful technique that is applicable in many distributed operating systems.

The final structural issue that we will consider here is whether or not file, directory, and other servers should maintain state information about clients. This issue is moderately controversial, with two competing schools of thought in existence.

One school thinks that servers should be stateless. In other words, when a client sends a request to a server, the server carries out the request, sends the reply, and then removes from its internal tables all information about the request. Between requests, no client-specific information is kept on the server. The other school of thought maintains that it is all right for servers to maintain state information about clients between requests. After all, centralized operating systems maintain state information about active processes, so why should this traditional behavior suddenly become unacceptable?

To better understand the difference, consider a file server that has commands to open, read, write, and close files. After a file has been opened, the server must maintain information about which client has which file open. Typically, when a file is opened, the client is given a file descriptor or other number which is used in subsequent calls to identify the file. When a request comes in, the server uses the file descriptor to determine which file is needed. The table mapping the file descriptors onto the files themselves is state information.

With a stateless server, each request must be self-contained. It must contain the full file name and the offset within the file, in order to allow the server to do the work. This information increases message length.

Another way to look at state information is to consider what happens if a server crashes and all its tables are lost forever. When the server is rebooted, it no longer knows which clients have which files open. Subsequent attempts to read and write open files will then fail, and recovery, if possible at all, will be entirely up to the clients. As a consequence, stateless servers tend to be more fault tolerant than those that maintain state, which is one of the arguments in favor of the former.

<table>
<thead>
<tr>
<th>Advantages of stateless servers</th>
<th>Advantages of stateful servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault tolerance</td>
<td>Shorter request messages</td>
</tr>
<tr>
<td>No OPEN/CLOSE calls needed</td>
<td>Better performance</td>
</tr>
<tr>
<td>No server space wasted on tables</td>
<td>Readahead possible</td>
</tr>
<tr>
<td>No limits on number of open files</td>
<td>Idempotency easier</td>
</tr>
<tr>
<td>No problems if a client crashes</td>
<td>File locking possible</td>
</tr>
</tbody>
</table>

**Fig. 5-8.** A comparison of stateless and stateful servers.
The arguments both ways are summarized in Fig. 5-8. Stateless servers are inherently more fault tolerant, as we just mentioned. OPEN and CLOSE calls are not needed, which reduces the number of messages, especially for the common case in which the entire file is read in a single blow. No server space is wasted on tables. When tables are used, if too many clients have too many files open at once, the tables can fill up and new files cannot be opened. Finally, with a stateful server, if a client crashes when a file is open, the server is in a bind. If it does nothing, its tables will eventually fill up with junk. If it times out inactive open files, a client that happens to wait too long between requests will be refused service, and correct programs will fail to function correctly. Statelessness eliminates these problems.

Stateful servers also have things going for them. Since READ and WRITE messages do not have to contain file names, they can be shorter, thus using less network bandwidth. Better performance is frequently possible since information about open files (in UNIX terms, the i-nodes) can be kept in main memory until the files are closed. Blocks can be read in advance to reduce delay, since most files are read sequentially. If a client ever times out and sends the same request twice, for example, APPEND, it is much easier to detect this with state (by having a sequence number in each message). Achieving idempotency in the face of unreliable communication with stateless operation takes more thought and effort. Finally, file locking is impossible to do in a truly stateless system, since the only effect setting a lock has is to enter state into the system. In stateless systems, file locking has to be done by a special lock server.

5.2.3. Caching

In a client-server system, each with main memory and a disk, there are four potential places to store files, or parts of files: the server's disk, the server's main memory, the client's disk (if available), or the client's main memory, as illustrated in Fig. 5-9. These different storage locations all have different properties, as we shall see.

![Diagram](image)

**Fig. 5-9.** Four places to store files or parts of files.

The most straightforward place to store all files is on the server's disk. There is generally plenty of space there and the files are then accessible to all clients. Furthermore, with only one copy of each file, no consistency problems arise.

The problem with using the server's disk is performance. Before a client can read a file, the file must be transferred from the server's disk to the server's main memory, and then again over the network to the client's main memory. Both transfers take time.

A considerable performance gain can be achieved by **caching** (i.e., holding) the most recently used files in the server's main memory. A client reading a file that happens to be in the server's cache eliminates the disk transfer, although the network transfer still has to be done. Since main memory is invariably smaller than the disk, some algorithm is needed to determine which files or parts of files should be kept in the cache.
This algorithm has two problems to solve. First, what is the unit the cache manages? It can be either whole files or disk blocks. If entire files are cached, they can be stored contiguously on the disk (or at least in very large chunks), allowing high-speed transfers between memory and disk and generally good performance. Disk block caching, however, uses cache and disk space more efficiently.

Second, the algorithm must decide what to do when the cache fills up and something must be evicted. Any of the standard caching algorithms can be used here, but because cache references are so infrequent compared to memory references, an exact implementation of LRU using linked lists is generally feasible. When something has to be evicted, the oldest one is chosen. If an up-to-date copy exists on disk, the cache copy is just discarded. Otherwise, the disk is first updated.

Having a cache in the server's main memory is easy to do and totally transparent to the clients. Since the server can keep its memory and disk copies synchronized, from the clients' point of view, there is only one copy of each file, so no consistency problems arise.

Although server caching eliminates a disk transfer on each access, it still has a network access. The only way to get rid of the network access is to do caching on the client side, which is where all the problems come in. The trade-off between using the client's main memory or its disk is one of space versus performance. The disk holds more but is slower. When faced with a choice between having a cache in the server's main memory versus the client's disk, the former is usually somewhat faster, and it is always much simpler. Of course, if large amounts of data are being used, a client disk cache may be better. In any event, most systems that do client caching do it in the client's main memory, so we will concentrate on that.

If the designers decide to put the cache in the client's main memory, three options are open as to precisely where to put it. The simplest is to cache files directly inside each user process' own address space, as shown in Fig. 5-10(b). Typically, the cache is managed by the system call library. As files are opened, closed, read, and written, the library simply keeps the most heavily used ones around, so that when a file is reused, it may already be available. When the process exits, all modified files are written back to the server. Although this scheme has an extremely low overhead, it is effective only if individual processes open and close files repeatedly. A data base manager process might fit this description, but in the usual program development environment, most processes read each file only once, so caching within the library wins nothing.

The second place to put the cache is in the kernel, as shown in Fig. 5-10(c).
The disadvantage here is that a kernel call is needed in all cases, even on a cache hit, but the fact that the cache survives the process more than compensates. For example, suppose that a two-pass compiler runs as two processes. Pass one writes an intermediate file read by pass two. In Fig. 5-10(c), after the pass one process terminates, the intermediate file will probably be in the cache, so no server calls will have to be made when the pass two process reads it in.

The third place for the cache is in a separate user-level cache manager process, as shown in Fig. 5-10(d). The advantage of a user-level cache manager is that it keeps the (micro)kernel free of file system code, is easier to program because it is completely isolated, and is more flexible.

On the other hand, when the kernel manages the cache, it can dynamically decide how much memory to reserve for programs and how much for the cache.

With a user-level cache manager running on a machine with virtual memory, it is conceivable that the kernel could decide to page out some or all of the cache to a disk, so that a so-called "cache hit" requires one or more pages to be brought in. Needless to say, this defeats the idea of client caching completely. However, if it is possible for the cache manager to allocate and lock in memory some number of pages, this ironic situation can be avoided.

When evaluating whether caching is worth the trouble at all, it is important to note that in Fig. 5-10(a), it takes exactly one RPC to make a file request, no matter what. In both Fig. 5-10(c) and Fig. 5-10(d) it takes either one or two, depending on whether or not the request can be satisfied out of the cache. Thus the mean number of RPCs is always greater when caching is used. In a situation in which RPCs are fast and network transfers are slow (fast CPUs, slow networks), caching

**Fig. 5-10.** Various ways of doing caching in client memory. (a) No caching. (b) Caching within each process. (c) Caching in the kernel. (d) The cache manager as a user process.
can give a big gain in performance. If, however, network transfers are very fast (e.g., with high-speed fiber optic networks), the network transfer time will matter less, so the extra RPCs may eat up a substantial fraction of the gain. Thus the performance gain provided by caching depends to some extent on the CPU and network technology available, and of course, on the applications.

**Cache Consistency**

As usual in computer science, you never get something for nothing. Client caching introduces inconsistency into the system. If two clients simultaneously read the same file and then both modify it, several problems occur. For one, when a third process reads the file from the server, it will get the original version, not one of the two new ones. This problem can be defined away by adopting session semantics (officially stating that the effects of modifying a file are not supposed to be visible globally until the file is closed). In other words, this "incorrect" behavior is simply declared to be the "correct" behavior. Of course, if the user expects UNIX semantics, the trick does not work.

Another problem, unfortunately, that cannot be defined away at all is that when the two files are written back to the server, the one written last will overwrite the other one. The moral of the story is that client caching has to be thought out fairly carefully. Below we will discuss some of the problems and proposed solutions.

One way to solve the consistency problem is to use the **write-through algorithm**. When a cache entry (file or block) is modified, the new value is kept in the cache, but is also sent immediately to the server. As a consequence, when another process reads the file, it gets the most recent value.

However, the following problem arises. Suppose that a client process on machine $A$ reads a file, $f$. The client terminates but the machine keeps $f$ in its cache. Later, a client on machine $B$ reads the same file, modifies it, and writes it through to the server. Finally, a new client process is started up on machine $A$. The first thing it does is open and read $f$, which is taken from the cache. Unfortunately, the value there is now obsolete.

A possible way out is to require the cache manager to check with the server before providing any client with a file from the cache. This check could be done by comparing the time of last modification of the cached version with the server's version. If they are the same, the cache is up-to-date. If not, the current version must be fetched from the server. Instead of using dates, version numbers or checksums can also be used. Although going to the server to verify dates, version numbers, or checksums takes an RPC, the amount of data exchanged is small. Still, it takes some time.

Another trouble with the write-through algorithm is that although it helps on reads, the network traffic for writes is the same as if there were no caching at all. Many system designers find this unacceptable, and cheat: instead of going to the server the instant the write is done, the client just makes a note that a file has been updated. Once every 30 seconds or so, all the file updates are gathered together and sent to the server at once. A single bulk write is usually more efficient than many small ones.

Besides, many programs create scratch files, write them, read them back, and then delete them, all in quick succession. In the event that this entire sequence happens before it is time to send all modified files back to the server, the now-deleted file does not have to be written back at all. Not having to use the file server at all for temporary files can be a major performance gain.

Of course, delaying the writes muddies the semantics, because when another process reads the file, what it gets depends on the timing. Thus postponing the writes is a trade-off between better performance and cleaner semantics (which translates into easier programming).
The next step in this direction is to adopt session semantics and write a file back to the server only after it has been closed. This algorithm is called write-on-close. Better yet, wait 30 seconds after the close to see if the file is going to be deleted. As we saw earlier, going this route means that if two cached files are written back in succession, the second one overwrites the first one. The only solution to this problem is to note that it is not nearly as bad as it first appears. In a single CPU system, it is possible for two processes to open and read a file, modify it within their respective address spaces, and then write it back. Consequently, write-on-close with session semantics is not that much worse than what can happen on a single CPU system.

A completely different approach to consistency is to use a centralized control algorithm. When a file is opened, the machine opening it sends a message to the file server to announce this fact. The file server keeps track of who has which file open, and whether it is open for reading, writing, or both. If a file is open for reading, there is no problem with letting other processes open it for reading, but opening it for writing must be avoided. Similarly, if some process has a file open for writing, all other accesses must be prevented. When a file is closed, this event must be reported, so the server can update its tables telling which client has which file open. The modified file can also be shipped back to the server at this point.

When a client tries to open a file and the file is already open elsewhere in the system, the new request can either be denied or queued. Alternatively, the server can send an unsolicited message to all clients having the file open, telling them to remove that file from their caches and disable caching just for that one file. In this way, multiple readers and writers can run simultaneously, with the results being no better and no worse than would be achieved on a single CPU system.

Although sending unsolicited messages is clearly possible, it is inelegant, since it reverses the client and server roles. Normally, servers do not spontaneously send messages to clients or initiate RPCs with them. If the clients are multithreaded, one thread can be permanently allocated to waiting for server requests, but if they are not, the unsolicited message must cause an interrupt.

Even with these precautions, one must be careful. In particular, if a machine opens, caches, and then closes a file, upon opening it again the cache manager must still check to see if the cache is valid. After all, some other process might have subsequently opened, modified, and closed the file. Many variations of this centralized control algorithm are possible, with differing semantics. For example, servers can keep track of cached files, rather than open files. All these methods have a single point of failure and none of them scale well to large systems.

<table>
<thead>
<tr>
<th>Method</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write through</td>
<td>Works, but does not affect write traffic</td>
</tr>
<tr>
<td>Delayed write</td>
<td>Better performance but possibly ambiguous semantics</td>
</tr>
<tr>
<td>Write on close</td>
<td>Matches session semantics</td>
</tr>
<tr>
<td>Centralized control</td>
<td>UNIX semantics, but not robust and scales poorly</td>
</tr>
</tbody>
</table>

Fig. 5-11. Four algorithms for managing a client file cache.
The four cache management algorithms discussed above are summarized in Fig. 5-11. To summarize the subject of caching as a whole, server caching is easy to do and almost always worth the trouble, independent of whether client caching is present or not. Server caching has no effect on the file system semantics seen by the clients. Client caching, in contrast, offers better performance at the price of increased complexity and possibly fuzzier semantics. Whether it is worth doing or not depends on how the designers feel about performance, complexity, and ease of programming.

Earlier in this chapter, when we were discussing the semantics of distributed file systems, we pointed out that one of the design options is immutable files. One of the great attractions of an immutable file is the ability to cache it on machine A without having to worry about the possibility that machine B will change it. Changes are not permitted. Of course, a new file may have been created and bound to the same symbolic name as the cached file, but this can be checked for whenever a cached file is reopened. This model has the same RPC overhead discussed above, but the semantics are less fuzzy.

### 5.2.4. Replication

Distributed file systems often provide file replication as a service to their clients. In other words, multiple copies of selected files are maintained, with each copy on a separate file server. The reasons for offering such a service vary, but among the major reasons are:

1. To increase reliability by having independent backups of each file. If one server goes down, or is even lost permanently, no data are lost. For many applications, this property is extremely desirable.

2. To allow file access to occur even if one file server is down. The motto here is: The show must go on. A server crash should not bring the entire system down until the server can be rebooted.

3. To split the workload over multiple servers. As the system grows in size, having all the files on one server can become a performance bottleneck. By having files replicated on two or more servers, the least heavily loaded one can be used.

The first two relate to improving reliability and availability; the third concerns performance. All are important.

A key issue relating to replication is transparency (as usual). To what extent are the users aware that some files are replicated? Do they play any role in the replication process, or is it handled entirely automatically? At one extreme, the users are fully aware of the replication process and can even control it. At the other, the system does everything behind their backs. In the latter case, we say that the system is **replication transparent**.

Figure 5-12 shows three ways replication can be done. The first way, shown in Fig. 5-12(a), is for the programmer to control the entire process. When a process makes a file, it does so on one specific server. Then it can make additional copies on other servers, if desired. If the directory server permits multiple copies of a file, the network addresses of all copies can then be associated with the file name, as shown at the bottom of Fig. 5-12(a), so that when the name is looked up, all copies will be found. When the file is subsequently opened, the copies can be tried sequentially in some order, until an available one is found.
To make the concept of explicit replication more familiar, consider how it can be done in a system based on remote mounting in UNIX. Suppose that a programmer’s home directory is `/machine1/usr/ast`. After creating a file, for example the file, `/machine1/usr/ast/xyz`, the programmer, process, or library can use the `cp` command (or equivalent) to make copies in `/machine2/usr/ast/xyz` and `/machine3/usr/ast/xyz`. Programs can be written to accept strings like `/usr/ast/xyz` as arguments, and successively try to open the copies until one succeeds. While this scheme can be made to work, it is a lot of trouble. For this reason, a distributed system should do better.

In Fig. 5-12(b) we see an alternative approach, lazy replication. Here, only one copy of each file is created, on some server. Later, the server itself makes replicas on other servers automatically, without the programmer’s knowledge. The system must be smart enough to be able to retrieve any of these copies if need be. When making copies in the background like this, it is important to pay attention to the possibility that the file might change before the copies can be made.

Our final method is to use group communication, as shown in Fig. 5-13(c). In this scheme, all write system calls are simultaneously transmitted to all the servers, so extra copies are made at the same time the original is made. There are two principal differences between lazy replication and using a group. First, with lazy replication, one server is addressed rather than a group. Second, lazy replication happens in the background, when the server has some free time, whereas when group communication is used, all copies are made at the same time.

### Update Protocols

Above we looked at the problem of how replicated files can be created. Now let us see how existing ones can be modified. Just sending an update message to each copy in sequence is not a good idea because if the process doing the update crashes partway through, some copies will be changed and others not. As a result, some future reads may get the old value and others may get
the new value, hardly a desirable situation. We will now look at two well-known algorithms that solve this problem.

The first algorithm is called **primary copy replication**. When it is used, one server is designated as the primary. All the others are secondaries. When a replicated file is to be updated, the change is sent to the primary server, which makes the change locally and then sends commands to the secondaries, ordering them to change, too. Reads can be done from any copy, primary or secondary.

To guard against the situation that the primary crashes before it has had a chance to instruct all the secondaries, the update should be written to stable storage prior to changing the primary copy. In this way, when a server reboots after a crash, a check can be made to see if any updates were in progress at the time of the crash. If so, they can still be carried out. Sooner or later, all the secondaries will be updated.

Although the method is straightforward, it has the disadvantage that if the primary is down, no updates can be performed. To get around this asymmetry, Gifford (1979) proposed a more robust method, known as **voting**. The basic idea is to require clients to request and acquire the permission of multiple servers before either reading or writing a replicated file.

As a simple example of how the algorithm works, suppose that a file is replicated on *N* servers. We could make a rule stating that to update a file, a client must first contact at least half the servers plus 1 (a majority) and get them to agree to do the update. Once they have agreed, the file is changed and a new version number is associated with the new file. The version number is used to identify the version of the file and is the same for all the newly updated files.

To read a replicated file, a client must also contact at least half the servers plus 1 and ask them to send the version numbers associated with the file. If all the version numbers agree, this must be the most recent version because an attempt to update only the remaining servers would fail because there are not enough of them.

For example, if there are five servers and a client determines that three of them have version 8, it is impossible that the other two have version 9. After all, any successful update from version 8 to version 9 requires getting three servers to agree to it, not just two.

Gifford's scheme is actually somewhat more general than this. In it, to read a file of which *N* replicas exist, a client needs to assemble a **read quorum**, an arbitrary collection of any *N*<sub>r</sub> servers, or more. Similarly, to modify a file, a **write quorum** of at least *N*<sub>w</sub> servers is required. The values of *N*<sub>r</sub> and *N*<sub>w</sub> are subject to the constraint that *N*<sub>r</sub>+*N*<sub>w</sub>*N*. Only after the appropriate number of servers has agreed to participate can a file be read or written.

To see how this algorithm works, consider Fig. 5-13(a), which has *N*<sub>r</sub>=3 and *N*<sub>w</sub>=10. Imagine that the most recent write quorum consisted of the 10 servers *C* through *L*. All of these get the new version and the new version number. Any subsequent read quorum of three servers will have to contain at least one member of this set. When the client looks at the version numbers, it will know which is most recent and take that one.
Fig. 5-13. Three examples of the voting algorithm.

In Fig. 5-13(b) and (c), we see two more examples. The latter is especially interesting because it sets \( N_r \) to 1, making it possible to read a replicated file by finding any copy and using it. The price paid, however, is that write updates need to acquire all copies.

An interesting variation on voting is voting with ghosts (Van Renesse and Tanenbaum, 1988). In most applications, reads are much more common than writes, so \( N_r \) is typically a small number and \( N_w \) is nearly \( N \). This choice means that if a few servers are down, it may be impossible to obtain a write quorum.

Voting with ghosts solves this problem by creating a dummy server, with no storage, for each real server that is down. A ghost is not permitted in a read quorum (it does not have any files, after all), but it may join a write quorum, in which case it just throws away the file written to it. A write succeeds only if at least one server is real.

When a failed server is rebooted, it must obtain a read quorum to locate the most recent version, which it then copies to itself before starting normal operation. The algorithm works because it has the same property as the basic voting scheme, namely, \( N_r \) and \( N_w \) are chosen so that acquiring a read quorum and a write quorum at the same time is impossible. The only difference here is that dead machines are allowed in a write quorum, subject to the condition that when they come back up they immediately obtain the current version before going into service.

Other replication algorithms are described in (Bernstein and Goodman, 1984; Brereton, 1986; Pu et al., 1986; and Purdin et al., 1987).

5.2.5. An Example: Sun’s Network File System

In this section we will examine an example network file system, Sun Microsystem’s Network File System, universally known as NFS. NFS was originally designed and implemented by Sun Microsystems for use on its UNIX-based workstations. Other manufacturers now support it as well, for both UNIX and other operating systems (including MS-DOS). NFS supports heterogeneous systems, for example, MS-DOS clients using UNIX servers. It is not even required that all the machines use the same hardware. It is common to find MS-DOS clients running on Intel 386 CPUs getting service from UNIX file servers running on Motorola 68030 or Sun SPARC CPUs.

Three aspects of NFS are of interest: the architecture, the protocol, and the implementation. Let us look at these in turn.

**NFS Architecture**
The basic idea behind NFS is to allow an arbitrary collection of clients and servers to share a common file system. In most cases, all the clients and servers are on the same LAN, but this is not required. It is possible to run NFS over a wide-area network. For simplicity we will speak of clients and servers as though they were on distinct machines, but in fact, NFS allows every machine to be both a client and a server at the same time.

Each NFS server exports one or more of its directories for access by remote clients. When a directory is made available, so are all of its subdirectories, so in fact, entire directory trees are normally exported as a unit. The list of directories a server exports is maintained in the `/etc/exports` file, so these directories can be exported automatically whenever the server is booted.

Clients access exported directories by mounting them. When a client mounts a (remote) directory, it becomes part of its directory hierarchy, as shown in Fig. 5-13. Many Sun workstations are diskless. If it so desires, a diskless client can mount a remote file system on its root directory, resulting in a file system that is supported entirely on a remote server. Those workstations that do have local disks can mount remote directories anywhere they wish on top of their local directory hierarchy, resulting in a file system that is partly local and partly remote. To programs running on the client machine, there is (almost) no difference between a file located on a remote file server and a file located on the local disk.

Thus the basic architectural characteristic of NFS is that servers export directories and clients mount them remotely. If two or more clients mount the same directory at the same time, they can communicate by sharing files in their common directories. A program on one client can create a file, and a program on a different one can read the file. Once the mounts have been done, nothing special has to be done to achieve sharing. The shared files are just there in the directory hierarchy of multiple machines and can be read and written the usual way. This simplicity is one of the great attractions of NFS.

**NFS Protocols**

Since one of the goals of NFS is to support a heterogeneous system, with clients and servers possibly running different operating systems on different hardware, it is essential that the interface between the clients and servers be well defined. Only then is it possible for anyone to be able to write a new client implementation and expect it to work correctly with existing servers, and vice versa.

NFS accomplishes this goal by defining two client-server protocols. A protocol is a set of requests sent by clients to servers, along with the corresponding replies sent by the servers back to the clients. (Protocols are an important topic in distributed systems; we will come back to them later in more detail.) As long as a server recognizes and can handle all the requests in the protocols, it need not know anything at all about its clients. Similarly, clients can treat servers as "black boxes" that accept and process a specific set of requests. How they do it is their own business.

The first NFS protocol handles mounting. A client can send a path name to a server and request permission to mount that directory somewhere in its directory hierarchy. The place where it is to be mounted is not contained in the message, as the server does not care where it is to be mounted. If the path name is legal and the directory specified has been exported, the server returns a file handle to the client. The file handle contains fields uniquely identifying the file system type, the disk, the i-node number of the directory, and security information. Subsequent calls to read and write files in the mounted directory use the file handle.
Many clients are configured to mount certain remote directories without manual intervention. Typically, these clients contain a file called `/etc/rc`, which is a shell script containing the remote mount commands. This shell script is executed automatically when the client is booted.

Alternatively, Sun's version of UNIX also supports **automounting**. This feature allows a set of remote directories to be associated with a local directory. None of these remote directories are mounted (or their servers even contacted) when the client is booted. Instead, the first time a remote file is opened, the operating system sends a message to each of the servers. The first one to reply wins, and its directory is mounted.

Automounting has two principal advantages over static mounting via the `/etc/rc` file. First, if one of the NFS servers named in `/etc/rc` happens to be down, it is impossible to bring the client up, at least not without some difficulty, delay, and quite a few error messages. If the user does not even need that server at the moment, all that work is wasted. Second, by allowing the client to try a set of servers in parallel, a degree of fault tolerance can be achieved (because only one of them need to be up), and the performance can be improved (by choosing the first one to reply — presumably the least heavily loaded).

On the other hand, it is tacitly assumed that all the file systems specified as alternatives for the automount are identical. Since NFS provides no support for file or directory replication, it is up to the user to arrange for all the file systems to be the same. Consequently, automounting is most often used for read-only file systems containing system binaries and other files that rarely change.

The second NFS protocol is for directory and file access. Clients can send messages to servers to manipulate directories and to read and write files. In addition, they can also access file attributes, such as file mode, size, and time of last modification. Most UNIX system calls are supported by NFS, with the perhaps surprising exception of `OPEN` and `CLOSE`.

The omission of `OPEN` and `CLOSE` is not an accident. It is fully intentional. It is not necessary to open a file before reading it, nor to close it when done. Instead, to read a file, a client sends the server a message containing the file name, with a request to look it up and return a file handle, which is a structure that identifies the file. Unlike an `OPEN` call, this `LOOKUP` operation does not copy any information into internal system tables. The `READ` call contains the file handle of the file to read, the offset in the file to begin reading, and the number of bytes desired. Each such message is self-contained. The advantage of this scheme is that the server does not have to remember anything about open connections in between calls to it. Thus if a server crashes and then recovers, no information about open files is lost, because there is none. A server like this that does not maintain state information about open files is said to be **stateless**.

In contrast, in UNIX System V, the **Remote File System (RFS)** requires a file to be opened before it can be read or written. The server then makes a table entry keeping track of the fact that the file is open, and where the reader currently is, so each request need not carry an offset. The disadvantage of this scheme is that if a server crashes and then quickly reboots, all open connections are lost, and client programs fail. NFS does not have this property.

Unfortunately, the NFS method makes it difficult to achieve the exact UNIX file semantics. For example, in UNIX a file can be opened and locked so that other processes cannot access it. When the file is closed, the locks are released. In a stateless server such as NFS, locks cannot be associated with open files, because the server does not know which files are open. NFS therefore needs a separate, additional mechanism to handle locking.

NFS uses the UNIX protection mechanism, with the `rwx` bits for the owner, group, and others. Originally, each request message simply contained the user and group ids of the caller, which the NFS server used to validate the access. In effect, it trusted the clients not to cheat. Several years' experience abundantly demonstrated that such an assumption was — how shall we put it? — naive. Currently, public key cryptography can be used to establish a secure key for validating the client and server on each request and reply. When this option is enabled, a malicious client cannot
impersonate another client because it does not know that client's secret key. As an aside, cryptography is used only to authenticate the parties. The data themselves are never encrypted.

All the keys used for the authentication, as well as other information are maintained by the NIS (Network Information Service). The NIS was formerly known as the yellow pages. Its function is to store (key, value) pairs. When a key is provided, it returns the corresponding value. Not only does it handle encryption keys, but it also stores the mapping of user names to (encrypted) passwords, as well as the mapping of machine names to network addresses, and other items.

The network information servers are replicated using a master/slave arrangement. To read their data, a process can use either the master or any of the copies (slaves). However, all changes must be made only to the master, which then propagates them to the slaves. There is a short interval after an update in which the data base is inconsistent.

**NFS Implementation**

Although the implementation of the client and server code is independent of the NFS protocols, it is interesting to take a quick peek at Sun's implementation. It consists of three layers, as shown in Fig. 5-14. The top layer is the system call layer. This handles calls like OPEN, READ, and CLOSE. After parsing the call and checking the parameters, it invokes the second layer, the virtual file system (VFS) layer.

![Fig. 5-14. NFS layer structure.](image)

The task of the VFS layer is to maintain a table with one entry for each open file, analogous to the table of i-nodes for open files in UNIX. In ordinary UNIX, an i-node is indicated uniquely by a
(device, i-node number) pair. Instead, the VFS layer has an entry, called a v-node (virtual i-node), for every open file. V-nodes are used to tell whether the file is local or remote. For remote files, enough information is provided to be able to access them.

To see how v-nodes are used, let us trace a sequence of MOUNT, OPEN, and READ system calls. To mount a remote file system, the system administrator calls the mount program specifying the remote directory, the local directory on which it is to be mounted, and other information. The mount program parses the name of the remote directory to be mounted and discovers the name of the machine on which the remote directory is located. It then contacts that machine asking for a file handle for the remote directory. If the directory exists and is available for remote mounting, the server returns a file handle for the directory. Finally, it makes a MOUNT system call, passing the handle to the kernel.

The kernel then constructs a v-node for the remote directory and asks the NFS client code in Fig. 5-14 to create an r-node (remote i-node) in its internal tables to hold the file handle. The v-node points to the r-node. Each v-node in the VFS layer will ultimately contain either a pointer to an r-node in the NFS client code, or a pointer to an i-node in the local operating system (see Fig. 5-14). Thus from the v-node it is possible to see if a file or directory is local or remote, and if it is remote, to find its file handle.

When a remote file is opened, at some point during the parsing of the path name, the kernel hits the directory on which the remote file system is mounted. It sees that this directory is remote and in the directory's v-node finds the pointer to the r-node. It then asks the NFS client code to open the file. The NFS client code looks up the remaining portion of the path name on the remote server associated with the mounted directory and gets back a file handle for it. It makes an r-node for the remote file in its tables and reports back to the VFS layer, which puts in its tables a v-node for the file that points to the r-node. Again here we see that every open file or directory has a v-node that points to either an r-node or an i-node.

The caller is given a file descriptor for the remote file. This file descriptor is mapped onto the v-node by tables in the VFS layer. Note that no table entries are made on the server side. Although the server is prepared to provide file handles upon request, it does not keep track of which files happen to have file handles outstanding and which do not. When a file handle is sent to it for file access, it checks the handle, and if it is valid, uses it. Validation can include verifying an authentication key contained in the RPC headers, if security is enabled.

When the file descriptor is used in a subsequent system call, for example, read, the VFS layer locates the corresponding v-node, and from that determines whether it is local or remote and also which i-node or r-node describes it.

For efficiency reasons, transfers between client and server are done in large chunks, normally 8192 bytes, even if fewer bytes are requested. After the client's VFS layer has gotten the 8K chunk it needs, it automatically issues a request for the next chunk, so it will have it should it be needed shortly. This feature, known as read ahead, improves performance considerably.

For writes an analogous policy is followed. If a WRITE system call supplies fewer than 8192 bytes of data, the data are just accumulated locally. Only when the entire 8K chunk is full is it sent to the server. However, when a file is closed, all of its data are sent to the server immediately.

Another technique used to improve performance is caching, as in ordinary UNIX. Servers cache data to avoid disk accesses, but this is invisible to the clients. Clients maintain two caches, one for file attributes (i-nodes) and one for file data. When either an i-node or a file block is needed, a check is made to see if it can be satisfied out of the cache. If so, network traffic can be avoided.

While client caching helps performance enormously, it also introduces some nasty problems. Suppose that two clients are both caching the same file block and that one of them modifies it. When the other one reads the block, it gets the old (stale) value. The cache is not coherent. We
saw the same problem with multiprocessors earlier. However, there it was solved by having the caches snoop on the bus to detect all writes and invalidate or update cache entries accordingly. With a file cache that is not possible, because a write to a file that results in a cache hit on one client does not generate any network traffic. Even if it did, snooping on the network is nearly impossible with current hardware.

Given the potential severity of this problem, the NFS implementation does several things to mitigate it. For one, associated with each cache block is a timer. When the timer expires, the entry is discarded. Normally, the timer is 3 sec for data blocks and 30 sec for directory blocks. Doing this reduces the risk somewhat. In addition, whenever a cached file is opened, a message is sent to the server to find out when the file was last modified. If the last modification occurred after the local copy was cached, the cache copy is discarded and the new copy fetched from the server. Finally, once every 30 sec a cache timer expires, and all the dirty (i.e., modified) blocks in the cache are sent to the server.

Still, NFS has been widely criticized for not implementing the proper UNIX semantics. A write to a file on one client may or may not be seen when another client reads the file, depending on the timing. Furthermore, when a file is created, it may not be visible to the outside world for as much as 30 sec. Similar problems exist as well.

From this example we see that although NFS provides a shared file system, because the resulting system is kind of a patched-up UNIX, the semantics of file access are not entirely well defined, and running a set of cooperating programs again may give different results, depending on the timing. Furthermore, the only issue NFS deals with is the file system. Other issues, such as process execution, are not addressed at all. Nevertheless, NFS is popular and widely used.

**5.2.6. Lessons Learned**

Based on his experience with various distributed file systems, Satyanarayanan (1990b) has stated some general principles that he believes distributed file system designers should follow. We have summarized these in Fig. 5-15. The first principle says that workstations have enough CPU power that it is wise to use them wherever possible. In particular, given a choice of doing something on a workstation or on a server, choose the workstation because server cycles are precious and workstation cycles are not.

The second principle says to use caches. They can frequently save a large amount of computing time and network bandwidth.

<table>
<thead>
<tr>
<th>Workstations have cycles to burn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache whenever possible</td>
</tr>
<tr>
<td>Exploit the usage properties</td>
</tr>
<tr>
<td>Minimize systemwide knowledge and change</td>
</tr>
<tr>
<td>Trust the fewest possible entities</td>
</tr>
<tr>
<td>Batch work where possible</td>
</tr>
</tbody>
</table>

*Fig. 5-15. Distributed file system design principles.*
The third principle says to exploit usage properties. For example, in a typical UNIX system, about a third of all file references are to temporary files, which have short lifetimes and are never shared. By treating these specially, considerable performance gains are possible. In all fairness, there is another school of thought that says: "Pick a single mechanism and stick to it. Do not have five ways of doing the same thing." Which view one takes depends on whether one prefers efficiency or simplicity.

Minimizing systemwide knowledge and change is important for making the system scale. Hierarchical designs help in this respect.

Trusting the fewest possible entities is a long-established principle in the security world. If the correct functioning of the system depends on 10,000 workstations all doing what they are supposed to, the system has a big problem.

Finally, batching can lead to major performance gains. Transmitting a 50K file in one blast is much more efficient than sending it as 50 1K blocks.

5.3. TRENDS IN DISTRIBUTED FILE SYSTEMS

Although rapid change has been a part of the computer industry since its inception, new developments seem to be coming faster than ever in recent years, both in the hardware and software areas. Many of these hardware changes are likely to have major impact on the distributed file systems of the future. In addition to all the improvements in the technology, changing user expectations and applications are also likely to have a major impact. In this section, we will survey some of the changes that can be expected in the foreseeable future and discuss some of the implications these changes may have for file systems. This section will raise more questions than it will answer, but it will suggest some interesting directions for future research.

5.3.1. New Hardware

Before looking at new hardware, let us look at old hardware with new prices. As memory continues to get cheaper and cheaper, we may see a revolution in the way file servers are organized. Currently, all file servers use magnetic disks for storage. Main memory is often used for server caching, but this is merely an optimization for better performance. It is not essential.

Within a few years, memory may become so cheap that even small organizations can afford to equip all their file servers with gigabytes of physical memory. As a consequence, the file system may permanently reside in memory, and no disks will be needed. Such a step will give a large gain in performance and will greatly simplify file system structure.

Most current file systems organize files as a collection of blocks, either as a tree (e.g., UNIX) or as a linked list (e.g., MS-DOS). With an in-core file system, it may be much simpler to store each file contiguously in memory, rather than breaking it up into blocks. Contiguously stored files are easier to keep track of and can be shipped over the network faster. The reason that contiguous files are not used on disk is that if a file grows, moving it to an area of the disk with more room is too expensive. In contrast, moving a file to another area of memory is feasible.

Main memory file servers introduce a serious problem, however. If the power fails, all the files are lost. Unlike disks, which do not lose information in a power failure, main memory is erased when the electricity is removed. The solution may be to make continuous or at least incremental backups onto videotape. With current technology, it is possible to store about 5 gigabytes on a single 8mm videotape that costs less than 10 dollars. While access time is long, if access is needed only once or twice a year to recover from power failures, this scheme may prove irresistible.

A hardware development that may affect file systems is the optical disk. Originally, these devices had the property that they could be written once (by burning holes in the surface with a
laser), but not changed thereafter. They were sometimes referred to as **WORM (Write Once Read Many)** devices. Some current optical disks use lasers to affect the crystal structure of the disk, but do not damage them, so they can be erased.

Optical disks have three important properties:

1. They are slow.
2. They have huge storage capacities.
3. They have random access.

They are also relatively cheap, although more expensive than videotape. The first two properties are the same as videotape, but the third opens the following possibility. Imagine a file server with an \( n \)-gigabyte file system in main memory, and an \( n \)-gigabyte optical disk as backup. When a file is created, it is stored in main memory and marked as not yet backed up. All accesses are done using main memory. When the workload is low, files that are not yet backed up are transferred to the optical disk in the background, with byte \( k \) in memory going to byte \( k \) on the disk. Like the first scheme, what we have here is a main memory file server, but with a more convenient backup device having a one-to-one mapping with the memory.

Another interesting hardware development is very fast fiber optic networks. As we discussed earlier, the reason for doing client caching, with all its inherent complications, is to avoid the slow transfer from the server to the client. But suppose that we could equip the system with a main memory file server and a fast fiber optic network. It might well become feasible to get rid of the client's cache and the server's disk and just operate out of the server's memory, backed up by optical disk. This would certainly simplify the software.

When studying client caching, we saw that a large fraction of the trouble is caused by the fact that if two clients are caching the same file and one of them modifies it, the other does not discover this, which leads to inconsistencies. A little thought will reveal that this situation is highly analogous to memory caches in a multiprocessor. Only there, when one processor modifies a shared word, a hardware signal is sent over the memory bus to the other caches to allow them to invalidate or update that word. With distributed file systems, this is not done.

Why not, actually? The reason is that current network interfaces do not support such signals. Nevertheless, it should be possible to build network interfaces that do. As a very simple example, consider the system of Fig. 5-16 in which each network interface has a bit map, one bit per cached file. To modify a file, a processor sets the corresponding bit in the interface, which is 0 if no processor is currently updating the file. Setting a bit causes the interface to create and send a packet around the ring that checks and sets the corresponding bit in all interfaces. If the packet makes it all the way around without finding any other machines trying to use the file, some other register in the interface is set to 1. Otherwise, it is set to 0. In effect, this mechanism provides a way to globally lock the file on all machines in a few microseconds.

After the lock has been set, the processor updates the file. Each block of the file that is changed is noted (e.g., using bits in the page table). When the update is complete, the processor clears the bit in the bit map, which causes the network interface to locate the file using a table in memory and automatically deposit all the modified blocks in their proper locations on the other machines. When the file has been updated everywhere, the bit in the bit map is cleared on all machines.
Clearly, this is a simple solution that can be improved in many ways, but it shows how a small amount of well-designed hardware can solve problems that are difficult to handle in software. It is likely that future distributed systems will be assisted by specialized hardware of various kinds.

5.3.2. Scalability

A definite trend in distributed systems is toward larger and larger systems. This observation has implications for distributed file system design. Algorithms that work well for systems with 100 machines may work poorly for systems with 1000 machines and not at all for systems with 10,000 machines. For starters, centralized algorithms do not scale well. If opening a file requires contacting a single centralized server to record the fact that the file is open, that server will eventually become a bottleneck as the system grows.

A general way to deal with this problem is to partition the system into smaller units and try to make each one relatively independent of the others. Having one server per unit scales much better than a single server. Even having the servers record all the opens may be acceptable under these circumstances.

Broadcasts are another problem area. If each machine issues one broadcast per second, with \( n \) machines, a total of \( n \) broadcasts per second appear on the network, generating a total of \( n^2 \) interrupts total. Obviously, as \( n \) grows, this will eventually be a problem. Resources and algorithms should not be linear in the number of users, so having a server maintain a linear list of users for protection or other purposes is not a good idea. In contrast, hash tables are acceptable, since the access time is more or less constant, almost independent of the number of entries.

In general, strict semantics, such as UNIX semantics, get harder to implement as systems get bigger. Weaker guarantees are much easier to implement. Clearly, there is a trade-off here, since programmers prefer easily well-defined semantics, but these are precisely the ones that do not scale well.

In a very large system, the concept of a single UNIX-like file tree may have to be reexamined. It is inevitable that as the system grows, the length of path names will grow too, adding more overhead. At some point it may be necessary to partition the tree into smaller trees.

5.3.3. Wide Area Networking

Most current work on distributed systems focuses on LAN-based systems. In the future, many LAN-based distributed systems will be interconnected to form transparent distributed systems covering countries and continents. As an example, the French PTT is currently putting a small computer in every apartment and house in France. Although the initial goal is to eliminate the need for information operators and telephone books, at some point in time someone is going to ask if it is possible to connect 10 million or more computers spread over all of France into a single
transparent system, for applications as yet undreamed of. What kind of file system would be needed to serve all of France? All of Europe? The entire world? At present, no one knows.

Although the French machines are all identical, in most wide-area networks, a large variety of equipment is encountered. This diversity is inevitable when multiple buyers with different-sized budgets and goals are involved, and the purchasing is spread over many years in an era of rapid technological change. Thus a wide-area distributed system must of necessity deal with heterogeneity. This raises issues such as how should you store a character file if not everyone uses ASCII, or what format one should use for files containing floating-point numbers if multiple representations are in use.

Also important is the expected change in applications. Most experimental distributed systems being built at universities focus on programming in a UNIX-like environment as the canonical application, because that is what the researchers themselves do all day (at least when they are not in committee meetings or writing grant proposals). Initial data suggest that not all 50 million French citizens are going to list C programming as their primary activity. As distributed systems become more widespread, we are likely to see a shift to electronic mail, electronic banking, accessing data bases, and recreational activities, which will change file usage, access patterns, and a great deal more in ways we as yet do not know.

An inherent problem with massive distributed systems is that the network bandwidth is extremely low. If the telephone line is the main connection, getting more than 64 Kbps out of it seems unlikely. Bringing fiber optics into everyone's house will take decades and cost billions. On the other hand, vast amounts of data can be stored cheaply on compact disks and videotapes. Instead of logging into the telephone company's computer to look up a telephone number, it may be cheaper for them to send everyone a disk or tape containing the entire data base. We may have to develop file systems in which a distinction is made between static, read-only information (e.g., the phone book), and dynamic information (e.g., electronic mail). This distinction may have to become the basis of the entire file system.

5.3.4. Mobile Users

Portable computers are the fastest-growing segment of the computer business. Laptop computers, notebook computers, and pocket computers can be found everywhere these days, and they are multiplying like rabbits. Although computing while driving is hard, computing while flying is not. Telephones are now common in airplanes, so can flying FAXes and mobile modems be far behind? Nevertheless, the total bandwidth available from an airplane to the ground is quite low, and many places users want to go have no online connection at all.

The inevitable conclusion is that a large fraction of the time, the user will be off-line, disconnected from the file system. Few current systems were designed for such use, although Satyanarayanan (1990b) has reported some initial work in this direction.

Any solution is probably going to have to be based on caching. While connected, the user downloads to the portable those files expected to be needed later. These are used while disconnected. When reconnect occurs, the files in the cache will have to be merged with those in the file tree. Since disconnect can last for hours or days, the problems of maintaining cache consistency are much more severe than in online systems.

Another problem is that when reconnect does occur, the user may be in a city far away from his home base. Placing a phone call to the home machine is one way to get resynchronized, but the telephone bandwidth is low. Besides, in a truly distributed system contacting the local file server should be enough. The design of a worldwide, fully transparent distributed file system for simultaneous use by millions of mobile and frequently disconnected users is left as an exercise for the reader.

5.3.5. Fault Tolerance
Current computer systems, except for very specialized ones like air traffic control, are not fault tolerant. When the computer goes down, the users are expected to accept this as a fact of life. Unfortunately, the general population expects things to work. If a television channel, the phone system, or the electric power company goes down for half an hour, there are many unhappy people the next day. As distributed systems become more and more widespread, the demand for systems that essentially never fail will grow. Current systems cannot meet this need.

Obviously, such systems will need considerable redundancy in hardware and the communication infrastructure, but they will also need it in software and especially data. File replication, often an afterthought in current distributed systems, will become an essential requirement in future ones. Systems will also have to be designed that manage to function when only partial data are available, since insisting that all the data be available all the time does not lead to fault tolerance. Down times that are now considered acceptable by programmers and other sophisticated users, will be increasingly unacceptable as computer use spreads to nonspecialists.

**Distributed Shared Memory**

There are two kinds of multiple-processor systems exist: multiprocessors and multicomputers. In a multiprocessor, two or more CPUs share a common main memory. Any process, on any processor, can read or write any word in the shared memory simply by moving data to or from the desired location. In a multicomputer, in contrast, each CPU has its own private memory. Nothing is shared.

To make an agricultural analogy, a multiprocessor is a system with a herd of pigs (processes) eating from a single feeding trough (shared memory). A multicomputer is a design in which each pig has its own private feeding trough. To make an educational analogy, a multiprocessor is a blackboard in the front of the room which all the students are looking at, whereas a multicomputer is each student looking at his or her own notebook. Although this difference may seem minor, it has far-reaching consequences.

The consequences affect both hardware and software. Let us first look at the implications for the hardware. Designing a machine in which many processors use the same memory simultaneously is surprisingly difficult. Bus-based multiprocessors, as described in Sec. 1.3.1, cannot be used with more than a few dozen processors because the bus tends to become a bottleneck. Switched multiprocessors, as described in Sec. 1.3.2, can be made to scale to large systems, but they are relatively expensive, slow, complex, and difficult to maintain.

In contrast, large multicomputers are easier to build. One can take an almost unlimited number of single-board computers, each containing a CPU, memory, and a network interface, and connect them together. Multicomputers with thousands of processors are commercially available from various manufacturers. (Please note that throughout this chapter we use the terms "CPU" and "processor" interchangeably.) From a hardware designer's perspective, multicomputers are generally preferable to multiprocessors.

Now let us consider the software. Many techniques are known for programming multiprocessors. For communication, one process just writes data to memory, to be read by all the others. For synchronization, critical regions can be used, with semaphores or monitors providing the necessary mutual exclusion. There is an enormous body of literature available on interprocess communication and synchronization on shared-memory machines. Every operating systems textbook written in the past twenty years devotes one or more chapters to the subject. In short, a large amount of theoretical and practical knowledge exists about how to program a multiprocessor.

With multicomputers, the reverse is true. Communication generally has to use message passing, making input/output the central abstraction. Message passing brings with it many complicating issues, among them flow control, lost messages, buffering, and blocking. Although various solutions have been proposed, programming with message passing remains tricky.
To hide some of the difficulties associated with message passing, Birrell and Nelson (1984) proposed using remote procedure calls. In their scheme, now widely used, the actual communication is hidden away in library procedures. To use a remote service, a process just calls the appropriate library procedure, which packs the operation code and parameters into a message, sends it over the network, and waits for the reply. While this frequently works, it cannot easily be used to pass graphs and other complex data structures containing pointers. It also fails for programs that use global variables, and it makes passing large arrays expensive, since they must be passed by value rather than by reference.

In short, from a software designer’s perspective, multiprocessors are definitely preferable to multicomputers. Herein lies the dilemma. Multicomputers are easier to build but harder to program. Multiprocessors are the opposite: harder to build but easier to program. What we need are systems that are both easy to build and easy to program. Attempts to build such systems form the subject of this chapter.

6.1. INTRODUCTION

In the early days of distributed computing, everyone implicitly assumed that programs on machines with no physically shared memory (i.e., multicomputers) obviously ran in different address spaces. Given this mindset, communication was naturally viewed in terms of message passing between disjoint address spaces, as described above. In 1986, Li proposed a different scheme, now known under the name distributed shared memory (DSM) (Li, 1986; and Li and Hudak, 1989). Briefly summarized, Li and Hudak proposed having a collection of workstations connected by a LAN share a single paged, virtual address space. In the simplest variant, each page is present on exactly one machine. A reference to a local page is done in hardware, at full memory speed. An attempt to reference a page on a different machine causes a hardware page fault, which traps to the operating system. The operating system then sends a message to the remote machine, which finds the needed page and sends it to the requesting processor. The faulting instruction is then restarted and can now complete.

In essence, this design is similar to traditional virtual memory systems: when a process touches a nonresident page, a trap occurs and the operating system fetches the page and maps it in. The difference here is that instead of getting the page from the disk, the operating system gets it from another processor over the network. To the user processes, however, the system looks very much like a traditional multiprocessor, with multiple processes free to read and write the shared memory at will. All communication and synchronization can be done via the memory, with no communication visible to the user processes. In effect, Li and Hudak devised a system that is both easy to program (logically shared memory) and easy to build (no physically shared memory).

Unfortunately, there is no such thing as a free lunch. While this system is indeed easy to program and easy to build, for many applications it exhibits poor performance, as pages are hurled back and forth across the network. This behavior is analogous to thrashing in single-processor virtual memory systems. In recent years, making these distributed shared memory systems more efficient has been an area of intense research, with numerous new techniques discovered. All of these have the goal of minimizing the network traffic and reducing the latency between the moment a memory request is made and the moment it is satisfied.

One approach is not to share the entire address space, only a selected portion of it, namely just those variables or data structures that need to be used by more than one process. In this model, one does not think of each machine as having direct access to an ordinary memory but rather, to a collection of shared variables, giving a higher level of abstraction. Not only does this strategy greatly reduce the amount of data that must be shared, but in most cases, considerable information about the shared data is available, such as their types, which can help optimize the implementation.
One possible optimization is to replicate the shared variables on multiple machines. By sharing replicated variables instead of entire pages, the problem of simulating a multiprocessor has been reduced to that of how to keep multiple copies of a set of typed data structures consistent. Potentially, reads can be done locally without any network traffic, and writes can be done using a multicopy update protocol. Such protocols are widely used in distributed data base systems, so ideas from that field may be of use.

Going still further in the direction of structuring the address space, instead of just sharing variables we could share encapsulated data types, often called objects. These differ from shared variables in that each object has not only some data, but also procedures, called methods, that act on the data. programs may only manipulate an object’s data by invoking its methods. Direct access to the data is not permitted. By restricting access in this way, various new optimizations become possible.

Doing everything in software has a different set of advantages and disadvantages from using the paging hardware. In general, it tends to put more restrictions on the programmer but may achieve better performance. Many of these restrictions (e.g., working with objects) are considered good software engineering practice and are desirable in their own right. We will come back to this subject later.

Before getting into distributed shared memory in more detail, we must first take a few steps backward to see what shared memory really is and how shared-memory multiprocessors actually work. After that we will examine the semantics of sharing, since they are surprisingly subtle. Finally, we will come back to the design of distributed shared memory systems. Because distributed shared memory can be intimately related to computer architecture, operating systems, runtime systems, and even programming languages, all of these topics will come into play in this chapter.

6.2. WHAT IS SHARED MEMORY?

In this section we will examine several kinds of shared memory multiprocessors, ranging from simple ones that operate over a single bus, to advanced ones with highly sophisticated caching schemes. These machines are important for an understanding of distributed shared memory because much of the DSM work is being inspired by advances in multiprocessor architecture. Furthermore, many of the algorithms are so similar that it is sometimes difficult to tell whether an advanced machine is a multiprocessor or a multicomputer using a hardware implementation of distributed shared memory. We will conclude by comparing the various multiprocessor architectures to some distributed shared memory systems and discover that there is a spectrum of possible designs, from those entirely in hardware to those entirely in software. By examining the entire spectrum, we can get a better feel for where DSM fits in.

6.2.1. On-Chip Memory

Although most computers have an external memory, self-contained chips containing a CPU and all the memory also exist. Such chips are produced by the millions, and are widely used in cars, appliances, and even toys. In this design, the CPU portion of the chip has address and data lines that directly connect to the memory portion. Figure 6-1(a) shows a simplified diagram of such a chip.
Fig. 6-1. (a) A single-chip computer. (b) A hypothetical shared-memory multiprocessor.

One could imagine a simple extension of this chip to have multiple CPUs directly sharing the same memory, as shown in Fig. 6-1(b). While it is possible to construct a chip like this, it would be complicated, expensive, and highly unusual. An attempt to construct a one-chip multiprocessor this way, with, say, 100 CPUs directly accessing the same memory would be impossible for engineering reasons. A different approach to sharing memory is needed.

**6.2.2. Bus-Based Multiprocessors**

If we look closely at Fig. 6-1(a), we see that the connection between the CPU and the memory is a collection of parallel wires, some holding the address the CPU wants to read or write, some for sending or receiving data, and the rest for controlling the transfers. Such a collection of wires is called a **bus**. This bus is on-chip, but in most systems, buses are external and are used to connect printed circuit boards containing CPUs, memories, and I/O controllers. On a desktop computer, the bus is typically etched onto the main board (the parent-board), which holds the CPU and some of the memory, and into which I/O cards are plugged. On minicomputers the bus is sometimes a flat cable that wends its way among the processors, memories, and I/O controllers.

A simple but practical way to build a multiprocessor is to base it on a bus to which more than one CPU is connected. Fig. 6-2(a) illustrates a system with three CPUs and a memory shared among all of them. When any of the CPUs wants to read a word from the memory, it puts the address of the word it wants on the bus and asserts (puts a signal on) a bus control line indicating that it wants to do a read. When the memory has fetched the requested word, it puts the word on the bus and asserts another control line to announce that it is ready. The CPU then reads in the word. Writes work in an analogous way.
To prevent two or more CPUs from trying to access the memory at the same time, some kind of bus arbitration is needed. Various schemes are in use. For example, to acquire the bus, a CPU might first have to request it by asserting a special request line. Only after receiving permission would it be allowed to use the bus. The granting of this permission can be done in a centralized way, using a bus arbitration device, or in a decentralized way, with the first requesting CPU along the bus winning any conflict.

The disadvantage of having a single bus is that with as few as three or four CPUs the bus is likely to become overloaded. The usual approach taken to reduce the bus load is to equip each CPU with a snooping cache (sometimes called a snoopy cache), so called because it "snoops" on the bus. Caches are shown in Fig. 6-2(b). They have been the subject of a large amount of research over the years (Agarwal et al., 1988; Agarwal and Cherian, 1989; Archibald and Baer, 1986; Cheong and Veenenbaum, 1988; Dahlgren et al., 1994; Eggers and Katz, 1989a, 1989b; Nayfeh and Olukotun, 1994; Przybylski et al., 1988; Scheurich and Dubois, 1987; Thekkath and Eggers, 1994; Vernon et al., 1988; and Weber and Gupta, 1989). All of these papers present slightly different cache consistency protocols, that is, rules for making sure that different caches do not contain different values for the same memory location.

One particularly simple and common protocol is called write through. When a CPU first reads a word from memory, that word is fetched over the bus and is stored in the cache of the CPU making the request. If that word is needed again later, the CPU can take it from the cache without making a memory request, thus reducing bus traffic. These two cases, read miss (word not cached) and read hit (word cached) are shown in Fig. 6-3 as the first two lines in the table. In simple systems, only the word requested is cached, but in most, a block of words of say, 16 or 32 words, is transferred and cached on the initial access and kept for possible future use.

<table>
<thead>
<tr>
<th>Event</th>
<th>Action taken by a cache in response to its own CPU's operation</th>
<th>Action taken by a cache in response to a remote CPU's operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read miss</td>
<td>Fetch data from memory and store in cache</td>
<td>(No action)</td>
</tr>
<tr>
<td>Read hit</td>
<td>Fetch data from local cache</td>
<td>(No action)</td>
</tr>
<tr>
<td>Write miss</td>
<td>Update data in memory and store in cache</td>
<td>(No action)</td>
</tr>
</tbody>
</table>
Fig. 6-3. The *write-through* cache consistency protocol. The entries for *hit* in the third column mean that the snooping CPU has the word in its cache, not that the requesting CPU has it.

Each CPU does its caching independent of the others. Consequently, it is possible for a particular word to be cached at two or more CPUs at the same time. Now let us consider what happens when a write is done. If no CPU has the word being written in its cache, the memory is just updated, as if caching were not being used. This operation requires a normal bus cycle. If the CPU doing the write has the only copy of the word, its cache is updated and memory is updated over the bus as well.

So far, so good. The trouble arises when a CPU wants to write a word that two or more CPUs have in their caches. If the word is currently in the cache of the CPU doing the write, the cache entry is updated. Whether it is or not, it is also written to the bus to update memory. All the other caches see the write (because they are snooping on the bus) and check to see if they are also holding the word being modified. If so, they invalidate their cache entries, so that after the write completes, memory is up-to-date and only one machine has the word in its cache.

An alternative to invalidating other cache entries is to update all of them. Updating is slower than invalidating in most cases, however. Invalidating requires supplying just the address to be invalidated, whereas updating needs to provide the new cache entry as well. If these two items must be presented on the bus consecutively, extra cycles will be required. Even if it is possible to put an address and a data word on the bus simultaneously, if the cache block size is more than one word, multiple bus cycles will be needed to update the entire block. The issue of invalidate vs. update occurs in all cache protocols and also in DSM systems.

The complete protocol is summarized in Fig. 6-3. The first column lists the four basic events that can happen. The second one tells what a cache does in response to its own CPU's actions. The third one tells what happens when a cache sees (by snooping) that a different CPU has had a hit or miss. The only time cache S (the snooper) must do something is when it sees that another CPU has written a word that S has cached (a write hit from S's point of view). The action is for S to delete the word from its cache.

The *write-through* protocol is simple to understand and implement but has the serious disadvantage that all writes use the bus. While the protocol certainly reduces bus traffic to some extent, the number of CPUs that can be attached to a single bus is still too small to permit large-scale multiprocessors to be built using it.

Fortunately, for many actual programs, once a CPU has written a word, that CPU is likely to need the word again, and it is unlikely that another CPU will use the word quickly. This situation suggests that if the CPU using the word could somehow be given temporary “ownership” of the word, it could avoid having to update memory on subsequent writes until a different CPU exhibited interest in the word. Such cache protocols exist. Goodman (1983) devised the first one, called *write once*. However, this protocol was designed to work with an existing bus and was therefore more complicated than is strictly necessary. Below we will describe a simplified version of it, which is typical of all ownership protocols. Other protocols are described and compared by Archibald and Baer (1986).

Our protocol manages cache blocks, each of which can be in one of the following three states:

1. INVALID — This cache block does not contain valid data.
2. CLEAN — Memory is up-to-date; the block may be in other caches.
3. DIRTY — Memory is incorrect; no other cache holds the block.
The basic idea is that a word that is being read by multiple CPUs is allowed to be present in all their caches. A word that is being heavily written by only one machine is kept in its cache and not written back to memory on every write to reduce bus traffic.

The operation of the protocol can best be illustrated by an example. For simplicity in this example, we will assume that each cache block consists of a single word. Initially, B has a cached copy of the word at address \( W \) as illustrated in Fig. 6-4(a). The value is \( W_1 \). The memory also has a valid copy. In Fig. 6-4(b), A requests and gets a copy of \( W \) from the memory. Although B sees the read request go by, it does not respond to it.

![Diagram](image)

**Fig. 6-4.** An example of how a cache ownership protocol works.

Now A writes a new value, \( W_2 \) to \( W \). B sees the write request and responds by invalidating its cache entry. A’s state is changed to DIRTY, as shown in Fig. 6-4(c). The DIRTY state means that A has the only cached copy of \( W \) and that memory is out-of-date for \( W \).
At this point, $A$ overwrites the word again, as shown in Fig. 6-4(d). The write is done locally, in the cache, with no bus traffic. All subsequent writes also avoid updating memory.

Sooner or later, some other CPU, $C$ in Fig. 6-4(e), accesses the word. $A$ sees the request on the bus and asserts a signal that inhibits memory from responding. Instead, $A$ provides the needed word and invalidates its own entry. $C$ sees that the word is coming from another cache, not from memory, and that it is in DIRTY state, so it marks the entry accordingly. $C$ is now the owner, which means that it can now read and write the word without making bus requests. However, it also has the responsibility of watching out for other CPUs that request the word, and servicing them itself. The word remains in DIRTY state until it is purged from the cache it is currently residing in for lack of space. At that time it disappears from all caches and is written back to memory.

Many small multiprocessors use a cache consistency protocol similar to this one, often with small variations. It has three important properties:

1. Consistency is achieved by having all the caches do bus snooping.
2. The protocol is built into the memory management unit.
3. The entire algorithm is performed in well under a memory cycle.

As we will see later, some of these do not hold for larger (switched) multiprocessors, and none of them hold for distributed shared memory.

### 6.2.3. Ring-Based Multiprocessors

The next step along the path toward distributed shared memory systems are ring-based multiprocessors, exemplified by **Memnet** (Delp, 1988; Delp et al., 1991; and Tarn et al., 1990). In Memnet, a single address space is divided into a private part and a shared part. The private part is divided up into regions so that each machine has a piece for its stacks and other unshared data and code. The shared part is common to all machines (and distributed over them) and is kept consistent by a hardware protocol roughly similar to those used on bus-based multiprocessors. Shared memory is divided into 32-byte blocks, which is the unit in which transfers between machines take place.

All the machines in Memnet are connected together in a modified token-passing ring. The ring consists of 20 parallel wires, which together allow 16 data bits and 4 control bits to be sent every 100 nsec, for a data rate of 160 Mbps. The ring is illustrated in Fig. 6-5(a). The ring interface, MMU (Memory Management Unit), cache, and part of the memory are integrated together in the **Memnet device**, which is shown in the top third of Fig. 6-5(b).
Unlike the bus-based multiprocessors of Fig. 6-2, in Memnet there is no centralized global memory. Instead, each 32-byte block in the shared address space has a home machine on which physical memory is always reserved for it, in the Home memory field of Fig. 6-5(b). A block may be cached on a machine other than its home machine. (The cache and home memory areas share the same buffer pool, but since they are used slightly differently, we treat them here as separate entities.) A read-only block may be present on multiple machines; a read-write block may be present on only one machine. In both cases, a block need not be present on its home machine. All the home machine does is provide a guaranteed place to store the block if no other machine wants to cache it. This feature is needed because there is no global memory. In effect, the global memory has been spread out over all the machines.

The Memnet device on each machine contains a table, shown in Fig. 6-5(c), which contains an entry for each block in the shared address space, indexed by block number. Each entry contains a Valid bit telling whether the block is present in the cache and up to date, an Exclusive bit, specifying whether the local copy, if any, is the only one, a Home bit, which is set only if this is the block's home machine, an Interrupt bit, used for forcing interrupts, and a Location field that tells where the block is located in the cache if it is present and valid.

Having looked at the architecture of Memnet, let us now examine the protocols it uses. When the CPU wants to read a word from shared memory, the memory address to be read is passed to the Memnet device, which checks the block table to see if the block is present. If so, the request is...
satisfied immediately. If not, the Memnet device waits until it captures the circulating token, then puts a request packet onto the ring and suspends the CPU. The request packet contains the desired address and a 32-byte dummy field.

As the packet passes around the ring, each Memnet device along the way checks to see if it has the block needed. If so, it puts the block in the dummy field and modifies the packet header to inhibit subsequent machines from doing so. If the block’s Exclusive bit is set, it is cleared. Because the block has to be somewhere, when the packet comes back to the sender, it is guaranteed to contain the block requested. The CPU sending the request then stores the block, satisfies the request, and releases the CPU.

A problem arises if the requesting machine has no free space in its cache to hold the incoming block. To make space, it picks a cached block at random and sends it home, thus freeing up a cache slot. Blocks whose Home bit are set are never chosen since they are already home.

Writes work slightly differently than reads. Three cases have to be distinguished. If the block containing the word to be written is present and is the only copy in the system (i.e., the Exclusive bit is set), the word is just written locally.

If the needed block is present but it is not the only copy, an invalidation packet is first sent around the ring to force all other machines to discard their copies of the block about to be written. When the invalidation packet arrives back at the sender, the Exclusive bit is set for that block and the write proceeds locally.

If the block is not present, a packet is sent out that combines a read request and an invalidation request. The first machine that has the block copies it into the packet and discards its own copy. All subsequent machines just discard the block from their caches. When the packet comes back to the sender, it is stored there and written.

Memnet is similar to a bus-based multiprocessor in most ways. In both cases, read operations always return the value most recently written. Also, in both designs, a block may be absent from a cache, present in multiple caches for reading, or present in a single cache for writing. The protocols are similar, too; however, Memnet has no centralized global memory.

The biggest difference between bus-based multiprocessors and ring-based multiprocessors such as Memnet is that the former are tightly coupled, with the CPUs normally being in a single rack. In contrast, the machines in a ring-based multiprocessor can be much more loosely coupled, potentially even on desktops spread around a building, like machines on a LAN, although this loose coupling can adversely effect performance. Furthermore, unlike a bus-based multiprocessor, a ring-based multiprocessor like Memnet has no separate global memory. The caches are all there is. In both respects, ring-based multiprocessors are almost a hardware implementation of distributed shared memory.

One is tempted to say that a ring-based multiprocessor is like a duck-billed platypus — theoretically it ought not exist because it combines the properties of two categories said to be mutually exclusive (multiprocessors and distributed shared memory machines; mammals and birds, respectively). Nevertheless, it does exist, and shows that the two categories are not quite so distinct as one might think.

6.2.4. Switched Multiprocessors

Although bus-based multiprocessors and ring-based multiprocessors work fine for small systems (up to around 64 CPUs), they do not scale well to systems with hundreds or thousands of CPUs. As CPUs are added, at some point the bus or ring bandwidth saturates. Adding additional CPUs does not improve the system performance.

Two approaches can be taken to attack the problem of not enough bandwidth:
1. Reduce the amount of communication.
2. Increase the communication capacity.
We have already seen an example of an attempt to reduce the amount of communication by using caching. Additional work in this area might center on improving the caching protocol, optimizing the block size, reorganizing the program to increase locality of memory references, and so on.

Nevertheless, eventually there comes a time when every trick in the book has been used, but the insatiable designers still want to add more CPUs and there is no bus bandwidth left. The only way out is to add more bus bandwidth. One approach is to change the topology, going, for example, from one bus to two buses or to a tree or grid. By changing the topology of the interconnection network, it is possible to add additional communication capacity.

A different method is to build the system as a hierarchy. Continue to put some number of CPUs on a single bus, but now regard this entire unit (CPUs plus bus) as a cluster. Build the system as multiple clusters and connect the clusters using an intercluster bus, as shown in Fig. 6-6(a). As long as most CPUs communicate primarily within their own cluster, there will be relatively little intercluster traffic. If one intercluster bus proves to be inadequate, add a second intercluster bus, or arrange the clusters in a tree or grid. If still more bandwidth is needed, collect a bus, tree, or grid of clusters together into a super-cluster, and break the system into multiple superclusters. The superclusters can be connected by a bus, tree, or grid, and so on. Fig. 6-6(b) shows a system with three levels of buses.

Fig. 6-6. (a) Three clusters connected by an intercluster bus to form one supercluster. (b) Two superclusters connected by a supercluster bus.

In this section we will look at a hierarchical design based on a grid of clusters. The machine, called Dash, was built as a research project at stanford university (Lenoski et al., 1992). Although many other researchers are doing similar work, this one is a typical example. In the remainder of this section we with focus on the 64-CPU prototype that was actually constructed, but the design principles have been chosen carefully so that one could equally well build a much larger version.
The description given below has been simplified slightly in a few places to avoid going into unnecessary detail.

A simplified diagram of the Dash prototype is presented in Fig. 6-7(a). It consists of 16 clusters, each cluster containing a bus, four CPUs, 16M of the global memory, and some I/O equipment (disks, etc.). To avoid clutter in the figure, the I/O equipment and two of the CPUs have been omitted from each cluster. Each CPU is able to snoop on its local bus, as in Fig. 6-2(b), but not on other buses.

The total address space available in the prototype is 256M, divided up into 16 regions of 16M each. The global memory of cluster 0 holds addresses 0 to 16M. The global memory of cluster 1 holds addresses 16M to 32M, and so on. Memory is cached and transferred in units of 16-byte blocks, so each cluster has 1M memory blocks within its address space.

**Directories**

Each cluster has a **directory** that keeps track of which clusters currently have copies of its blocks. Since each cluster owns 1M memory blocks, it has 1M entries in its directory, one per block. Each entry holds a bit map with one bit per cluster telling whether or not that cluster has the block currently cached. The entry also has a 2-bit field telling the state of the block. The directories are essential to the operation of Dash, as we shall see. In fact, the name Dash comes from "Directory Architecture for Shared memory."

Having 1M entries of 18 bits each means that the total size of each directory is over 2M bytes. With 16 clusters, the total directory memory is just over 36M, or about 14 percent of the 256M. If the number of CPUs per cluster is increased, the amount of directory memory is not changed. Thus having more CPUs per cluster allows the directory cost to be amortized over a larger number of CPUs, reducing the cost per CPU. Also, the cost of the directory and bus controllers per CPU are reduced. In theory, the design works fine with one CPU per cluster, but the cost of the directory and bus hardware per CPU then becomes larger.

A bit map is not the only way to keep track of which cluster holds which cache block. An alternative approach is to organize each directory entry as an explicit list telling which clusters hold the corresponding cache block. If there is little sharing, the list approach will require fewer bits, but if there is substantial sharing, it will require more bits. Lists also have the disadvantage of being variable-length data structures, but these problems can be solved. The M.I.T. Alewife multiprocessor (Agarwal et al., 1991; and Kranz et al., 1993), for example, is similar to Dash in many respects, although it uses lists instead of bit maps in its directories and handles directory overflows in software.
Fig. 6-7. (a) A simplified view of the Dash architecture. Each cluster actually has four CPUs, but only two are shown here. (b) A Dash directory.

Each cluster in Dash is connected to an interface that allows the cluster to communicate with other clusters. The interfaces are connected by intercluster links (primitive buses) in a rectangular grid, as shown in Fig. 6-7(a). As more clusters are added to the system, more intercluster links are added, too, so the bandwidth increases and the system scales. The intercluster link system uses **wormhole routing**, which means that the first part of a packet can be forwarded even...
before the entire packet has been received, thus reducing the delay at each hop. Although not shown in the figure, there are actually two sets of intercluster links, one for request packets and one for reply packets. The intercluster links cannot be snooped upon.

**Caching**

Caching is done on two levels: a first-level cache and a larger second-level cache. The first-level cache is a subset of the second-level cache, so only the latter will concern us here. Each (second-level) cache monitors the local bus using a protocol somewhat similar to the cache ownership protocol of Fig. 6-4.

Each cache block can be in one of the following three states:

1. **UNCACHED** — The only copy of the block is in this memory.
2. **CLEAN** — Memory is up-to-date; the block may be in several caches.
3. **DIRTY** — Memory is incorrect; only one cache holds the block.

The state of each cache block is stored in the *State* field of its directory entry, as shown in Fig. 6-7(b).

**Protocols**

The Dash protocols are based on ownership and invalidation. At every instant, each cache block has a unique owner. For UNCACHED or CLEAN blocks, the block's home cluster is the owner. For dirty blocks, the cluster holding the one and only copy is the owner. Writing on a CLEAN block requires first finding and invalidating all existing copies. This is where the directories come in.

To see how this mechanism works, let us first consider how a CPU reads a memory word. It first checks its own caches. If neither cache has the word, a request is issued on the local cluster bus to see if another CPU in the cluster has the block containing it. If one does, a cache-to-cache transfer of the block is executed to place the block in the requesting CPU's cache. If the block is CLEAN, a copy is made; if it is DIRTY, the home directory is informed that the block is now CLEAN and shared. Either way, a hit from one of the caches satisfies the instruction but does not affect any directory's bit map.

If the block is not present in any of the cluster's caches, a request packet is sent to the block's home cluster, which can be determined by examining the upper 4 bits of the memory address. The home cluster might well be the requester's cluster, in which case the message is not sent physically. The directory management hardware at the home cluster examines its tables to see what state the block is in. If it is UNCACHED or CLEAN, the hardware fetches the block from its global memory and sends it back to the requesting cluster. It then updates its directory, marking the block as cached in the requester's cluster (if it was not already so marked).

If, however, the needed block is DIRTY, the directory hardware looks up the identity of the cluster holding the block and forwards the request there. The cluster holding the dirty block then sends it to the requesting cluster and marks its own copy as CLEAN because it is now shared. It also sends a copy back to the home cluster so that memory can be updated and the block state changed to CLEAN. All these cases are summarized in Fig. 6-8(a). Where a block is marked as being in a new state, it is the home directory that is changed, as it is the home directory that keeps track of the state.

Writes work differently. Before a write can be done, the CPU doing the write must be sure that it is the owner of the only copy of the cache block in the system. If it already has the block in its on-board cache and the block is dirty, the write can proceed immediately. If it has the block but
it is clean, a packet is first sent to the home cluster requesting that all other copies be tracked down and invalidated.

If the requesting CPU does not have the cache block, it issues a request on the local bus to see if any of the neighbors have it. If so, a cache-to-cache (or memory-to-cache) transfer is done. If the block is CLEAN, all other copies, if any, must be invalidated by the home cluster.

If the local broadcast fails to turn up a copy and the block is homed elsewhere, a packet is sent to the home cluster. Three cases can be distinguished here. If the block is UNCA\textit{ACHED}, it is marked dirty and sent to the requester. If it is CLEAN, all copies are invalidated and then the procedure for UNCA\textit{ACHED} is followed. If it is DIRTY, the request is forwarded to the remote cluster currently owning the block (if needed). This cluster invalidates its own copy and satisfies the request. The various cases are shown in Fig. 6-8(b).

<table>
<thead>
<tr>
<th>Location where the block was found</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Block state</strong></td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>UNCA\textit{ACHED}</td>
</tr>
<tr>
<td>CLEAN</td>
</tr>
<tr>
<td>DIRTY</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>Location where the block was found</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Block state</strong></td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>UNCA\textit{ACHED}</td>
</tr>
</tbody>
</table>
Fig. 6-8. Dash protocols. The columns show where the block was found. The rows show the state it was in. The contents of the boxes show the action taken. R refers to the requesting CPU. An empty box indicates an impossible situation. (a) Reads. (b) Writes.

Obviously, maintaining memory consistency in Dash (or any large multiprocessor) is nothing at all like the simple model of Fig. 6-1(b). A single memory access may require a substantial number of packets to be sent. Furthermore, to keep memory consistent, the access usually cannot be completed until all the packets have been acknowledged, which can have a serious effect on performance. To get around these problems, Dash uses a variety of special techniques, such as two sets of intercluster links, pipelined writes, and different memory semantics than one might expect. We will discuss some of these issues later. For the time being, the bottom line is that this implementation of "shared memory" requires a large data base (the directories), a considerable amount of computing power (the directory management hardware), and a potentially large number of packets that must be sent and acknowledged. We will see later that implementing distributed shared memory has precisely the same properties. The difference between the two lies much more in the implementation technique than in the ideas, architecture, or algorithms.

6.2.5. NUMA Multiprocessors

If nothing else, it should be abundantly clear by now that hardware caching in large multiprocessors is not simple. Complex data structures must be maintained by the hardware and intricate protocols, such as those of Fig. 6-8, must be built into the cache controller or MMU. The inevitable consequence is that large multiprocessors are expensive and not in widespread use.

However, researchers have spent a considerable amount of effort looking at alternative designs that do not require elaborate caching schemes. One such architecture is the NUMA (NonUniform Memory Access) multiprocessor. Like a traditional UMA (Uniform Memory Access) multiprocessor, a numa machine has a single virtual address space that is visible to all CPUs. When any CPU writes a value to location a, a subsequent read of a by a different processor will return the value just written.

The difference between UMA and NUMA machines lies not in the semantics but in the performance. On a NUMA machine, access to a remote memory is much slower than access to a
local memory, and no attempt is made to hide this fact by hardware caching. The ratio of a remote access to a local access is typically 10:1, with a factor of two variation either way not being unusual. Thus a CPU can directly execute a program that resides in a remote memory, but the program may run an order of magnitude slower than it would have had it been in local memory.

Examples of NUMA Multiprocessors

To make the concept of a NUMA machine clearer, consider the example of Fig. 6-9(a), Cm*, the first NUMA machine (Jones et al., 1977). The machine consisted of a number of clusters, each consisting of a CPU, a microprogrammable MMU, a memory module, and possibly some I/O devices, all connected by a bus. No caches were present, and no bus snooping occurred. The clusters were connected by intercluster buses, one of which is shown in the figure.

When a CPU made a memory reference, the request went to the CPU's MMU, which then examined the upper bits of the address to see which memory was needed. If the address was local, the MMU just issued a request on the local bus. If it was to a distant memory, the MMU built a request packet containing the address (and for a write, the data word to be written), and sent it to the destination cluster over an intercluster bus. Upon receiving the packet, the destination MMU carried out the operation and returned the word (for a read) or an acknowledgement (for a write). Although it was possible for a CPU to run entirely from a remote memory, sending a packet for each word read and each word written slowed down operation by an order of magnitude.
Fig. 6-9. (a) A simplified view of the Cm* system. (b) The BBN Butterfly. The CPUs on the right are the same as those on the left (i.e., the architecture is really a cylinder).

Figure 6-9(b) shows another NUMA machine, the BBN Butterfly. In this design, each CPU is coupled directly to one memory. Each of the small squares in Fig. 6-9(b) represents a CPU plus memory pair. The CPUs on the right-hand side of the figure are the same as those on the left. The CPUs are wired up via eight switches, each having four input ports and four output ports. Local memory requests are handled directly; remote requests are turned into request packets and sent to the appropriate memory via the switching network. Here, too, programs can run remotely, but at a tremendous penalty in performance.

Although neither of these examples has any global memory, NUMA machines can be equipped with memory that is not attached to any CPU.

Bolosky et al. (1989), for example, describe a bus-based NUMA machine that has a global memory that does not belong to any CPU but can be accessed by all of them (in addition to the local memories).

Properties of NUMA Multiprocessors

NUMA machines have three key properties that are of concern to us:
1. Access to remote memory is possible.
2. Accessing remote memory is slower than accessing local memory.
3. Remote access times are not hidden by caching.

The first two points are self explanatory. The third may require some clarification. In Dash and most other modern UMA multiprocessors, remote access is slower than local access as well. What makes this property bearable is the presence of caching. When a remote word is touched, a block of memory around it is fetched to the requesting processor's cache, so that subsequent references go at full speed. Although there is a slight delay to handle the cache fault, running out of remote memory can be only fractionally more expensive than running out of local memory. The consequence of this observation is that it does not matter so much which pages live in which memory: code and data are automatically moved by the hardware to wherever they are needed (although a bad choice of the home cluster for each page in Dash adds extra overhead).

NUMA machines do not have this property, so it matters a great deal which page is located in which memory (i.e., on which machine). The key issue in NUMA software is the decision of where to place each page to maximize performance. Below we will briefly summarize some ideas due to LaRowe and Ellis (1991). Other work is described in (Cox and Fowler, 1989; LaRowe et al., 1991; and Ramanathan and Ni, 1991).

When a program on a NUMA machine starts up, pages may or may not be manually prepositioned on certain processors' machines (their home processors). In either case, when a CPU tries to access a page that is not currently mapped into its address space, it causes a page fault. The operating system catches the fault and has to make a decision. If the page is read-only, the choice is to replicate the page (i.e., make a local copy without disturbing the original) or to map the virtual page onto the remote memory, thus forcing a remote access for all addresses on that page. If the page is read-write, the choice is to migrate the page to the faulting processor (invalidating the original page) or to map the virtual page onto the remote memory.

The trade-offs involved here are simple. If a local copy is made (replication or migration) and the page is not reused much, considerable time will have been wasted fetching it for nothing. On the other hand, if no copy is made, the page is mapped remote, and many accesses follow, they
will all be slow. In essence, the operating system has to guess if the page will be heavily used in
the future. If it guesses wrong, a performance penalty will be extracted.

Whichever decision is made, the page is mapped in, either local or remote, and the faulting
instruction restarted. Subsequent references to that page are done in hardware, with no software
intervention. If no other action were taken, then a wrong decision once made could never be
reversed.

**PAGE-BASED DISTRIBUTED SHARED MEMORY**

Having studied the principles behind distributed shared memory systems, let us now turn to
these systems themselves. In this section we will study "classical" distributed shared memory, the
first of which was IVY (Li 1986; and Li and Hudak 1989). These systems are built on top of
multicomputers, that is, processors connected by a specialized message-passing network,
workstations on a LAN, or similar designs. The essential element here is that no processor can
directly access any other processor's memory. Such systems are sometimes called NORMA (NO
Remote Memory Access) systems to contrast them with NUMA systems.

The big difference between NUMA and NORMA is that in the former, every processor can
directly reference every word in the global address space just by reading or writing it. Pages can be
randomly distributed among memories without affecting the results that programs give. When a
processor references a remote page, the system has the option of fetching it or using it remotely.
The decision affects the performance, but not the correctness. NUMA machines are true
multiprocessors — the hardware allows every processor to reference every word in the address
space without software intervention.

Workstations on a LAN are fundamentally different from a multiprocessor. Processors can only
reference their own local memory. There is no concept of a global shared memory, as there is with
a NUMA or UMA multiprocessor. The goal of the DSM work, however, is to add software to the
system to allow a multicomputer to run multiprocessor programs. Consequently, when a processor
references a remote page, that page *must* be fetched. There is no choice as there is in the NUMA
case.

Much of the early research on DSM systems was devoted to the question of how to run
existing multiprocessor programs on multicomputers. Sometimes this is referred to as the "dusty
deck" problem. The idea is to breathe new life into old programs just by running them on new
(DSM) systems. The concept is especially attractive for applications that need all the CPU cycles
they can get and whose authors are thus interested in using large-scale multicomputers rather than
small-scale multiprocessors.

Since programs written for multiprocessors normally assume that memory is sequentially
consistent, the initial work on DSM was carefully done to provide sequentially consistent memory,
so that old multiprocessor programs could work without modification. Subsequent experience has
shown that major performance gains can be had by relaxing the memory model, at the cost of
reprogramming existing applications and writing new ones in a different style. We will come back to
this point later, but first we will look at the major design issues in classical DSM systems of the IVY
type.

**6.4.1. Basic Design**

The idea behind DSM is simple: try to emulate the cache of a multiprocessor using the MMU
and operating system software. In a DSM system, the address space is divided up into chunks, with
the chunks being spread over all the processors in the system. When a processor references an
address that is not local, a trap occurs, and the DSM software fetches the chunk containing the
address and restarts the faulting instruction, which now completes successfully. This concept is
illustrated in Fig. 6-25(a) for an address space with 16 chunks and four processors, each capable of
holding four chunks.
In this example, if processor 1 references instructions or data in chunks 0, 2, 5, or 9, the references are done locally. References to other chunks cause traps. For example, a reference to an address in chunk 10 will cause a trap to the DSM software, which then moves chunk 10 from machine 2 to machine 1, as shown in Fig. 6-25(b).

6.4.2. Replication

One improvement to the basic system that can improve performance considerably is to replicate chunks that are read only, for example, program text, readonly constants, or other read-only data structures. For example, if chunk 10 in Fig. 6-25 is a section of program text, its use by processor 1 can result in a copy being sent to processor 1, without the original in processor 2's memory being disturbed, as shown in Fig. 6-25(c). In this way, processors 1 and 2 can both reference chunk 10 as often as needed without causing traps to fetch missing memory.

Fig. 6-25. (a) Chunks of address space distributed among four machines. (b) Situation after CPU 1 references chunk 10. (c) Situation if chunk 10 is read only and replication is used.

Another possibility is to replicate not only read-only chunks, but all chunks. As long as reads are being done, there is effectively no difference between replicating a read-only chunk and
replicating a read-write chunk. However, if a replicated chunk is suddenly modified, special action has to be taken to prevent having multiple, inconsistent copies in existence. How inconsistency is prevented will be discussed in the following sections.

6.4.3. Granularity

DSM systems are similar to multiprocessors in certain key ways. In both systems, when a nonlocal memory word is referenced, a chunk of memory containing the word is fetched from its current location and put on the machine making the reference (main memory or cache, respectively). An important design issue is how big should the chunk be? Possibilities are the word, block (a few words), page, or segment (multiple pages).

With a multiprocessor, fetching a single word or a few dozen bytes is feasible because the MMU knows exactly which address was referenced and the time to set up a bus transfer is measured in nanoseconds. Memnet, although not strictly a multiprocessor, also uses a small chunk size (32 bytes). With DSM systems, such fine granularity is difficult or impossible, due to the way the MMU works.

When a process references a word that is absent, it causes a page fault. An obvious choice is to bring in the entire page that is needed. Furthermore, integrating DSM with virtual memory makes the total design simpler, since the same unit, the page, is used for both. On a page fault, the missing page is just brought in from another machine instead of from the disk, so much of the page fault handling code is the same as in the traditional case.

However, another possible choice is to bring in a larger unit, say a region of 2, 4, or 8 pages, including the needed page. In effect, doing this simulates a larger page size. There are advantages and disadvantages to a larger chunk size for DSM. The biggest advantage is that because the startup time for a network transfer is substantial, it does not take much longer to transfer 1024 bytes than it does to transfer 512 bytes. By transferring data in large units, when a large piece of address space has to be moved, the number of transfers may often be reduced. This property is especially important because many programs exhibit locality of reference, meaning that if a program has referenced one word on a page, it is likely to reference other words on the same page in the immediate future.

On the other hand, the network will be tied up longer with a larger transfer, blocking other faults caused by other processes. Also, too large an effective page size introduces a new problem, called false sharing, illustrated in Fig. 6-26. Here we have a page containing two unrelated shared variables, A and B. Processor 1 makes heavy use of A, reading and writing it. Similarly, process 2 uses B. Under these circumstances, the page containing both variables will constantly be traveling back and forth between the two machines.

![Fig. 6-26. False sharing of a page containing two unrelated variables.](image)

The problem here is that although the variables are unrelated, since they appear by accident on the same page, when a process uses one of them, it also gets the other. The larger the effective page size, the more often false sharing will occur, and conversely, the smaller the effective page size.
size, the less often it will occur. Nothing analogous to this phenomenon is present in ordinary virtual memory systems.

Clever compilers that understand the problem and place variables in the address space accordingly can help reduce false sharing and improve performance. However, saying this is easier than doing it. Furthermore, if the false sharing consists of processor 1 using one element of an array and processor 2 using a different element of the same array, there is little that even a clever compiler can do to eliminate the problem.

6.4.4. Achieving Sequential Consistency

If pages are not replicated, achieving consistency is not an issue. There is exactly one copy of each page, and it is moved back and forth dynamically as needed. With only one copy of each page, there is no danger that different copies will have different values.

If read-only pages are replicated, there is also no problem. The read-only pages are never changed, so all the copies are always identical. Only a single copy is kept of each read-write page, so inconsistencies are impossible here, too.

The interesting case is that of replicated read-write pages. In many DSM systems, when a process tries to read a remote page, a local copy is made because the system does not know what is on the page or whether it is writable. Both the local copy (in fact, all copies) and the original page are set up in their respective MMUs as read only. As long as all references are reads, everything is fine.

However, if any process attempts to write on a replicated page, a potential consistency problem arises because changing one copy and leaving the others alone is unacceptable. This situation is analogous to what happens in a multiprocessor when one processor attempts to modify a word that is present in multiple caches, so let us review what multiprocessors do under these circumstances.

In general, multiprocessors take one of two approaches: update or invalidation. With update, the write is allowed to take place locally, but the address of the modified word and its new value are broadcast on the bus simultaneously to all the other caches. Each of the caches holding the word being updated sees that an address it is caching is being modified, so it copies the new value from the bus to its cache, overwriting the old value. The final result is that all caches that held the word before the update also hold it afterward, and all acquire the new value.

The other approach multiprocessors can take is invalidation. When this strategy is used, the address of the word being updated is broadcast on the bus, but the new value is not. When a cache sees that one of its words is being updated, it invalidates the cache block containing the word, effectively removing it from the cache. The final result with invalidation is that only one cache now holds the modified word, so consistency problems are avoided. If one of the processors that now holds an invalid copy of the cache block tries to use it, it will get a cache miss and fetch the block from the one processor holding a valid copy.

Whereas these two strategies are approximately equally easy to implement in a multiprocessor, they differ radically in a DSM system. Unlike in a multiprocessor, where the MMU knows which word is to be written and what the new value is, in a DSM system the software does not know which word is to be written or what the new value will be. To find out, it could make a secret copy of the page about to be changed (the page number is known), make the page writable, set the hardware trap bit, which forces a trap after every instruction, and restart the faulting process. One instruction later, it catches the trap and compares the current page with the secret copy it just made, to see which word has been changed. It could then broadcast a short packet giving the address and new value on the network. The processors receiving this packet could then check to see if they have the page in question, and if so, update it.

The amount of work here is enormous, but worse yet, the scheme is not foolproof. If several updates, originating on different processors, take place simultaneously, different processors may
see them in a different order, so the memory will not be sequentially consistent. In a multiprocessor this problem does not occur because broadcasts on the bus are totally reliable (no lost messages), and the order is unambiguous.

Another issue is that a process may make thousands of consecutive writes to the same page because many programs exhibit locality of reference. Having to catch all these updates and pass them to remote machines is horrendously expensive in the absence of multiprocessor-type snooping.

For these reasons, page-based DSM systems typically use an invalidation protocol instead of an update protocol. Various protocols are possible. Below we will describe a typical example, in which all pages are potentially writable (i.e., the DSM software does not know what is on which page).

In this protocol, at any instant of time, each page is either in $R$ (readable) or $W$ (readable and writable) state. The state a page is in may change as execution progresses. Each page has an owner, namely the process that most recently wrote on the page. When a page is in $W$ state, only one copy exists, mapped into the owner’s address space in read-write mode. When a page is in $R$ state, the owner has a copy (mapped read only), but other processes may have copies, too.

Six cases can be distinguished, as shown in Fig. 6-27. In all the examples in the figure, process $P$ on processor 1 wants to read or write a page. The cases differ in terms of whether $P$ is the owner, whether $P$ has a copy, whether other processes have copies, and what the state of the page is, as shown.

Let us now consider the actions taken in each of the cases. In the first four cases of Fig. 6-27(a), $P$ just does the read. In all four cases the page is mapped into its address space, so the read is done in hardware. No trap occurs. In the fifth and sixth cases, the page is not mapped in, so a page fault occurs and the DSM software gets control. It sends a message to the owner asking for a copy. When the copy comes back, the page is mapped in and the faulting instruction is restarted. If the owner had the page in $W$ state, it must degrade to $R$ state, but may keep the page. In this protocol, the other process keeps ownership, but in a slightly different protocol that could be transferred as well.

Writes are handled differently, as depicted in Fig. 6-27(b). In the first case, the write just happens, without a trap, since the page is mapped in read-write mode. In the second case (no other copies), the page is changed to $W$ state and written. In the third case there are other copies, so they must first be invalidated before the write can take place.

In the next three cases, some other process is the owner at the time $P$ does the write. In all three cases, $P$ must ask the current owner to invalidate any existing copies, pass ownership to $P$, and send a copy of the page unless $P$ already has a copy. Only then may the write take place. In all three cases, $P$ ends up with the only copy of the page, which is in $W$ state.

In all six cases, before a write is performed the protocol guarantees that only one copy of the page exists, namely in the address space of the process about to do the write. In this way, consistency is maintained.

**6.4.5. Finding the Owner**

We glossed over a few points in the description above. One of them is how to find the owner of the page. The simplest solution is by doing a broadcast, asking for the owner of the specified page to respond. Once the owner has been located this way, the protocol can proceed as above.

An obvious optimization is not just to ask who the owner is, but also to tell whether the sender wants to read or write and say whether it needs a copy of the page. The owner can then send a single message transferring ownership and the page as well, if needed.
Broadcasting has the disadvantage of interrupting each processor, forcing it to inspect the request packet. For all the processors except the owner’s, handling the interrupt is essentially wasted time. Broadcasting may use up considerable network bandwidth, depending on the hardware.

Li and Hudak (1989) describe several other possibilities as well. In the first of these, one process is designated as the page manager. It is the job of the manager to keep track of who owns each page. When a process, $P$, wants to read a page it does not have or wants to write a page it does not own, it sends a message to the page manager telling which operation it wants to perform and on which page. The manager then sends back a message telling who the owner is. $P$ now

---

**Fig. 6-27.** (a) Process $P$ wants to read a page. (b) Process $P$ wants to write a page.
contacts the owner to get the page and/or the ownership, as required. Four messages are needed for this protocol, as illustrated in Fig. 6-28(a).

![Diagram of ownership location protocol](image)

**Fig. 6-28.** Ownership location using a central manager. (a) Four-message protocol. (b) Three-message protocol.

An optimization of this ownership location protocol is shown in Fig. 6-28(b). Here the page manager forwards the request directly to the owner, which then replies directly back to P, saving one message.

A problem with this protocol is the potentially heavy load on the page manager, handling all the incoming requests. This problem can be reduced by having multiple page managers instead of just one. Splitting the work over multiple managers introduces a new problem, however — finding the right manager. A simple solution is to use the low-order bits of the page number as an index into a table of managers. Thus with eight page managers, all pages that end with 000 are handled by manager 0, all pages that end with 001 are handled by manager 1, and so on.

Still another possible algorithm is having each process (or more likely, each processor) keep track of the probable owner of each page. Requests for ownership are sent to the probable owner, which forwards them if ownership has changed. If ownership has changed several times, the request message will also have to be forwarded several times. At the start of execution and every n times ownership changes, the location of the new owner should be broadcast, to allow all processors to update their tables of probable owners.

### 6.4.6. Finding the Copies

Another important detail is how all the copies are found when they must be invalidated. Again, two possibilities present themselves. The first is to broadcast a message giving the page number and ask all processors holding the page to invalidate it. This approach works only if broadcast messages are totally reliable and can never be lost.

The second possibility is to have the owner or page manager maintain a list or **copyset** telling which processors hold which pages, as depicted in Fig. 6-29. Here page 4, for example, is owned by a process on CPU 1, as indicated by the double box around the 4. The copyset consists of 2 and 4, because copies of page 4 can be found on those machines.
The owner of each page maintains a copyset telling which other CPUs are sharing that page. Page ownership is indicated by the double boxes.

When a page must be invalidated, the old owner, new owner, or page manager sends a message to each processor holding the page and waits for an acknowledgement. When each message has been acknowledged, the invalidation is complete.

Dash and Memnet also need to invalidate pages when a new writer suddenly appears, but they do it differently. Dash uses directories. The writing process sends a packet to the directory (the page manager in our terminology), which then finds all the copies from its bit map, sends each one an invalidation packet, and collects all the acknowledgements. Memnet fetches the needed page and invalidates all copies by broadcasting an invalidation packet on the ring. The first processor having a copy puts it in the packet and sets a header bit saying it is there. Subsequent processors just invalidate their copies. When the packet comes around the ring and arrives back at the sender, the needed data are present and all other copies are gone. In effect, Memnet implements DSM in hardware.

6.4.7. Page Replacement

In a DSM system, as in any system using virtual memory, it can happen that a page is needed but that there is no free page frame in memory to hold it. When this situation occurs, a page must be evicted from memory to make room for the needed page. Two subproblems immediately arise: which page to evict and where to put it.

To a large extent, the choice of which page to evict can be made using traditional virtual memory algorithms, such as some approximation to the least recently used (LRU) algorithm. One complication that occurs with DSM is that pages can be invalidated spontaneously (due to the activities of other processes), which affects the possible choices. However, by maintaining the estimated LRU order of only those pages that are currently valid, any of the traditional algorithms can be used.

As with conventional algorithms, it is worth keeping track of which pages are "clean" and which are "dirty." In the context of DSM, a replicated page that another process owns is always a prime candidate to evict because it is known that another copy exists. Consequently, the page does not have to be saved anywhere. If a directory scheme is being used to keep track of copies, the owner or page manager must be informed of this decision, however. If pages are located by broadcasting, the page can just be discarded.

The second best choice is a replicated page that the evicting process owns. It is sufficient to pass ownership to one of the other copies by informing that process, the page manager, or both, depending on the implementation. The page itself need not be transferred, which results in a smaller message.

If no replicated pages are suitable candidates, a nonreplicated page must be chosen, for example, the least recently used valid page. There are two possibilities for getting rid of it. The first is to write it to a disk, if present. The other is to hand it off to another processor.

Choosing a processor to hand a page off to can be done in several ways. For example, each page could be assigned a home machine, which must accept it, although this probably implies reserving a large amount of normally wasted space to hold pages that might be sent home some day. Alternatively, the number of free page frames could be piggybacked on each message sent, with each processor building up an idea of how free memory was distributed around the network. An occasional broadcast message giving the exact count of free page frames could help keep these numbers up to date.

As an aside, note that a conflict may exist between choosing a replicated page (which may just be discarded) and choosing a page that has not been referenced in a long time (which may be
the only copy). The same problem exists in traditional virtual memory systems, however, so the same compromises and heuristics apply.

One problem that is unique to DSM systems is the network traffic generated when processes on different machines are actively sharing a writable page, either through false sharing or true sharing. An ad hoc way to reduce this traffic is to enforce a rule that once a page has arrived at any processor, it must remain there for some time AT. If requests for it come in from other machines, these requests are simply queued until the timer expires, thus allowing the local process to make many memory references without interference.

As usual, it is instructive to see how page replacement is handled in multiprocessors. In Dash, when a cache fills up, the option always exists of writing the block back to main memory. In DSM systems, that possibility does not exist, although using a disk as the ultimate repository for pages nobody wants is often feasible. In Memnet, every cache block has a home machine, which is required to reserve storage for it. This design is also possible in a DSM system, although it is wasteful in both Memnet and DSM.

6.4.8. Synchronization

In a DSM system, as in a multiprocessor, processes often need to synchronize their actions. A common example is mutual exclusion, in which only one process at a time may execute a certain part of the code. In a multiprocessor, the TEST-AND-SET-LOCK (TSL) instruction is often used to implement mutual exclusion. In normal use, a variable is set to 0 when no process is in the critical section and to 1 when one process is. The TSL instruction reads out the variable and sets it to 1 in a single, atomic operation. If the value read is 1, the process just keeps repeating the TSL instruction until the process in the critical region has exited and set the variable to 0.

In a DSM system, this code is still correct, but is a potential performance disaster. If one process, A, is inside the critical region and another process, B, (on a different machine) wants to enter it, B will sit in a tight loop testing the variable, waiting for it to go to zero. The page containing the variable will remain on ZTs machine. When A exits the critical region and tries to write 0 to the variable, it will get a page fault and pull in the page containing the variable. Immediately thereafter, B will also get a page fault, pulling the page back. This performance is acceptable.

The problem occurs when several other processes are also trying to enter the critical region. Remember that the TSL instruction modifies memory (by writing a 1 to the synchronization variable) every time it is executed. Thus every time one process executes a TSL instruction, it must fetch the entire page containing the synchronization variable from whoever has it. With multiple processes each issuing a TSL instruction every few hundred nanoseconds, the network traffic can become intolerable.

For this reason, an additional mechanism is often needed for synchronization. One possibility is a synchronization manager (or managers) that accept messages asking to enter and leave critical regions, lock and unlock variables, and so on, sending back replies when the work is done. When a region cannot be entered or a variable cannot be locked, no reply is sent back immediately, causing the sender to block. When the region becomes available or the variable can be locked, a message is sent back. In this way, synchronization can be done with a minimum of network traffic, but at the expense of centralizing control per lock.
UNIT VI

INTRODUCTION TO AMOEBA

In this section we will give an introduction to Amoeba, starting with a brief history and its current research goals. Then we will look at the architecture of a typical Amoeba system. Finally, we will begin our study of the Amoeba software, both the kernel and the servers.

History of Amoeba

Amoeba originated at the Vrije Universiteit, Amsterdam, The Netherlands in 1981 as a research project in distributed and parallel computing. It was designed primarily by Andrew S. Tanenbaum and three of his Ph.D. students, Frans Kaashoek, Sape J. Mullender, and Robbert van Renesse, although many other people also contributed to the design and implementation. By 1983, an initial prototype, Amoeba 1.0, was operational.

Starting in 1984, the Amoeba fissioned, and a second group was set up at the Centre for Mathematics and Computer Science, also in Amsterdam, under Mullender's leadership. In the succeeding years, this cooperation was extended to sites in England and Norway in a wide-area distributed system project sponsored by the European Community. This work used Amoeba 3.0, which unlike the earlier versions, was based on RPC. Using Amoeba 3.0, it was possible for clients in Tromso to access servers in Amsterdam transparently, and vice versa.

The system evolved for several years, acquiring such features as partial UNIX emulation, group communications and a new low-level protocol. The version described in this chapter is Amoeba 5.2.

Research Goals

Many research projects in distributed operating systems have started with an existing system (e.g., UNIX) and added new features, such as networking and a shared file system, to make it more distributed. The Amoeba project took a different approach. It started with a clean slate and developed a new system from scratch. The idea was to make a fresh start and experiment with new ideas without having to worry about backward compatibility with any existing system. To avoid the chore of having to rewrite a huge amount of application software from scratch as well, a UNIX emulation package was added later.

The primary goal of the project was to build a transparent distributed operating system. To the average user, using Amoeba is like using a traditional timesharing system like UNIX. One logs in, edits and compiles programs, moves files around, and so on. The difference is that each of these actions makes use of multiple machines over the network. These include process servers, file servers, directory servers, compute servers, and other machines, but the user is not aware of any of this. At the terminal, it just looks like a timesharing system.

An important distinction between Amoeba and most other distributed systems is that Amoeba has no concept of a "home machine." When a user logs in, it is to the system as a whole, not to a specific machine. Machines do not have owners. The initial shell, started upon login, runs on some arbitrary machine, but as commands are started up, in general they do not run on the same machine as the shell. Instead, the system automatically looks around for the most lightly loaded machine to run each new command on. During the course of a long terminal session, the processes that run on behalf of any one user will be spread more-or-less uniformly spread over all the machines in the system, depending on the load, of course. In this respect, Amoeba is highly location transparent.

In other words, all resources belong to the system as a whole and are managed by it. They are not dedicated to specific users, except for short periods of time to run individual processes. This model attempts to provide the transparency that is the holy grail of all distributed systems designers.
A simple example is *amake*, the Amoeba replacement for the UNIX *make* program. When the user types *amake*, all the necessary compilations happen, as expected, except that the system (and not the user) determines whether they happen sequentially or in parallel, and on which machine or machines this occurs. None of this is visible to the user.

A secondary goal of Amoeba is to provide a testbed for doing distributed and parallel programming. While some users just use Amoeba the same way they would use any other timesharing system most users are specifically interested in experimenting with distributed and parallel algorithms, languages, tools, and applications. Amoeba supports these users by making the underlying parallelism available to people who want to take advantage of it. In practice, most of Amoeba's current user base consists of people who are specifically interested in distributed and parallel computing in its various forms. A language, Orca, has been specifically designed and implemented on Amoeba for this purpose. Orca and its applications are described in (Bal, 1991; Bal et al., 1990; and Tanenbaum et al., 1992). Amoeba itself, however, is written in C.

**The Amoeba System Architecture**

Before describing how Amoeba is structured, it is useful first to outline the kind of hardware configuration for which Amoeba was designed, since it differs somewhat from what most organizations presently have. Amoeba was designed with two assumptions about the hardware in mind:

1. Systems will have a very large number of CPUs.
2. Each CPU will have tens of megabytes of memory.

These assumptions are already true at some installations, and will probably become true at almost all corporate, academic, and governmental sites within a few years.

The driving force behind the system architecture is the need to incorporate large numbers of CPUs in a straightforward way. In other words, what do you do when you can afford 10 or 100 CPUs per user? One solution is to give each user a personal 10-node or 100-node multiprocessor.

Although giving everyone a personal multiprocessor is certainly a possibility, doing so is not an effective way to spend the available budget. Most of the time, nearly all the processors will be idle, but some users will want to run massively parallel programs and will not be able to harness all the idle CPU cycles because they are in other users' personal machines.

Instead of this personal multiprocessor approach, Amoeba is based on the model shown in Fig. 7-1. In this model, all the computing power is located in one or more **processor pools**. A processor pool consists of a substantial number of CPUs, each with its own local memory and network connection. Shared memory is not required, or even expected, but if it is present it could be used to optimize message passing by doing memory-to-memory copying instead of sending messages over the network.

![Fig. 7-1. The Amoeba system architecture.](image)
The CPUs in a pool can be of different architectures, for example, a mixture of 680x0, 386, and SPARC machines. Amoeba has been designed to deal with multiple architectures and heterogeneous systems. It is even possible for the children of a single process to run on different architectures.

Pool processors are not "owned" by any one user. When a user types a command, the operating system dynamically chooses one or more processors on which to run that command. When the command completes, the processes are terminated and the resources held go back into the pool, waiting for the next command, very likely from a different user. If there is a shortage of pool processors, individual processors are timeshared, with new processes being assigned to the most lightly loaded CPUs. The important point to note here is that this model is quite different from current systems in which each user has exactly one personal workstation for all his computing activities.

The expected presence of large memories in future systems has influenced the design in many ways. Many time-space trade-offs have been made to provide high performance at the cost of using more memory. We will see examples later.

The second element of the Amoeba architecture is the terminal. It is through the terminal that the user accesses the system. A typical Amoeba terminal is an X terminal, with a large bit-mapped screen and a mouse. Alternatively, a personal computer or workstation running X windows can also be used as a terminal. Although Amoeba does not forbid running user programs on the terminal, the idea behind this model is to give the users relatively cheap terminals and concentrate the computing cycles into a common pool so that they can be used more efficiently.

Pool processors are inherently cheaper than workstations because they consist of just a single board with a network connection. There is no keyboard, monitor, or mouse, and the power supply can be shared by many boards. Thus, instead of buying 100 high-performance workstations for 100 users, one might buy 50 high-performance pool processors and 100 X terminals for the same price (depending on the economics, obviously). Since the pool processors are allocated only when needed, an idle user only ties up an inexpensive X terminal instead of an expensive workstation. The trade-offs inherent in the pool processor model versus the workstation model were discussed in Chap. 4.

To avoid any confusion, the pool processors do not have to be single-board computers. If these are not available, a subset of the existing personal computers or workstations can be designated as pool processors. They also do not need to be located in one room. The physical location is actually irrelevant. The pool processors can even be in different countries, as we will discuss later.

Another important component of the Amoeba configuration consists of specialized servers, such as file servers, which for hardware or software reasons need to run on a separate processor. In some cases a server is able to run on a pool processor, being started up as needed, but for performance reasons it is better to have it running all the time.

Servers provide services. A service is an abstract definition of what the server is prepared to do for its clients. This definition defines what the client can ask for and what the results will be, but it does not specify how many servers are working together to provide the service. In this way, the system has a mechanism for providing fault-tolerant services by having multiple servers doing the work.

An example is the directory server. There is nothing inherent about the directory server or the system design that would prevent a user from starting up a new directory server on a pool processor every time he wanted to look up a file name. However, doing so would be horrendously inefficient, so one or more directory servers are kept running all the time, generally on dedicated machines to enhance their performance. The decision to have some servers always running and others to be started explicitly when needed is up to the system administrator.
The Amoeba Microkernel

Having looked at the Amoeba hardware model, let us now turn to the software model. Amoeba consists of two basic pieces: a microkernel, which runs on every processor, and a collection of servers that provide most of the traditional operating system functionality. The overall structure is shown in Fig. 7-2.

The Amoeba microkernel runs on all machines in the system. The same kernel can be used on the pool processors, the terminals (assuming that they are computers, rather than X terminals), and the specialized servers. The microkernel has four primary functions:

1. Manage processes and threads.
2. Provide low-level memory management support.
3. Support communication.
4. Handle low-level I/O.

Let us consider each of these in turn.

Like most operating systems, Amoeba supports the concept of a process. In addition, Amoeba also supports multiple threads of control within a single address space. A process with one thread is essentially the same as a process in UNIX. Such a process has a single address space, a set of registers, a program counter, and a stack.

In contrast, although a process with multiple threads still has a single address space shared by all threads, each thread logically has its own registers, its own program counter, and its own stack. In effect, a collection of threads in a process is similar to a collection of independent processes in UNIX, with the one exception that they all share a single common address space.

A typical use for multiple threads might be in a file server, in which every incoming request is assigned to a separate thread to work on. That thread might begin processing the request, then block waiting for the disk, then continue work. By splitting the server up into multiple threads, each thread can be purely sequential, even if it has to block waiting for I/O. Nevertheless, all the threads can, for example, have access to a single shared software cache. Threads can synchronize using semaphores or mutexes to prevent two threads from accessing the shared cache simultaneously.

The second task of the kernel is to provide low-level memory management. Threads can allocate and deallocate blocks of memory, called segments. These segments can be read and written, and can be mapped into and out of the address space of the process to which the calling thread belongs. A process must have at least one segment, but it may also have many more of them. Segments can be used for text, data, stack, or any other purpose the process desires. The operating system does not enforce any particular pattern on segment usage.
The third job of the kernel is to handle interprocess communication. Two forms of communication are provided: point-to-point communication and group communication. These are closely integrated to make them similar.

Point-to-point communication is based on the model of a client sending a message to a server, then blocking until the server has sent a reply back. This request/reply exchange is the basis on which almost everything else is built.

The other form of communication is group communication. It allows a message to be sent from one source to multiple destinations. Software protocols provide reliable, fault-tolerant group communication to user processes in the presence of lost messages and other errors.

The fourth function of the kernel is to manage low-level I/O. For each I/O device attached to a machine, there is a device driver in the kernel. The driver manages all I/O for the device. Drivers are linked with the kernel and cannot be loaded dynamically.

Device drivers communicate with the rest of the system by the standard request and reply messages. A process, such as a file server, that needs to communicate with the disk driver, sends it request messages and gets back replies. In general, the client does not have to know that it is talking to a driver. As far as it is concerned, it is just communicating with a thread somewhere.

Both the point-to-point message system and the group communication make use of a specialized protocol called FLIP. This protocol is a network layer protocol and has been designed specifically to meet the needs of distributed computing. It deals with both unicasting and multicasting on complex internetworks. It will be discussed later.

**The Amoeba Servers**

Everything that is not done by the kernel is done by server processes. The idea behind this design is to minimize kernel size and enhance flexibility. By not building the file system and other standard services into the kernel, they can be changed easily, and multiple versions can run simultaneously for different user populations.

Amoeba is based on the client-server model. Clients are typically written by the users and servers are typically written by the system programmers, but users are free to write their own servers if they wish. Central to the entire software design is the concept of an object, which is like an abstract data type. Each object consists of some encapsulated data with certain operations defined on it. File objects have a READ operation, for example, among others.

Objects are managed by servers. When a process creates an object, the server that manages the object returns to the client a cryptographically protected capability for the object. To use the object later, the proper capability must be presented. All objects in the system, both hardware and software, are named, protected, and managed by capabilities. Among the objects supported this way are files, directories, memory segments, screen windows, processors, disks, and tape drives. This uniform interface to all objects provides generality and simplicity.

All the standard servers have stub procedures in the library. To use a server, a client normally just calls the stub, which marshals the parameters, sends the message, and blocks until the reply comes back. This mechanism hides all the details of the implementation from the user. A stub compiler is available for users who wish to produce stub procedures for their own servers.

Probably the most important server is the file server, known as the bullet server. It provides primitives to manage files, creating them, reading them, deleting them, and so on. Unlike most file servers, the files it creates are immutable. Once created, a file cannot be modified, but it can be deleted. Immutable files make automatic replication easier since they avoid many of the race conditions inherent in replicating files that are subject to being changed during the replication process.

Another important server is the directory server, for obscure historical reasons also known as the soap server. It is the directory server that manages directories and path names and maps
them onto capabilities. To read a file, a process asks the directory server to look up the path name. On a successful lookup, the directory server returns the capability for the file (or other object). Subsequent operations on the file do not use the directory server, but go straight to the file server. Splitting the file system into these two components increases flexibility and makes each one simpler, since it only has to manage one type of object (directories or files), not two.

Other standard servers are present for handling object replication, starting processes, monitoring servers for failures, and communicating with the outside world. User servers perform a wide variety of application-specific tasks.

The rest of this chapter is structured as follows. First we will describe objects and capabilities, since these are the heart of the entire system. Then we will look at the kernel, focusing on process management, memory management, and communication. Finally, we will examine some of the main servers, including the bullet server, the directory server, the replication server, and the run server.

**OBJECTS AND CAPABILITIES IN AMOEBA**

The basic unifying concept underlying all the Amoeba servers and the services they provide is the **object**. An object is an encapsulated piece of data upon which certain well-defined operations may be performed. It is, in essence, an abstract data type. Objects are passive. They do not contain processes or methods or other active entities that "do" things. Instead, each object is managed by a server process.

To perform an operation on an object, a client does an RPC with the server, specifying the object, the operation to be performed, and optionally, any parameters needed. The server does the work and returns the answer. Operations are performed synchronously, that is, after initiating an RPC with a server to get some work done, the client thread is blocked until the server replies. Other threads in the same process are still runnable, however.

Clients are unaware of the locations of the objects they use and the servers that manage these objects. A server might be running on the same machine as the client, on a different machine on the same LAN, or even on a machine thousands of kilometers away. Furthermore, although most servers run as user processes, a few low-level ones, such as the segment (i.e., memory) server and process server, run as threads in the kernel. This distinction, too, is invisible to clients. The RPC protocol for talking to user servers or kernel servers, whether local or remote, is identical in all cases. Thus a client is concerned entirely with what it wants to do, not where objects are stored and where servers run. A certain directory contains the capabilities for all the accessible file servers along with a specification of the default choice, so a user can override the default in cases where it matters. Usually, the system administrator sets up the default to be the local one.

### 7.2.1. Capabilities

Objects are named and protected in a uniform way, by special tickets called **capabilities**. To create an object, a client does an rpc with the appropriate server specifying what it wants. the server then creates the object and returns a capability to the client. On subsequent operations, the client must present the capability to identify the object. A capability is just a long binary number. The Amoeba 5.2 format is shown in Fig. 7-3.

![Fig. 7-3. A capability in Amoeba.](image-url)

When a client wants to perform an operation on an object, it calls a stub procedure that builds a message containing the object's capability and then traps to the kernel. The kernel extracts
the Server port field from the capability and looks it up in its cache to locate the machine on which the server resides. If the port is not in the cache, it is located by broadcasting, as will be described later. The port is effectively a logical address at which the server can be reached. Server ports are thus associated with a particular server (or a set of servers), not with a specific machine. If a server moves to a new machine, it takes its server port with it. Many server ports, like that of the file server, are publicly known and stable for years. The only way a server can be addressed is via its port, which it initially chose itself.

The rest of the information in the capability is ignored by the kernels and passed to the server for its own use. The Object field is used by the server to identify the specific object in question. For example, a file server might manage thousands of files, with the object number being used to tell it which one is being operated on. In a sense, the Object field in a file capability is analogous to a UNIX i-node number.

The Rights field is a bit map telling which of the allowed operations the holder of a capability may perform. For example, although a particular object may support reading and writing, a specific capability may be constructed with all the rights bits except read turned off.

The Check field is used for validating the capability. Capabilities are manipulated directly by user processes. Without some form of protection, there would be no way to prevent user processes from forging capabilities.

Object Protection

The basic algorithm used to protect objects is as follows. When an object is created, the server picks a random Check field and stores it both in the new capability and inside its own tables. All the rights bits in a new capability are initially on, and it is this owner capability that is returned to the client. When the capability is sent back to the server in a request to perform an operation, the Check field is verified.

To create a restricted capability, a client can pass a capability back to the server, along with a bit mask for the new rights. The server takes the original Check field from its tables, EXCLUSIVE ORs it with the new rights (which must be a subset of the rights in the capability), and then runs the result through a one-way function. Such a function, \( y = f(x) \), has the property that given \( x \) it is easy to find \( y \), but given only \( y \), finding \( x \) requires an exhaustive search of all possible \( x \) values (Evans et al., 1974).

The server then creates a new capability, with the same value in the Object field, but the new rights bits in the Rights field and the output of the one-way function in the Check field. The new capability is then returned to the caller. The client may send this new capability to another process, if it wishes, as capabilities are managed entirely in user space.

The method of generating restricted capabilities is illustrated in Fig. 7-4. In this example, the owner has turned off all the rights except one. For example, the restricted capability might allow the object to be read, but nothing else. The meaning of the Rights field is different for each object type since the legal operations themselves also vary from object type to object type.
Fig. 7-4. Generation of a restricted capability from an owner capability.

When the restricted capability comes back to the server, the server sees from the Rights field that it is not an owner capability because at least one bit is turned off. The server then fetches the original random number from its tables, EXCLUSIVE ORs it with the Rights field from the capability, and runs the result through the one-way function. If the result agrees with the Check field, the capability is accepted as valid.

It should be obvious from this algorithm that a user who tries to add rights that he does not have will simply invalidate the capability. Inverting the Check field in a restricted capability to get the argument (C XOR 00000001 in Fig. 7-4) is impossible because the function \( f \) is a one-way function (that is what "one-way" means — no algorithm exists for inverting it). It is through this cryptographic technique that capabilities are protected from tampering.

Capabilities are used throughout Amoeba for both naming of all objects and for protecting them. This single mechanism leads to a uniform naming and protection scheme. It also is fully location transparent. To perform an operation on an object, it is not necessary to know where the object resides. In fact, even if this knowledge were available, there would be no way to use it.

Note that Amoeba does not use access control lists for authentication. The protection scheme used requires almost no administrative overhead. However, in an insecure environment, additional cryptography (e.g., link encryption) may be required to keep capabilities from being disclosed accidentally to wiretappers on the network.

**Standard Operations**

Although many operations on objects depend on the object type, there are some operations that apply to most objects. These are listed in Fig. 7-5. Some of these require certain rights bits to be set, but others can be done by anyone who can present a server with a valid capability for one of its objects.

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>Perform a garbage collection cycle</td>
</tr>
<tr>
<td>Copy</td>
<td>Duplicate the object and return a capability for the copy</td>
</tr>
<tr>
<td>Destroy</td>
<td>Destroy the object and reclaim its storage</td>
</tr>
<tr>
<td>Getparams</td>
<td>Get parameters associated with the server</td>
</tr>
<tr>
<td>Info</td>
<td>Get an ASCII string briefly describing the object</td>
</tr>
<tr>
<td>Restrict</td>
<td>Produce a new, restricted capability for the object</td>
</tr>
<tr>
<td>Setparams</td>
<td>Set parameters associated with the server</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>Status</td>
<td>Get current status information from the server</td>
</tr>
<tr>
<td>Touch</td>
<td>Pretend the object was just used</td>
</tr>
</tbody>
</table>

**Fig. 7-5.** The standard operations valid on most objects.

It is possible to create an object in Amoeba and then lose the capability, so some mechanism is needed to get rid of old objects that are no longer accessible. The way that has been chosen is to have servers run a garbage collector periodically, removing all objects that have not been used in garbage collection cycles. The AGE call starts a new garbage collection cycle. The TOUCH call tells the server that the object touched is still in use. When objects are entered into the directory server, they are touched periodically, to keep the garbage collector at bay. Rights for some of the standard operations, such as AGE, are normally present only in capabilities owned by the system administrator.

The COPY operation is a shortcut that makes it possible to duplicate an object without actually transferring it. Without this operation, copying a file would require sending it over the network twice: from the server to the client and then back again. COPY can also fetch remote objects or send objects to remote machines.

The DESTROY operation deletes the object. It always needs the appropriate right, for obvious reasons.

The GETPARAMS and SETPARAMS calls normally deal with the server as a whole rather than with a particular object. They allow the system administrator to read and write parameters that control server operation. For example, the algorithm used to choose processors can be selected using this mechanism.

The INFO and STATUS calls return status information. The former returns a short ASCII string describing the object briefly. The information in the string is server dependent, but in general, it indicates the type of object and tells something useful about it (e.g., for files, it tells the size). The latter gets information about the server as a whole, for example, how much free memory it has. This information helps the system administrator monitor the system better.

The RESTRICT call generates a new capability for the object, with a subset of the current rights, as described above.

**PROCESS MANAGEMENT IN AMOEBA**

A process in Amoeba is basically an address space and a collection of threads that run in it. A process with one thread is roughly analogous to a UNIX or MS-DOS process in terms of how it behaves and what it can do. In this section we will explain how processes and threads work, and how they are implemented.

**Processes**

A process is an object in Amoeba. When a process is created, the parent process is given a capability for the child process, just as with any other newly created object. Using this capability, the child can be suspended, restarted, signaled, or destroyed.

Process creation in Amoeba is different from UNIX. The UNIX model of creating a child process by cloning the parent is inappropriate in a distributed system due to the considerable overhead of first creating a copy somewhere (FORK) and almost immediately afterward replacing the copy with a new program (EXEC). Instead, in Amoeba it is possible to create a new process on a specific processor with the intended memory image starting right at the beginning. In this one respect, process creation in Amoeba is similar to MS-DOS. However, in contrast to MS-DOS, a process can continue executing in parallel with its child, and thus can create an arbitrary number of additional children. The children can create their own children, leading to a tree of processes.
Process management is handled at three different levels in Amoeba. At the lowest level are the process servers, which are kernel threads running on every machine. To create a process on a given machine, another process does an RPC with that machine's process server, providing it with the necessary information.

At the next level up we have a set of library procedures that provide a more convenient interface for user programs. Several flavors are provided. They do their job by calling the low-level interface procedures.

Finally, the simplest way to create a process is to use the run server, which does most of the work of determining where to run the new process. We will discuss the run server later in this chapter.

Some of the process management calls use a data structure called a **process descriptor** to provide information about the process to be run. One field in the process descriptor (see Fig. 7-6) tells which CPU architecture the process can run on. In heterogeneous systems, this field is essential to make sure that 386 binaries are not run on SPARCs, and so on.

Another field contains the process' owner's capability. When the process terminates or is stunned (see below), RPCs will be done using this capability to report the event. It also contains descriptors for all the process' segments, which collectively define its address space, as well as descriptors for all its threads.

Finally, the process descriptor also contains a descriptor for each thread in the process. The content of a thread descriptor is architecture dependent, but as a bare minimum, it contains the thread's program counter and stack pointer. It may also contain additional information necessary to run the thread, including other registers, the thread's state, and various flags. Brand new processes contain only one thread in their process descriptors, but stunned processes may have created additional threads before being stunned.

The low-level process interface consists of about a half-dozen library procedures. Only three of these will concern us here. The first, **exec**, is the most important. It has two input parameters, the capability for a process server and a process descriptor. Its function is to do an RPC with the specified process server asking it to run the process. If the call is successful, a capability for the new process is returned to the caller for use in controlling the process later.
A second important library procedure is `getload`. It returns information about the CPU speed, current load, and amount of memory free at the moment. It is used by the run server to determine the best place to execute a new process. A third major library procedure is `stun`. A process' parent can suspend it by stunning it. More commonly, the parent can give the process' capability to a debugger, which can stun it and later restart it for interactive debugging purposes. Two kinds of stuns are supported: normal and emergency. They differ with respect to what happens if the process is blocked on one or more RPCs at the time it is stunned. With a normal stun, the process sends a message to the server it is currently waiting for, saying, in effect: "I have been stunned. Finish your work instantly and send me a reply." If the server is also blocked, waiting for another server, the message is propagated further, all the way down the line to the end. The server at the end of the line is expected to reply immediately with a special error message. In this way, all the pending RPCs are terminated almost immediately in a clean way, with all of the servers finishing properly. The nesting structure is not violated, and no "long jumps" are needed.

An emergency stun stops the process instantly and does not send any messages to servers that are currently working for the stunned process. The computations being done by the servers become orphans. When the servers finally finish and send replies, these replies are ultimately discarded.

The high-level process interface does not require a fully formed process descriptor. One of the calls, `newproc`, takes as its first three parameters, the name of the binary file and pointers to the argument and environment arrays, similar to UNIX. Additional parameters provide more detailed control of the initial state.

**Threads**

Amoeba supports a simple threads model. When a process starts up, it has one thread. During execution, the process can create additional threads, and existing threads can terminate. The
number of threads is therefore completely dynamic. When a new thread is created, the parameters

to the call specify the procedure to run and the size of the initial stack.

Although all threads in a process share the same program text and global data, each thread

has its own stack, its own stack pointer, and its own copy of the machine registers. In addition, if a

thread wants to create and use variables that are global to all its procedures but invisible to other

threads, library procedures are provided for that purpose. Such variables are called glocal. One

library procedure allocates a block of glocal memory of whatever size is requested, and returns a

pointer to it. Blocks of glocal memory are referred to by integers. A system call is available for a

thread to acquire its glocal pointer.

Three methods are provided for threads to synchronize: signals, mutexes, and semaphores. Signals

are asynchronous interrupts sent from one thread to another thread in the same process. They are

conceptually similar to UNIX signals, except that they are between threads rather than between

processes. Signals can be raised, caught, or ignored. Asynchronous interrupts between

processes use the stun mechanism.

The second form of interthread communication is the mutex. A mutex is like a binary

semaphore. It can be in one of two states, locked or unlocked. Trying to lock an unlocked mutex

causes it to become locked. The calling thread continues. Trying to lock a mutex that is already

locked causes the calling thread to block until another thread unlocks the mutex. If more than one

thread is waiting on a mutex, when the mutex is unlocked, exactly one thread is released. In

addition to the calls to lock and unlock mutexes, there is also one that tries to lock a mutex, but if it

is unable to do so within a specified interval, times out and returns an error code. Mutexes are fair

and respect thread priorities.

The third way that threads can communicate is by counting semaphores.

These are slower than mutexes, but there are times when they are needed. They work in the

usual way, except that here too an additional call is provided to allow a DOWN operation to time

out if it is unable to succeed within a specified interval.

All threads are managed by the kernel. The advantage of this design is that when a thread

does an RPC, the kernel can block that thread and schedule another one in the same process if one

is ready. Thread scheduling is done using priorities, with kernel threads getting higher priority than

user threads. Thread scheduling can be set up to be either pre-emptive or run-to-completion (i.e.,

threads continue to run until they block), as the process wishes.

MEMORY MANAGEMENT IN AMOEBA

Amoeba has an extremely simple memory model. A process can have any number of

segments it wants to have, and they can be located wherever it wants in the process' virtual

address space. Segments are not swapped or paged, so a process must be entirely memory

resident to run. Furthermore, although the hardware MMU is used, each segment is stored

contiguously in memory.

Although this design is perhaps somewhat unusual these days, it was done for three reasons: performance,
simplicity, and economics. Having a process entirely in memory all the time makes

RPC go faster. When a large block of data must be sent, the system knows that all of the data are

contiguous not only in virtual memory, but also in physical memory. This knowledge saves having
to check if all the pages containing the buffer happen to be around at the moment, and eliminates
having to wait for them if they are not. Similarly, on input, the buffer is always in memory, so the
incoming data can be placed there simply and without page faults. This design has allowed Amoeba
to achieve extremely high transfer rates for large RPCs.

The second reason for the design is simplicity. Not having paging or swapping makes the

system considerably simpler and makes the kernel smaller and more manageable. However, it is
the third reason that makes the first two feasible. Memory is becoming so cheap that within a few
years, all Amoeba machines will probably have tens of megabytes of it. Such large memories will
substantially reduce the need for paging and swapping, namely, to fit large programs into small machines.

**Segments**

Processes have several calls available to them for managing segments. Most important among these is the ability to create, destroy, read, and write segments. When a segment is created, the caller gets back a capability for it. This capability is used for reading and writing the segment and for all the other calls involving the segment.

When a segment is created it is given an initial size. This size may change during process execution. The segment may also be given an initial value, either from another segment or from a file.

Because segments can be read and written, it is possible to use them to construct a main memory file server. To start, the server creates a segment as large as it can. It can determine the maximum size by asking the kernel. This segment will be used as a simulated disk. The server then formats the segment as a file system, putting in whatever data structures it needs to keep track of files. After that, it is open for business, accepting requests from clients.

**Mapped Segments**

Virtual address spaces in Amoeba are constructed from segments. When a process is started, it must have at least one segment. However, once it is running, a process can create additional segments and map them into its address space at any unused virtual address. Figure 7-7 shows a process with three memory segments currently mapped in.

A process can also unmap segments. Furthermore, a process can specify a range of virtual addresses and request that the range be unmapped, after which those addresses are no longer legal. When a segment or a range of addresses is unmapped, a capability is returned so the segment may still be accessed, or even mapped back in again later, possibly at a different virtual address.

A segment may be mapped into the address space of two or more processes at the same time. This allows processes to operate on shared memory. However, usually it is better to create a single process with multiple threads when shared memory is needed. The main reason for having distinct processes is better protection, but if the two processes are sharing memory, protection is generally not desired.

**COMMUNICATION IN AMOEBA**

Amoeba supports two forms of communication: RPC, using point-to-point message passing, and group communication. At the lowest level, an RPC consists of a request message followed by a reply message. Group communication uses hardware broadcasting or multicasting if it is available; otherwise, the kernel transparently simulates it with individual messages. In this section we will describe both Amoeba RPC and Amoeba group communication and then discuss the underlying FLIP protocol that is used to support them.
Remote Procedure Call

Normal point-to-point communication in Amoeba consists of a client sending a message to a server followed by the server sending a reply back to the client. It is not possible for a client just to send a message and then go do something else except by bypassing the RPC interface, which is done only under very special circumstances. The RPC primitive that sends the request automatically blocks the caller until the reply comes back, thus forcing a certain amount of structure on programs. Separate send and receive primitives can be thought of as the distributed system's answer to the goto statement: parallel spaghetti programming. They should be avoided by user programs and used only by language runtime systems that have unusual communication requirements.

Each standard server defines a procedural interface that clients can call. These library routines are stubs that pack the parameters into messages and invoke the kernel primitives to send the message. During message transmission, the stub, and hence the calling thread, are blocked. When the reply comes back, the stub returns the status and results to the client. Although the kernel-level primitives are actually related to the message passing, the use of stubs makes this mechanism look like RPC to the programmer, so we will refer to the basic communication primitives as RPC, rather than the slightly more precise "request/reply message exchange."

In order for a client thread to do an RPC with a server thread, the client must know the server's address. Addressing is done by allowing any thread to choose a random 48-bit number, called a port, to be used as the address for messages sent to it. Different threads in a process may use different ports if they so desire. All messages are addressed from a sender to a destination port. A port is nothing more than a kind of logical thread address. There is no data structure and no storage associated with a port. It is similar to an IP address or an Ethernet address in that respect, except that it is not tied to any particular physical location. The first field in each capability gives the port of the server that manages the object (see Fig. 7-3).

Group Communication in Amoeba

RPC is not the only form of communication supported by Amoeba. It also supports group communication. A group in Amoeba consists of one or more processes that are cooperating to carry
out some task or provide some service. Processes can be members of several groups at the same time. Groups are closed, meaning that only members can broadcast to the group. The usual way for a client to access a service provided by a group is to do an RPC with one of its members. That member then uses group communication within the group, if necessary, to determine who will do what.

This design was chosen to provide a greater degree of transparency than an open group structure would have. The idea behind it is that clients normally use RPC to talk to individual servers, so they should use RPC to talk to groups as well. The alternative — open groups and using RPC to talk to single servers but using group communication to talk to group servers — is much less transparent. (Using group communication for everything would eliminate the many advantages of RPC that we have discussed earlier.) Once it has been determined that clients outside a group will use RPC to talk to the group (actually, to talk to one process in the group), the need for open groups vanishes, so closed groups, which are easier to implement, are adequate.

The operations available for group communication in Amoeba are listed in Fig. 7-10. CreateGroup creates a new group and returns a group identifier used in the other calls to identify which group is meant. The parameters specify various sizes and how much fault tolerance is required (how many dead members the group must be able to withstand and continue to function correctly).

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateGroup</td>
<td>Create a new group and set its parameters</td>
</tr>
<tr>
<td>JoinGroup</td>
<td>Make the caller a member of a group</td>
</tr>
<tr>
<td>LeaveGroup</td>
<td>Remove the caller from a group</td>
</tr>
<tr>
<td>SendToGroup</td>
<td>Reliably send a message to all members of a group</td>
</tr>
<tr>
<td>ReceiveFromGroup</td>
<td>Block until a message arrives from a group</td>
</tr>
<tr>
<td>ResetGroup</td>
<td>Initiate recovery after a process crash</td>
</tr>
</tbody>
</table>

Fig. 7-10. Amoeba group communication primitives.

JoinGroup and LeaveGroup allow processes to enter and exit from existing groups. One of the parameters of JoinGroup is a small message that is sent to all group members to announce the presence of a newcomer. Similarly, one of the parameters of LeaveGroup is another small message sent to all members to say goodbye and wish them good luck in their future activities. The point of the little messages is to make it possible for all members to know who their comrades are, in case they are interested, for example, to reconstruct the group if some members crash. When the last member of a group calls LeaveGroup, it turns out the lights and the group is destroyed.

SendToGroup atomically broadcasts a message to all members of a specified group, in spite of lost messages, finite buffers, and processor crashes. Amoeba supports global time ordering, so if two processes call SendToGroup simultaneously, the system ensures that all group members will receive the messages in the same order. This is guaranteed; programmers can count on it. If the two calls are exactly simultaneous, the first one to get its packet onto the LAN successfully is declared to be first. In terms of the semantics discussed in Chap. 6, this model corresponds to sequential consistency, not strict consistency.

ReceiveFromGroup tries to get a message from a group specified by one of its parameter. If no message is available (that is, currently buffered by the kernel), the caller blocks until one is available. If a message has already arrived, the caller gets the message with no delay. The protocol
ensures that in the absence of processor crashes, no messages are irretrievably lost. The protocol can also be made to tolerate crashes, at the cost of additional overhead, as discussed later.

The final call, ResetGroup, is used to recover from crashes. It specifies how many members the new group must have as a minimum. If the kernel is able to establish contact with the requisite number of processes and rebuild the group, it returns the size of the new group. Otherwise, it fails. In this case, recovery is up to the user program.

8
Case Study 2: Mach

Our second example of a modern, microkernel-based operating system is Mach. We will start out by looking at its history and how it has evolved from earlier systems. Then we will examine in some detail the microkernel itself, focusing on processes and threads, memory management, and communication. Finally, we will discuss UNIX emulation. More information about Mach can be found in (Accetta et al., 1986; Baron et al., 1985; Black et al., 1992; Boykin et al., 1993; Draves et al., 1991; Rashid, 1986a; Rashid, 1986b; and Sansom et al., 1986).

INTRODUCTION TO MACH

In this section we will give a brief introduction to Mach. We will start with its history and goals. Then we will describe the main concepts of the Mach microkernel and the principal server that runs on the microkernel.

History of Mach

Mach's earliest roots go back to a system called RIG (Rochester Intelligent Gateway), which began at the university of Rochester in 1975 (Ball et al., 1976). RIG was written for a 16-bit Data General minicomputer called the Eclipse. Its main research goal was to demonstrate that operating systems could be structured in a modular way, as a collection of processes that communicated by message passing, including over a network. The system was designed and built, and indeed showed that such an operating system could be constructed.

When one of its designers, Richard Rashid, left the University of Rochester and moved to Carnegie-Mellon University in 1979, he wanted to continue developing message-passing operating systems but on more modern hardware. Various machines were considered. The machine selected was the PERQ, an early engineering workstation, with a bitmapped screen, mouse, and network connection. It was also microprogrammable. The new operating system for the PERQ was called Accent. It improved on RIG by adding protection, the ability to operate transparently over the network, 32-bit virtual memory, and other features. An initial version was up and running in 1981.

By 1984 Accent was being used on 150 PERQs but it was clearly losing out to UNIX. This observation led Rashid to begin a third-generation operating systems project called Mach. By making Mach compatible with UNIX, he hoped to be able to use the large volume of UNIX software becoming available. In addition, Mach had many other improvements over Accent, including threads, a better interprocess communication mechanism, multiprocessor support, and a highly imaginative virtual memory system.

Around this time, DARPA, the U.S. Department of Defense's Advanced Research Projects Agency, was hunting around for an operating system that supported multiprocessors as part of its Strategic Computing Initiative. CMU was selected, and with substantial DARPA funding, Mach was developed further. Initially, Mach consisted of a modified version of 4.1 BSD with additional features inserted for communication and memory management. As 4.2 BSD and 4.3 BSD became available, the Mach code was combined with them to give updated versions. Although this approach led to a large kernel, it did guarantee absolute compatibility with Berkeley UNIX, an important goal for DARPA.

The first version of Mach was released in 1986 for the VAX 11/784, a four-CPU multiprocessor. Shortly thereafter, ports to the IBM PC/RT and Sun 3 were done. By 1987, Mach was also running
on the Encore and Sequent multiprocessors. Although Mach had networking facilities, at this time it was conceived of primarily as a single machine or multiprocessor system rather than as a transparent distributed operating system for a collection of machines on a LAN.

Shortly thereafter, the Open Software Foundation, a consortium of computer vendors led by IBM, DEC, and Hewlett Packard was formed in an attempt to wrest control of UNIX from its owner, AT&T, which was then working closely with Sun Microsystems to develop System V Release 4. The OSF members feared that this alliance would give Sun a competitive advantage over them. After some missteps, OSF chose Mach 2.5 as the basis for its first operating system, OSF/1. Although Mach 2.5 and OSF/1 contained large amounts of Berkeley and AT&T code, the hope was that OSF would at least be able to control the direction in which UNIX was going.

As of 1988, the Mach 2.5 kernel was large and monolithic, due to the presence of a large amount of Berkeley UNIX code in the kernel. In 1988, CMU removed all the Berkeley code from the kernel and put it in user space. What remained was a microkernel consisting of pure Mach. In this chapter, we will focus on the Mach 3 microkernel and one user-level operating system emulator, for BSD UNIX. One difficulty, however, is that Mach is under development, so any description is at best a snapshot. Fortunately, most of the basic ideas discussed in this chapter are relatively stable, but some of the details may change in time.

Goals of Mach

Mach has evolved considerably since its first incarnation as RIG. The goals of the project have also changed as time has gone on. The current primary goals can be summarized as follows:

1. Providing a base for building other operating systems (e.g., UNIX).
2. Supporting large sparse address spaces.
3. Allowing transparent access to network resources.
4. Exploiting parallelism in both the system and the applications.
5. Making Mach portable to a larger collection of machines.

These goals encompass both research and development. The idea is to explore multiprocessor and distributed systems while being able to emulate existing systems, such as UNIX, MS-DOS, and the Macintosh operating system.

Much of the initial work on Mach concentrated on single- and multiprocessor systems. At the time Mach was designed, few systems had support for multiprocessors. Even now, few multiprocessor systems other than Mach are machine independent.

The Mach Microkernel

The Mach microkernel has been built as a base upon which UNIX and other operating systems can be emulated. This emulation is done by a software layer that runs outside the kernel, as shown in Fig. 8-1. Each emulator consists of a part that is present in its application programs' address space, as well as one or more servers that run independently from the application programs. It should be noted that multiple emulators can be running simultaneously, so it is possible to run 4.3BSD, System V, and MS-DOS programs on the same machine at the same time.
Fig. 8-1. The abstract model for UNIX emulation using Mach.

Like other microkernels, the Mach kernel, provides process management, memory management, communication, and I/O services. Files, directories, and other traditional operating system functions are handled in user space. The idea behind the Mach kernel is to provide the necessary mechanisms for making the system work, but leaving the policy to user-level processes.

The kernel manages five principal abstractions:
1. Processes.
2. Threads.
3. Memory objects.
4. Ports.
5. Messages.

In addition, the kernel manages several other abstractions either related to these or less central to the model.

A process is basically an environment in which execution can take place. It has an address space holding the program text and data, and usually one or more stacks. The process is the basic unit for resource allocation. For example, a communication channel is always "owned" by a single process.

As an aside, for the most part we will stick with the traditional nomenclature throughout this chapter, even though this means deviating from the terminology used in the Mach papers (e.g., Mach, of course, has processes, but they are called "tasks."

A thread in Mach is an executable entity. It has a program counter and a set of registers associated with it. Each thread is part of exactly one process. A process with one thread is similar to a traditional (e.g., UNIX) process.

A concept that is unique to Mach is the memory object, a data structure that can be mapped into a process' address space. Memory objects occupy one or more pages and form the basis of the Mach virtual memory system. When a process attempts to reference a memory object that is not presently in physical main memory, it gets a page fault. As in all operating systems, the kernel catches the page fault. However, unlike other systems, the Mach kernel can send a message to a user-level server to fetch the missing page.

Interprocess communication in Mach is based on message passing. To receive messages, a user process asks the kernel to create a kind of protected mailbox, called a port, for it. The port is stored inside the kernel, and has the ability to queue an ordered list of messages. Queues are not fixed in size, but for flow control reasons, if more than \( n \) messages are queued on a port, a process attempting to send to it is suspended to give the port a chance to be emptied. The parameter \( n \) is settable per port.
A process can give the ability to send to (or receive from) one of its ports to another process. This permission takes the form of a capability, and includes not only a pointer to the port, but also a list of rights that the other process has with respect to the port (e.g., SEND right). Once this permission has been granted, the other process can send messages to the port, which the first process can then read. All communication in Mach uses this mechanism. Mach does not support a full capability mechanism; ports are the only objects for which capabilities exist.

**The Mach BSD UNIX Server**

As we described above, the Mach designers have modified Berkeley UNIX to run in user space, as an application program. This structure has a number of significant advantages over a monolithic kernel. First, by breaking the system up into a part that handles the resource management (the kernel) and a part that handles the system calls (the UNIX server), both pieces become simpler and easier to maintain. In a way, this split is somewhat reminiscent of the division of labor in IBM's mainframe operating system VM/370, in which the kernel simulates a collection of bare 370s, each of which runs a single-user operating system.

Second, by putting UNIX in user space, it can be made extremely machine independent, enhancing its portability to a wide variety of computers. All the machine dependencies can be removed from UNIX and hidden away inside the Mach kernel.

Third, as we mentioned earlier, multiple operating systems can run simultaneously. On a 386, for example, Mach can run a UNIX program and an MS-DOS program at the same time. Similarly, it is possible to test a new experimental operating system and run a production operating system at the same time.

Fourth, real-time operation can be added to the system because all the traditional obstacles that UNIX presents to real-time work, such as disabling interrupts in order to update critical tables are either eliminated altogether or moved into user space. The kernel can be carefully structured not to have this type of hindrance to real-time applications.

Finally, this arrangement can be used to provide better security between processes, if need be. If each process has its own version of UNIX, it is very difficult for one process to snoop on the other one's files.

**PROCESS MANAGEMENT IN MACH**

Process management in Mach deals with processes, threads, and scheduling. In this section we will look at each of these in turn.

**Processes**

A process in Mach consists primarily of an address space and a collection of threads that execute in that address space. Processes are passive. Execution is associated with the threads. Processes are used for collecting all the resources related to a group of cooperating threads into convenient containers.

Figure 8-2 illustrates a Mach process. In addition to an address space and threads, it has some ports and other properties. The ports shown in the figure all have special functions. The process port is used to communicate with the kernel. Many of the kernel services that a process can request are done by sending a message to the process port, rather than making a system call. This mechanism is used throughout Mach to reduce the actual system calls to a bare minimum. A small number of them will be discussed in this chapter, to give an idea of what they are like.

In general, the programmer is not even aware of whether or not a service requires a system call. All services, including both those accessed by system calls and those accessed by message passing, have stub procedures in the library. It is these procedures that are described in the manuals and called by application programs. The procedures are generated from a service definition by the MIG (Mach Interface Generator) compiler.
The **bootstrap port** is used for initialization when a process starts up. The very first process reads the bootstrap port to learn the names of kernel ports that provide essential services. UNIX processes also use it to communicate with the UNIX emulator.

![Fig. 8-2. A Mach process.](image)

The **exception port** is used to report exceptions caused by the process. Typical exceptions are division by zero and illegal instruction executed. The port tells the system where the exception message should be sent. Debuggers also use the exception port.

The **registered ports** are normally used to provide a way for the process to communicate with standard system servers. For example, the name server makes it possible to present a string and get back the corresponding port for certain basic servers.

Processes also have other properties. A process can be runnable or blocked, independent of the state of its threads. If a process is runnable, those threads that are also runnable can be scheduled and run. If a process is blocked, its threads may not run, no matter what state they are in.

The per-process items also include scheduling parameters. These include the ability to specify which processors the process' threads can run on. This feature is most useful on a multiprocessor system. For example, the process can use this power to force each thread to run on a different processor, or to force them all to run on the same processor, or anything in between. In addition, each process has a default priority that is settable. When a thread is created, the new thread is given this priority. It is also possible to change the priority of all the existing threads.

Emulation addresses can be set to tell the kernel where in the process' address space system call handlers are located. The kernel needs to know these addresses to handle UNIX system calls that need to be emulated. These are set once when the UNIX emulator is started up and are inherited by all of the emulator’s children (i.e., all the UNIX processes).

Finally, every process has statistics associated with it, including the amount of memory consumed, the run times of the threads, and so on. A process that is interested in this information can acquire it by sending a message to the process port.

It is also worth mentioning what a Mach process does not have. A process does not have a uid, gid, signal mask, root directory, working directory, or file descriptor array, all of which UNIX
processes do have. All of this information is managed by the emulation package, so the kernel knows nothing at all about it.

## Process Management Primitives

Mach provides a small number of primitives for managing processes. Most of these are done by sending messages to the kernel via the process port, rather than actual system calls. The most important of these calls are shown in Fig. 8-3. These, like all calls in Mach, have prefixes indicating the group they belong to, but we have omitted these here (and in subsequent tables) for the sake of brevity.

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>Create a new process, inheriting certain properties</td>
</tr>
<tr>
<td>Terminate</td>
<td>Kill a specified process</td>
</tr>
<tr>
<td>Suspend</td>
<td>Increment suspend counter</td>
</tr>
<tr>
<td>Resume</td>
<td>Decrement suspend counter. If it is 0, unblock the process</td>
</tr>
<tr>
<td>Priority</td>
<td>Set the priority for current or future threads</td>
</tr>
<tr>
<td>Assign</td>
<td>Tell which processor new threads should run on</td>
</tr>
<tr>
<td>Info</td>
<td>Return information about execution time, memory usage, etc.</td>
</tr>
<tr>
<td>Threads</td>
<td>Return a list of the process’ threads</td>
</tr>
</tbody>
</table>

**Fig. 8-3.** Selected process management calls in Mach.

The first two calls in Fig. 8-3 are for creating and destroying processes, respectively. The process creation call specifies a prototype process, not necessarily the caller. The child is a copy of the prototype, except that the call has a parameter that tells whether or not the child is to inherit the parent's address space. If it does not inherit the parent's address space, objects (e.g., text, initialized data, and a stack) can be mapped in later. Initially, the child has no threads. It does, however, automatically get a process port, a bootstrap port, and an exception port. Other ports are not inherited automatically since each port may have only one reader.

Processes can be suspended and resumed under program control. Each process has a counter, incremented by the `suspend` call and decremented by the `resume` call, that can block or unblock it. When the counter is 0, the process is able to run. When it is positive, it is suspended. Having a counter is more general than just having a bit, and helps avoid race conditions.

The `priority` and `assign` calls give the programmer control over how and where its threads run on multiprocessor systems. CPU scheduling is done using priorities, so the programmer has fine-grain control over which threads are most important and which are least important. The `assign` call makes it possible to control which thread runs on which CPU or group of CPUs.

The last two calls of Fig. 8-3 return information about the process. The former gives statistical information and the latter returns a list of all the threads.

### Threads

The active entities in Mach are the threads. They execute instructions and manipulate their registers and address spaces. Each thread belongs to exactly one process. A process cannot do anything unless it has one or more threads.
All the threads in a process share the address space and all the process-wide resources shown in Fig. 8-2. Nevertheless, threads also have private per-thread resources. One of these is the thread port, which is analogous to the process port. Each thread has its own thread port, which it uses to invoke thread-specific kernel services, such as exiting when the thread is finished. Since ports are process-wide resources, each thread has access to its siblings' ports, so each thread can control the others if need be.

Mach threads are managed by the kernel, that is, they are what are sometimes called heavyweight threads rather than lightweight threads (pure user space threads). Thread creation and destruction are done by the kernel and involve updating kernel data structures. They provide the basic mechanisms for handling multiple activities within a single address space. What the user does with these mechanisms is up to the user.

On a single CPU system, threads are timeshared, first one running, then another. On a multiprocessor, several threads can be active at the same time. This parallelism makes mutual exclusion, synchronization, and scheduling more important than they normally are, because performance now becomes a major issue, along with correctness. Since Mach is intended to run on multiprocessors, these issues have received special attention.

Like a process, a thread can be runnable or blocked. The mechanism is similar, too: a counter per thread that can be incremented and decremented. When it is zero, the thread is runnable. When it is positive, the thread must wait until another thread lowers it to zero. This mechanism allows threads to control each other's behavior.

A variety of primitives is provided. The basic kernel interface provides about two dozen thread primitives, many of them concerned with controlling scheduling in detail. On top of these primitives one can build various thread packages.

Mach's approach is the C threads package. This package is intended to make the kernel thread primitives available to users in a simple and convenient form. It does not have the full power that the kernel interface offers, but it is enough for the average garden-variety programmer. It has also been designed to be portable to a wide variety of operating systems and architectures.

The C threads package provides sixteen calls for direct thread manipulation. The most important ones are listed in Fig. 8-4. The first one, fork, creates a new thread in the same address space as the calling thread. It runs the procedure specified by a parameter rather than the parent's code. After the call, the parent thread continues to run in parallel with the child. The thread is started with a priority and on a processor determined by the process' scheduling parameters, as discussed above.

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fork</td>
<td>Create a new thread running the same code as the parent thread</td>
</tr>
<tr>
<td>Exit</td>
<td>Terminate the calling thread</td>
</tr>
<tr>
<td>Join</td>
<td>Suspend the caller until a specified thread exits</td>
</tr>
<tr>
<td>Detach</td>
<td>Announce that the thread will never be jointed (waited for)</td>
</tr>
<tr>
<td>Yield</td>
<td>Give up the CPU voluntarily</td>
</tr>
<tr>
<td>Self</td>
<td>Return the calling thread's identity to it</td>
</tr>
</tbody>
</table>

**Fig. 8-4.** The principal C threads calls for direct thread management.

When a thread has done its work, it calls exit. If the parent is interested in waiting for the thread to finish, it can call join to block itself until a specific child thread terminates. If the thread
has already terminated, the parent continues immediately. These three calls are roughly analogous to the FORK, EXIT, and WAITPID system calls in UNIX.

The fourth call, *detach*, does not exist in UNIX. It provides a way to announce that a particular thread will never be waited for. If that thread ever exits, its stack and other state information will be deleted immediately. Normally, this cleanup happens only after the parent has done a successful *join*. In a server, it might be desirable to start up a new thread to service each incoming request. When it has finished, the thread exits. Since there is no need for the initial thread to wait for it, the server thread should be detached.

The *yield* call is a hint to the scheduler that the thread has nothing useful to do at the moment, and is waiting for some event to happen before it can continue. An intelligent scheduler will take the hint and run another thread. In Mach, which normally schedules its threads preemptively, *yield* is only optimization. In systems that have nonpreemptive scheduling, it is essential that a thread that has no work to do release the CPU, to give other threads a chance to run.

Finally, *self* returns the caller's identity, analogous to GETPID in UNIX.

The remaining calls (not shown in the figure), allow threads to be named, allow the program to control the number of threads and the sizes of their stacks, and provide interfaces to the kernel threads and message-passing mechanism.

Synchronization is done using mutexes and condition variables. The mutex primitives are *lock*, *trylock*, and *unlock*. Primitives are also provided to allocate and free mutexes. Mutexes work like binary semaphores, providing mutual exclusion, but not conveying information.

The operations on condition variables are *signal*, *wait*, and *broadcast*, which are used to allow threads to block on a condition and later be awakened when another thread has caused that condition to occur.

### Implementation of C Threads in Mach

Various implementations of C threads are available on Mach. The original one did everything in user space inside a single process. This approach timeshared all the C threads over one kernel thread, as shown in Fig. 8-5(a). This approach can also be used on UNIX or any other system that provides no kernel support. The threads were run as coroutines, which means that they were scheduled nonpreemptively. A thread could keep the CPU as long as it wanted or was able to. For the producer-consumer problem, the producer would eventually fill the buffer and then block, giving the consumer a chance to run. For other applications, however, threads had to call *yield* from time to time to give other threads a chance.

The original implementation package suffers from a problem inherent to most user-space threads packages that have no kernel support. If one thread makes a blocking system call, such as reading from the terminal, the whole process is blocked. To avoid this situation, the programmer must avoid blocking system calls. In Berkeley UNIX, there is a call SELECT that can be used to tell whether any characters are pending, but the whole situation is quite messy.
Fig. 8-5. (a) All C threads use one kernel thread. (b) Each C thread has its own kernel thread. (c) Each C thread has its own single-threaded process. (d) Arbitrary mapping of user threads to kernel threads.

A second implementation is to use one Mach thread per C thread, as shown in Fig. 8-5(b). These threads are scheduled preemptively. Furthermore, on a multiprocessor, they may actually run in parallel, on different CPUs. In fact, it is also possible to multiplex $m$ user threads on $n$ kernel threads, although the most common case is $m = n$.

A third implementation package has one thread per process, as shown in Fig. 8-5(c). The processes are set up so that their address spaces all map onto the same physical memory, allowing sharing in the same way as in the previous implementations. This implementation is only used when specialized virtual memory usage is required. The method has the drawback that ports, UNIX files, and other per-process resources cannot be shared, limiting its value appreciably.

The fourth package allows an arbitrary number of user threads to be mapped onto an arbitrary number of kernel threads, as shown in Fig. 8-5(d).

The main practical value of the first approach is that because there is no true parallelism, successive runs give reproducible results, allowing easier debugging. The second approach has the advantage of simplicity and was used for a long time. The third one is not normally used. The fourth one, although the most complicated, gives the greatest flexibility and is the one normally used at present.

**Scheduling**

Mach scheduling has been heavily influenced by its goal of running on multiprocessors. Since a single-processor system is effectively a special case of a multiprocessor (with only one CPU), our discussion will focus on scheduling in multiprocessor systems. For more information, see (Black, 1990).

The CPUs in a multiprocessor can be assigned to **processor sets** by software. Each CPU belongs to exactly one processor set. Threads can also be assigned to processor sets by software. Thus each processor set has a collection of CPUs at its disposal and a collection of threads that need computing power. The job of the scheduling algorithm is to assign threads to CPUs in a fair and efficient way. For purposes of scheduling, each processor set is a closed world, with its own resources and its own customers, independent of all the other processor sets.

This mechanism gives processes a large amount of control over their threads. A process can assign an important thread to a processor set with one CPU and no other threads, thus ensuring that the thread runs all the time. It can also dynamically reassign threads to processor sets as the work proceeds, keeping the load balanced. While the average compiler is not likely to use this facility, a database management system or a real-time system might well use it.

Thread scheduling in Mach is based on priorities. Priorities are integers from 0 to some maximum (usually 31 or 127), with 0 being the highest priority and 31 or 127 being the lowest priority. This priority reversal comes from UNIX. Each thread has three priorities assigned to it. The first priority is a base priority, which the thread can set itself, within certain limits. The second
priority is the lowest numerical value that the thread may set its base priority to. Since using a higher value gives worse service, a thread will normally set its value to the lowest value it is permitted, unless it is trying intentionally to defer to other threads. The third priority is the current priority, used for scheduling purposes. It is computed by the kernel by adding to the base priority a function based on the thread's recent CPU usage.

Mach threads are visible to the kernel, at least when the model of Fig. 8-5(b) is used. Each thread competes for CPU cycles with all other threads, without regard to which thread is in which process. When making scheduling decisions, the kernel does not take into account which thread belongs to which process.

Associated with each processor set is an array of run queues, as shown in Fig. 8-6. The array has 32 queues, corresponding to threads currently at priorities 0 through 31. When a thread at priority \( n \) becomes runnable, it is put at the end of queue \( n \). A thread that is not runnable is not present on any run queue.

Each run queue has three variables attached to it. The first is a mutex that is used to lock the data structure. It is used to make sure that only one CPU at a time is manipulating the queues. The second variable is the count of the number of threads on all the queues combined. If this count becomes 0, there is no work to do. The third variable is a hint as to where to find the highest-priority thread. It is guaranteed that no thread is at a higher priority, but the highest may be at a lower priority. This hint allows the search for the highest-priority thread to avoid the empty queues at the top.

![Fig. 8-6. The global run queues for a system with two processor sets.](image)

In addition to the global run queues shown in Fig. 8-6, each CPU has its own local run queue. Each local run queue holds those threads that are permanently bound to that CPU, for example, because they are device drivers for I/O devices attached to that CPU. These threads can run on only one CPU, so putting them on the global run queue is incorrect (because the "wrong" CPU might choose them).

We can now describe the basic scheduling algorithm. When a thread blocks, exits, or uses up its quantum, the CPU it is running on first looks on its local run queue to see if there are any active
threads. This check merely requires inspecting the count variable associated with the local run queue. If it is nonzero, the CPU begins searching the queue for the highest-priority thread, starting at the queue specified by the hint. If the local run queue is empty, the same algorithm is applied to the global run queue, the only difference being that the global run queue must be locked before it can be searched. If there are no threads to run on either queue, a special idle thread is run until some thread becomes ready.

If a runnable thread is found, it is scheduled and run for one quantum. At the end of the quantum, both the local and global run queues are checked to see if any other threads at its priority or higher are runnable, with the understanding that all threads on the local run queue have higher priority than all threads on the global run queue. If a suitable candidate is found, a thread switch occurs. If not, the thread is run for another quantum. Threads may also be preempted. On multiprocessors, the length of the quantum is variable, depending on the number of threads that are runnable. The more runnable threads and the fewer CPUs here are, the shorter the quantum. This algorithm gives good response time to short requests, even on heavily loaded systems, but provides high efficiency (i.e., long quanta) on lightly loaded systems.

On every clock tick, the CPU increments the priority counter of the currently running thread by a small amount. As the value goes up, the priority goes down and the thread will eventually move to a higher-numbered (i.e., lower-priority) queue. The priority counters are lowered by the passage of time.

For some applications, a large number of threads may be working together to solve a single problem, and it may be important to control the scheduling in detail. Mach provides a hook to give threads some additional control over their scheduling (in addition to the processor sets and priorities). The hook is a system call that allows a thread to lower its priority to the absolute minimum for a specified number of seconds. Doing so gives other threads a chance to run. When the time interval is over, the priority is restored to its previous value.

This system call has another interesting property: it can name its successor if it wants to. For example, after sending a message to another thread, the sending thread can give up the CPU and request that the receiving thread be allowed to run next. This mechanism, called **handoff scheduling**, bypasses the run queues entirely. Used wisely, it can enhance performance. The kernel also uses it in some circumstances, as an optimization.

Mach can be configured to do affinity scheduling, but generally this option is off. When it is on, the kernel schedules a thread on the CPU it last ran on, in hopes that part of its address space is still in that CPU’s cache. Affinity scheduling is only applicable to multiprocessors.

Finally, several other scheduling algorithms are supported in some versions, including algorithms useful for real-time applications.

**MEMORY MANAGEMENT IN MACH**

Mach has a powerful, elaborate, and highly flexible memory management system based on paging, including features found in few other operating systems. In particular, it separates the machine-independent parts of the memory management system from the machine-dependent parts in an extremely clear and unusual way. This separation makes the memory management far more portable than in other systems. In addition, the memory management system interacts closely with the communication system, which we will discuss in the following section.

The aspect of Mach’s memory management that sets it apart from all others is that the code is split into three parts. The first part is the **pmap** module, which runs in the kernel and is concerned with managing the MMU. It sets up the MMU registers and hardware page tables, and catches all page faults. This code depends on the MMU architecture and must be rewritten for each new MMU Mach has to support. The second part, the machine-independent kernel code, is concerned with processing page faults, managing address maps, and replacing pages.
The third part of the memory management code runs as a user process called a memory manager or sometimes an external pager. It handles the logical (as opposed to physical) part of the memory management system, primarily management of the backing store (disk). For example, keeping track of which virtual pages are in use, which are in main memory, and where pages are kept on disk when they are not in main memory are all done by the memory manager.

The kernel and the memory manager communicate through a well-defined protocol, making it possible for users to write their own memory managers. This division of labor gives users the ability to implement special-purpose paging systems in order to write systems with special requirements. It also has the potential for making the kernel smaller and simpler by moving a large section of the code out into user space. On the other hand, it has the potential for making it more complicated, since the kernel must protect itself from buggy or malicious memory managers, and with two active entities involved in handling memory, there is now the danger of race conditions.

Virtual Memory

The conceptual model of memory that Mach user processes see is a large, linear virtual address space. For most 32-bit CPU chips, the user address space runs from address 0 to address $2^{31} - 1$ because the kernel uses the top half for its own purposes. The address space is supported by paging. Since paging was designed to give the illusion of ordinary memory, but more of it than there really is, in principle there should be nothing else to say about how Mach manages virtual address space.

In reality, there is a great deal more to say. Mach provides a great deal of fine-grained control over how the virtual pages are used (for processes that are interested in that). To start with, the address space can be used in a sparse way. For example, a process might have dozens of sections of the virtual address space in use, each many megabytes from its nearest neighbor, with large holes of unused addresses between the sections.

Theoretically, any virtual address space can be used this way, so the ability to use a number of widely scattered sections is not really a property of the virtual address space architecture. In other words, any 32-bit machine should allow a process to have a 50K section of data spaced every 100 megabytes, from 0 to the 4-gigabyte limit. However, in many implementations, a linear page table from 0 to the highest used page is kept in kernel memory. On a machine with a 1K page size, this configuration requires 4 million page table entries, making it expensive, if not impossible. Even with a multilevel page table, such sparse usage is inconvenient at best. With Mach, the intention is to fully support sparse address spaces.

To determine which virtual addresses are in use and which are not, Mach provides a way to allocate and deallocate sections of virtual address space, called regions. The allocation call can specify a base address and a size, in which case the indicated region is allocated, or it can just specify a size, in which case the system finds a suitable address range and returns its base address. A virtual address is valid only if it falls in an allocated region. An attempt to use an address between allocated regions results in a trap, which, however, can be caught by the process if it so desires.
Fig. 8-7. An address space with allocated regions, mapped objects, and unused addresses.

A key concept relating to the use of virtual address space is the **memory object**. A memory object can be a page or a set of pages, but it can also be a file or other, more specialized data structure. A memory object can be mapped into an unused portion of the virtual address space, forming a new region, as shown in Fig. 8-7. When a file is mapped into the virtual address space, it can be read and written by normal machine instructions. Mapped files are paged in the usual way. When a process terminates, its mapped files automatically appear back in the file system, complete with all the changes that were made to them when they were mapped in. It is also possible to unmap files or other memory objects explicitly, freeing their virtual addresses and making them available for subsequent allocation or mapping.

As an aside, file mapping is not the only way to access files. They can also be read the conventional way. However, even then, the library may map the files behind the user's back rather than reading them using the I/O system. Doing so allows the file pages to use the virtual memory system, rather than using dedicated buffers elsewhere in the system.

Mach supports a number of calls for manipulating virtual address spaces. The main ones are listed in Fig. 8-8. None are true system calls. Instead, they all write messages to the caller's process port.

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocate</td>
<td>Make a region of virtual address space usable</td>
</tr>
<tr>
<td>Deallocate</td>
<td>Invalidate a region of virtual address space</td>
</tr>
<tr>
<td>Map</td>
<td>Map a memory object into the virtual address space</td>
</tr>
<tr>
<td>Copy</td>
<td>Make a copy of a region at another virtual address</td>
</tr>
<tr>
<td>Inherit</td>
<td>Set the inheritance attribute for a region</td>
</tr>
<tr>
<td>Read</td>
<td>Read data from another process’ virtual address space</td>
</tr>
</tbody>
</table>
Write data to another process’ virtual address space

**Fig. 8-8.** Selected Mach calls for managing virtual memory.

The first call, *allocate*, makes a region of virtual address space usable. A process may inherit allocated virtual address space and it may allocate more, but any attempt to reference an unallocated address will fail. The second call, *deallocate*, invalidates a region (i.e., removes it from the memory map), thus making it possible to allocate it again or map something into it, using the *map* call.

The *copy* call copies a memory object onto a new region. The original remains unchanged. In this way, a single memory object can appear multiple times in the address space. Conceptually, calling *copy* is no different than having the object copied by a programmed loop. However *copy* is implemented in an optimized way, using shared pages, to avoid physical copying.

The *inherit* call affects the way that regions are inherited when new processes are created. The address space can be set up so that some regions are inherited and others are not. It will be discussed in the next section.

The *read* and *write* calls allow a thread to access virtual memory belonging to another process. These calls require the caller to have possession of the process port belonging to the remote process, something that process can pass to its friends if it wants to.

In addition to the calls listed in Fig. 8-8, a few other calls also exist. These calls are concerned primarily with getting and setting attributes, protection modes, and various kinds of statistical information.

**Memory Sharing**

Sharing plays an important role in Mach. No special mechanism is needed for the threads in a process to share objects: they all see the same address space automatically. If one of them has access to a piece of data, they all do. More interesting is the possibility of two or more processes sharing the same memory objects, or just sharing data pages, for that matter. Sometimes sharing is important on single CPU systems. For example, in the classical producer-consumer problem, it may be desirable to have the producer and consumer be different processes, yet share a common buffer so that the producer can put data into the buffer and the consumer can take data out of it.

On multiprocessor systems, sharing of objects between two or more processes is frequently even more important. In many cases, a single problem is being solved by a collection of cooperating processes running in parallel on different CPUs (as opposed to being timeshared on a single CPU). These processes may need access to buffers, tables, or other data structures continuously, in order to do their work. It is essential that the operating system allow this sharing to take place. Early versions of UNIX did not have this ability, for example, although it was added later.

Consider, for example, a system that analyzes digitized satellite images of the earth in real time, as they are transmitted to the ground. Such analysis is time consuming, and the same picture has to be examined for use in weather forecasting, predicting crop harvests, and tracking pollution. As each picture is received, it is stored as a file.

A multiprocessor is available to do the analysis. Since the meteorological, agricultural, and environmental programs are all quite different, and were written by different people, it is not reasonable to make them threads of the same process. Instead, each is a separate process, and each maps the current photograph into its address space, as shown in Fig. 8-9. Note that the file containing the photograph may be mapped in at a different virtual address in each process. Although each page is present in memory only once, it may appear in each process’ page map at a different place. In this manner, all three processes can work on the same file at the same time in a convenient way.
Another important use of sharing is process creation. As in UNIX, in Mach the basic way for a new process to be created is as a copy of an existing process. In UNIX, a copy is always a clone of the process executing the fork system call, whereas in Mach the child can be a clone of a different process (the prototype). Either way, the child is a copy of some other process.

One way to create the child is to copy all the pages needed and map the copies into the child's address space. Although this method is valid, it is unnecessarily expensive. The program text is normally read-only, so it cannot change, and parts of the data may also be read-only. There is no reason to copy read-only pages, since mapping them into both processes will do the job. Writable pages cannot always be shared because the semantics of process creation (at least in UNIX) say that although at the moment of creation the parent and child are identical, subsequent changes to either one are not visible in the other's address space.

In addition, some regions (e.g., certain mapped files) may not be needed in the child. Why go to a lot of trouble to arrange for them to be present in the child if they are not needed there?

To achieve these various goals, Mach allows processes to assign an inheritance attribute to each region in its address space. Different regions may have different attributes. Three values are provided:

1. The region is unused in the child process.
2. The region is shared between the prototype process and the child.
3. The region in the child process is a copy of the prototype.

If a region has the first value, the corresponding region in the child is unallocated. References to it are treated as references to any other unallocated memory — they generate traps. The child is free to allocate the region for its own purposes or to map a memory object there.
The second option is true sharing. The pages of the region are present in both the prototype's address space and the child's. Changes made by either one are visible to the other. This choice is not used for implementing the fork system call in UNIX, but is frequently useful for other purposes.

The third possibility is to copy all the pages in the region and map the copies into the child's address space. FORK uses this option. Actually, Mach does not really copy the pages but uses a clever trick called copy-on-write instead. It places all the necessary pages in the child's virtual memory map, but marks them all read-only, as illustrated in Fig. 8-10. As long as the child makes only read references to these pages, everything works fine.

However, if the child attempts to write on any page, a protection fault occurs. The operating system then makes a copy of the page and maps the copy into the child's address space, replacing the read-only page that was there. The new page is marked read-write. In Fig. 8-10(b), the child has attempted to write to page 7. This action has resulted in page 7 being copied to page 8, and page 8 being mapped into the address space in place of page 7. Page 8 is marked read-write, so subsequent writes do not trap.

Copy-on-write has several advantages over doing all the copying at the time the new process is created. First, some pages are read-only, so there is no need to copy them. Second, other pages may never be referenced, so even if they are potentially writable, they do not have to be copied. Third, still other pages may be writable, but the child may deallocate them rather than using them. Here too, avoiding a copy is worthwhile. In this manner, only those pages that the child actually writes on have to be copied.

Fig. 8-10. Operation of copy-on-write. (a) After the FORK, all the child's pages are marked read-only. (b) When the child writes page 7, a copy is made.

Copy-on-write also has some disadvantages. For one thing, the administration is more complicated, since the system must keep track of the fact that some pages are genuinely read-only, with a write being a programming error, whereas other pages are to be copied if written. For another, copy-on-write requires multiple kernel traps, one for each page that is ultimately written. Depending on the hardware, one kernel trap followed by a multipage copy may not be that much more expensive than multiple kernel traps, each followed by a one-page copy. Finally, copy-on-write does not work over a network. Physical transport is always needed.

External Memory Managers
At the start of our discussion on memory management in Mach we briefly mentioned the existence of user-level memory managers. Let us now take a deeper look at them. Each memory object that is mapped in a process’ address space must have an external memory manager that controls it. Different classes of memory objects are handled by different memory managers. Each of these can implement its own semantics, can determine where to store pages that are not in memory, and can provide its own rules about what becomes of objects after they are mapped out.

To map an object into a process’ address space, the process sends a message to a memory manager asking it to do the mapping. Three ports are needed to do the job. The first, the **object port**, is created by the memory manager and will later be used by the kernel to inform the memory manager about page faults and other events relating to the object. The second, the **control port**, is created by the kernel itself so that the memory manager can respond to these events (many require some action on the memory manager’s part). The use of distinct ports is due to the fact that ports are unidirectional. The object port is written by the kernel and read by the memory manager; the control port works the other way around. The third port, the **name port**, is used as a kind of name to identify the object. For example, a thread can give the kernel a virtual address and ask which region it belongs to. The answer is a pointer to the name port. If addresses belong to the same region, they will be identified by the same name port.

When the memory manager maps in an object, it provides the capability for the object port as one of the parameters. The kernel then creates the other two ports and sends an initial message to the object port telling it about the control and name ports. The memory manager then sends back a reply telling the kernel what the object’s attributes are, and informing it whether or not to keep the object in its cache after it is unmapped. Initially, all the object’s pages are marked as unreadable/unwritable, to force a trap on the first use.

At this point the memory manager does a read on the object port and blocks. The memory manager remains idle until the kernel requests it to do something by writing a message to the object port. The thread that mapped the object in is now unblocked and allowed to execute.

Sooner or later, the thread will undoubtedly attempt to read or write a page belonging to the memory object. This operation will cause a page fault and a trap to the kernel. The kernel will then send a message to the memory manager via the object port, telling it which page has been referenced and asking it to please provide the page. This message is asynchronous because the kernel does not dare to block any of its threads waiting for a user process that may not reply. While waiting for a reply, the kernel suspends the faulting thread and looks for another thread to run.

When the memory manager hears about the page fault, it checks to see if the reference is legal. If not, it sends the kernel an error message. If it is legal, the memory manager gets the page by whatever method is appropriate for the object in question. If the object is a file, the memory manager seeks to the correct address and reads the page into its own address space. It then sends a reply back to the kernel providing a pointer to the page. The kernel maps the page into the faulting thread's address space, The thread can now be unblocked. This process is repeated as often as necessary to load all the pages needed.

To make sure that there is a steady supply of free page frames, a paging daemon thread in the kernel wakes up from time to time and checks the state of memory. If there are not enough free page frames, it picks a page to free using the second-chance algorithm. If the page is clean, it is normally just discarded. If the page is dirty, the daemon sends it to the memory manager in charge of the page's object. The memory manager is expected to write the page to disk and tell when it is done. If the page belongs to a file, the memory manager will first seek to the page's offset in the file, then write it there.

Pages can be marked as **precious**, in which cases they will never be just discarded, even if they are clean. They will always be returned to their memory manager. Precious pages can be used, for example, for pages shared over the network for which there is only one copy which must
never be discarded. Communication can also be initiated by the memory manager, for example, when a SYNC system call is done to flush the cache back to disk.

It is worth noting that the paging daemon is part of the kernel. Although the page replacement algorithm is completely machine independent, with a memory full of pages owned by different memory managers, there is no suitable way to let one of them decide which page to evict. The only method that might be possible is to statically partition the page frames among the various managers and let each one do page replacement on its set. However, since global algorithms are generally more efficient than local ones, this approach was not taken. Subsequent work has investigated this subject (Harty and Cheriton, 1992; and Subramian, 1991).

In addition to the memory managers for mapped files and other specialized objects, there is also a default memory manager for "ordinary" paged memory. When a process allocates a region of virtual address space using the allocate call, it is in fact mapping an object managed by the default manager. This manager provides zero-filled pages as needed. It uses a temporary file for swap space, rather than a separate swap area as UNIX does.

To make the idea of an external memory manager work, a strict protocol must be used for communication between the kernel and the memory managers. This protocol consists of a small number of messages that the kernel can send to a memory manager, and a small number of replies the memory manager can send back to the kernel. All communication is initiated by the kernel in the form of an asynchronous message on an object port for some memory object. Later, the memory manager sends an asynchronous reply on the control port.

Figure 8-11 lists the principal message types that the kernel sends to memory managers. When an object is mapped in using the map call of Fig. 8-8, the kernel sends an init message to the appropriate memory manager to let it initialize itself. The message specifies the ports to be used for discussing the object later. Requests from the kernel to ask for a page and deliver a page use data_request and data_write respectively. These handle the page traffic in both directions, and as such are the most important calls.

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>Initialize a newly mapped-in memory object</td>
</tr>
<tr>
<td>Data-request</td>
<td>Give kernel a specific page to handle a page fault</td>
</tr>
<tr>
<td>Data-write</td>
<td>Take a page from memory and write it out</td>
</tr>
<tr>
<td>Data-unlock</td>
<td>Unlock a page so kernel can use it</td>
</tr>
<tr>
<td>Lock-completed</td>
<td>Previous Lock_request has been completed</td>
</tr>
<tr>
<td>Terminate</td>
<td>Be informed that this object is no longer in use</td>
</tr>
</tbody>
</table>

*Fig. 8-11.* Selected message types from the kernel to the external memory managers.

Data_unlock is a request from the kernel for the memory manager to unlock a locked page so that the kernel can use it for another process. Lock_completed signals the end of a lock_request sequence, and will be described below. Finally, terminate tells the memory manager that the object named in the message is no longer in use and can be removed from memory. Some calls that are specific to the default memory manager also exist, as well as a few managing attributes and error handling.

The messages in Fig. 8-11 go from the kernel to the memory manager. The replies listed in Fig. 8-12 go the other way, from the memory manager back to the kernel. They are replies that the memory manager can use to respond to the above requests.
<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set-attributes</td>
<td>Reply to Init</td>
</tr>
<tr>
<td>Data-provided</td>
<td>Here is the requested page (Reply to Data-request)</td>
</tr>
<tr>
<td>Data-unavailable</td>
<td>No page is available (Reply to Data-request)</td>
</tr>
<tr>
<td>Lock-request</td>
<td>Ask kernel to clean, flush, or lock pages</td>
</tr>
<tr>
<td>Destroy</td>
<td>Destroy an object that is no longer needed</td>
</tr>
</tbody>
</table>

**Fig. 8-12.** Selected message types from the external memory managers to the kernel.

The first one, *set-attributes*, is a reply to *init*. It tells the kernel that it is ready to handle a newly mapped-in object. The reply also provides mode bits for the object and tells the kernel whether or not to cache the object, even if no process currently has it mapped in. The next two are replies to *data_request*. That call asks the memory manager to provide a page. Which reply it gives depends on whether or not it can provide the page. The former supplies the page; the latter does not.

*Lock_request* allows the memory manager to ask the kernel to make certain pages clean, that is, send it the pages so that they can be written to disk. This call can also be used to change the protection mode on pages (read, write, execute). Finally, *destroy* is used to tell the kernel that a certain object is no longer needed.

It is worth noting that when the kernel sends a message to a memory manager, it is effectively making an upcall. Although flexibility is gained this way, some system designers consider it inelegant for the kernel to call user programs to perform services for it. These people usually believe in hierarchical systems, with the lower layers providing services to the upper layers, not vice versa.

**Distributed Shared Memory in Mach**

The Mach external memory manager concept lends itself well to implementing a page-based distributed shared memory. In this section we will briefly describe some of the work done in this area. For more details, see (Forin et al., 1989). To review the basic concept, the idea is to have a single, linear, virtual address space that is shared among processes running on computers that do not have any physical shared memory. When a thread references a page that it does not have, it causes a page fault. Eventually, the page is located and shipped to the faulting machine, where it is installed so that the thread can continue executing.

Since Mach already has memory managers for different classes of objects, it is natural to introduce a new memory object, the shared page. Shared pages are managed by one or more special memory managers. One possibility is to have a single memory manager that handles all shared pages. Another is to have a different one for each shared page or collection of shared pages, to spread the load around.

Still another possibility is to have different memory managers for pages with different semantics. For example, one memory manager could guarantee complete memory coherence, meaning that any read following a write always sees the most recent data. Another memory manager could offer weaker semantics, for example, that a read never returns data that are more than 30 seconds out of date.

Let us consider the most basic case: one shared page, centralized control, and complete memory coherence. All other pages are local to a single machine. To implement this model, we need one memory manager that serves all the machines in the system. Let us call it the DSM (Distributed Shared Memory) server. The DSM server handles references to the shared page.
Conventional memory managers handle the other pages. Up until now we have tacitly assumed that the memory manager or managers that service a machine must be local to that machine. In fact, because communication is transparent in Mach, a memory manager need not reside on the machine whose memory it is managing.

The shared page is always either readable or writable. If it is readable, it may be replicated on multiple machines. If it is writable, there is only one copy. The DSM server always knows the state of the shared page as well as which machine or machines it is currently on. If the page is readable, DSM has a valid copy itself.

Suppose that the page is readable and a thread somewhere tries to read it. The DSM server just sends that machine a copy, updates its tables to indicate one more reader, and is finished. The page will be mapped in on the new machine for reading.

Now suppose that one of the readers tries to write the page. The DSM server sends a message to the kernel or kernels that have the page asking for it back. The page itself need not be transferred, because the DSM server has a valid copy itself. All that is needed is an acknowledgement that the page is no longer in use. When all the kernels have released the page, the writer is given a copy along with exclusive permission to use it (for writing).

If somebody else now wants the page (when it is writable), the DSM server tells the current owner to stop using it and to send it back. When the page arrives, it can be given to one or more readers or one writer. Many variations on this centralized algorithm are possible, such as not asking for a page back until the machine currently using it has had it for some minimum time. A distributed solution is also possible.

**COMMUNICATION IN MACH**

The goal of communication in Mach is to support a variety of styles of communication in a reliable and flexible way (Draves, 1990). It can handle asynchronous message passing, RPC, byte streams, and other forms as well. Mach’s interprocess communication mechanism is based on that of its ancestors, RIG and Accent. Due to this evolution, the mechanism used has been optimized for the local case (one node) rather than the remote case (distributed system).

We will first explain the single-node case in considerable detail, and then come back to how it has been extended for networking. It should be noted that in these terms, a multiprocessor is a single node, so communication between processes on different CPUs within the same multiprocessor uses the local case.

**Ports**

The basis of all communication in Mach is a kernel data structure called a **port**. A port is essentially a protected mailbox. When a thread in one process wants to communicate with a thread in another process, the sending thread writes the message to the port and the receiving thread takes it out. Each port is protected to ensure that only authorized processes can send to it and receive from it.

Ports support unidirectional communication, like pipes in UNIX. A port that can be used to send a request from a client to a server cannot also be used to send the reply back from the server to the client. A second port is needed for the reply.

Ports support reliable, sequenced, message streams. If a thread sends a message to a port, the system guarantees that it will be delivered. Messages are never lost due to errors, overflow, or other causes (at least if there are no crashes). Messages sent by a single thread are also guaranteed to be delivered in the order sent. If two threads write to the same port in an interleaved fashion, taking turns, the system does not provide any guarantee about message sequencing, since some buffering may take place in the kernel due to locking and other factors.

Unlike pipes, ports support message streams, not byte streams. Messages are never concatenated. If a thread writes five 100-byte messages to a port, the receiver will always see
them as five distinct messages, never as a single 500-byte message. Of course, higher-level software can ignore the message boundaries if they are not important to it.

A port is shown in Fig. 8-13. When a port is created, 64 bytes of kernel storage space are allocated and maintained until the port is destroyed, either explicitly, or implicitly under certain conditions, for example, when all the processes that are using it have exited. The port contains the fields shown in Fig. 8-13 and a few others.

Messages are not actually stored in the port itself but in another kernel data structure, the message queue. The port contains a count of the number of messages currently present in the message queue and the maximum permitted. If the port belongs to a port set, a pointer to the port set data structure is present in the port. As we mentioned briefly above, a process can give other processes capabilities to use its ports. For various reasons, the kernel has to know how many capabilities of each type are outstanding, so the port stores the counts.

If certain errors occur when using the port, they are reported by sending messages to other ports whose capabilities are stored there. Threads can block when reading from a port, so a pointer to the list of blocked threads is included. It is also important to be able to find the capability for reading from the port (there can only be one), so that information is present too. If the port is a process port, the next field holds a pointer to the process it belongs to. If it is a thread port, the field holds a pointer to the kernel’s data structure for the thread, and so on. A few miscellaneous fields not described here are also needed.

**Fig. 8-13.** A Mach port.

When a thread creates a port, it gets back an integer identifying the port, analogous to a file descriptor in UNIX. This integer is used in subsequent calls that send messages to the port or receive messages from it in order to identify which port is to be used. Ports are kept track of per process, not per thread, so if one thread creates a port and gets back the integer 3 to identify it, another thread in the same process will never get 3 to identify its new port. The kernel, in fact, does not even maintain a record of which thread created which port.

A thread can pass port access to another thread in a different process. Clearly, it cannot do so merely by putting the appropriate integer in a message, any more than a UNIX process can pass a file descriptor for standard output through a pipe by writing the integer 1 to the pipe. The exact mechanism used is protected by the kernel and will be discussed later. For the moment, it is sufficient to know that it can be done.
In Fig. 8-14 we see a situation in which two processes, A and B, each have access to the same port. A has just sent a message to the port, and B has just read the message. The header and body of the message are physically copied from A to the port and later from the port to B.

![Diagram of message passing](image)

**Fig. 8-14.** Message passing goes via a port.

Ports may be grouped into port sets for convenience. A port may belong to at most one port set. It is possible to read from a port set (but not write to one). A server, for example, can use this mechanism to read from a large number of ports at the same time. The kernel returns one message from one of the ports in the set. No promises are made about which port will be selected. If all the ports are empty, the server is blocked. In this way a server can maintain a different port for each of the many objects that it supports, and get messages for any of them without having to dedicate a thread to each one. The current implementation queues all the messages for the port set onto a single chain. In practice, the only difference between receiving from a port and receiving from a port set is that in the latter, the actual port sent to is identified to the receiver and in the former it is not.

Some ports are used in special ways. Every process has a special process port that it needs to communicate with the kernel. Most of the "system calls" associated with processes (see Fig. 8-3) are done by writing messages to this port. Similarly, each thread also has its own port for doing the "system calls" related to threads. Communication with I/O drivers also uses the port mechanism.

**Capabilities**

To a first approximation, for each process, the kernel maintains a table of all ports to which the process has access. This table is kept safely inside the kernel, where user processes cannot get at it. Processes refer to ports by their position in this table, that is, entry 1, entry 2, and so on. These table entries are effectively classical capabilities. We will refer to them as capabilities and will call the table containing the capabilities a capability list.

Each process has exactly one capability list. When a thread asks the kernel to create a port for it, the kernel does so and enters a capability for it in the capability list for the process to which the thread belongs. The calling thread and all the other threads in the same process have equal access to the capability. The integer returned to the thread to identify the capability is usually an index into the capability list (but it can also be a large integer, such as a machine address). We will refer to this integer as a capability name, (or sometimes just a capability, where the context makes it clear that we mean the index and not the capability itself). It is always a 32-bit integer, never a string.
Each capability consists not only of a pointer to a port, but also a rights field telling what access the holder of the capability has to the port. (All the threads in a process are equally considered holders of the process' capabilities.) Three rights exist: RECEIVE, SEND, and SEND-ONCE. The RECEIVE right gives the holder the ability to read messages from the port. Earlier we mentioned that communication in Mach is unidirectional. What this really means is that at any instant only one process may have the RECEIVE right for a port. A capability with a RECEIVE right may be transferred to another process, but doing so causes it to be removed from the sender's capability list. Thus for each port there is a single potential receiver.

A capability with the SEND right allows the holder to send messages to the specified port. Many processes may hold capabilities to send to a port. This situation is roughly analogous to the banking system in most countries: anyone who knows a bank account number can deposit money to that account, but only the owner can make withdrawals.

The SEND-ONCE right also allows a message to be sent, but only once. After the send is done, the kernel destroys the capability. This mechanism is used for request-reply protocols. For example, a client wants something from a server, so it creates a port for the reply message. It then sends the server a request message containing a (protected) capability for the reply port with the SEND-ONCE right. After the server sends the reply, the capability is deallocated from its capability list and the name is made available for a new capability in the future.

Capability names have meaning only within a single process. It is possible for two processes to have access to the same port but use different names for it, just as two UNIX processes may have access to the same open file but use different file descriptors to read it. In Fig. 8-15 both processes have a capability to send to port Y, but in A it is capability 3 and in B it is capability 4.

A capability list is tied to a specific process. When that process exits or is killed, its capability list is removed. Ports for which it holds a capability with the RECEIVE right are no longer usable and are therefore also destroyed, even if they contain undelivered (and now undeliverable) messages.

Fig. 8-15. Capability lists.

If different threads in a process acquire the same capability multiple times, only one entry is made in the capability list. To keep track of how many times each is present, the kernel maintains a reference count for each port. When a capability is deleted, the reference count is decremented. Only when it gets to zero is the capability actually removed from the capability list. This mechanism
is important because different threads may acquire and release capabilities without each other's knowledge, for example, the UNIX emulation library and the program being run.

Each capability list entry is one of the following four items:

1. A capability for a port.
2. A capability for a port set.
3. A null entry.
4. A code indicating that the port that was there is now dead.

The first possibility has already been explained in some detail. The second allows a thread to read from a set of ports without even being aware that the capability name is backed up by a set rather than by a single port. The third is a place holder that indicates that the corresponding entry is not currently in use. If an entry is allocated for a port that is later destroyed, the capability is replaced by a null entry to mark it as unused.

Finally, the fourth option marks ports that no longer exist but for which capabilities with SEND rights still exist. When a port is deleted, for example, because the process holding the RECEIVE capability for it has exited, the kernel tracks down all the SEND capabilities and marks them as dead. Attempts to send to null and dead capabilities fail with an appropriate error code. When all the SEND capabilities for a port are gone, for whatever reasons, the kernel (optionally) sends a message notifying the receiver that there are no senders left and no messages will be forthcoming.

Primitives for Managing Ports

Mach provides about 20 calls for managing ports. All of these are invoked by sending a message to a process port. A sampling of the most important ones is given in Fig. 8-16.

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocate</td>
<td>Create a port and insert its capability in the capability list</td>
</tr>
<tr>
<td>Destroy</td>
<td>Destroy a port and remove its capability from the list</td>
</tr>
<tr>
<td>Deallocate</td>
<td>Remove a capability from the capability list</td>
</tr>
<tr>
<td>Extract-right</td>
<td>Extract the n-th capability from another process</td>
</tr>
<tr>
<td>Insert-right</td>
<td>Insert a capability in another process' capability list</td>
</tr>
<tr>
<td>Move-member</td>
<td>Move a capability into a capability set</td>
</tr>
<tr>
<td>Set qlimit</td>
<td>Set the number of messages a port can hold</td>
</tr>
</tbody>
</table>

Fig. 8-16. Selected port management calls in Mach.

The first one, allocate, creates a new port and enters its capability into the caller's capability list. The capability is for reading from the port. A capability name is returned so that the port can be used.

The next two undo the work of the first. Destroy removes a capability. If it is a RECEIVE capability, the port is destroyed and all other capabilities for it in all processes are marked as dead. Deallocate decrements the reference count associated with a capability. If it is zero, the capability is removed but the port remains intact. Deallocate can only be used to remove SEND or SEND-ONCE capabilities or dead capabilities.

Extract_right allows a thread to select out a capability from another process' capability list and insert the capability in its own list. Of course, the calling thread needs access to the process port.
controlling the other process (e.g., its own child). Insert_right goes the other way. It allows a process to take one of its own capabilities and add it to (for example) a child's capability list.

The move_member call is used for managing port sets. It can add a port to a port set or remove one. Finally, set qlimit determines the number of messages a port can hold. When a port is created, the default is five messages, but with this call that number can be increased or decreased. The messages can be of any size since they are not physically stored in the port itself.

Sending and Receiving Messages

The purpose of having ports is to send messages to them. In this section we will look at how messages are sent, how they are received, and what they contain. Mach has a single system call for sending and receiving messages. The call is wrapped in a library procedure called mach_msg. It has seven parameters and a large number of options. To give an idea of its complexity, there are 35 different error messages that it can return. Below we will give a simplified sketch of some of its possibilities. Fortunately, it is used primarily in procedures generated by the stub compiler, rather than being written by hand.

The mach_msg call is used for both sending and receiving. It can send a message to a port and then return control to the caller immediately, at which time the caller can modify the message buffer without affecti

A typical call to mach_msg looks like this:

mach_msg(&hdr,options,send_size,rcv_size,rcv_port,timeout,notify_port);

The first parameter, hdr, is a pointer to the message to be sent or to the place where the incoming message is put, or both. The message begins with a fixed header and is followed directly by the message body. This layout is shown in Fig. 8-17. We will explain the details of the message format later, but for the moment just note that the header contains a capability name for the destination port. This information is needed so that the kernel can tell where to send the message. When doing a pure RECEIVE, the header is not filled in, since it will be overwritten entirely by the incoming message.

The second parameter of the mach_msg call, options, contains a bit specifying that a message is to be sent, and another one specifying that a message is to be received. If both are on, an RPC is done. Another bit enables a timeout, given by the timeout parameter, in milliseconds. If the requested operation cannot be performed within the timeout interval, the call returns with an error code. If the SEND portion of an RPC times out (e.g., due to the destination port being full too long), the receive is not even attempted.
Fig. 8-17. The Mach message format.

Other bits in options allow a SEND that cannot complete immediately to return control anyway, with a status report being sent to notify_port later. All kinds of errors can occur here if the capability for notify_port is unsuitable or changed before the notification can occur. It is even possible for the call to ruin notify_port itself (calls can have complex side effects, as we will see later).

The mach_msg call can be aborted part-way through by a software interrupt. Another options bit tells whether to give up or try again.

The send_size and rcv_size parameters tell how large the outgoing message is and how many bytes are available for storing the incoming message, respectively. Rcv_port is used for receiving messages. It is the capability name of the port or port set being listened to.

Now let us turn to the message format of Fig. 8-17. The first word contains a bit telling whether the message is simple or complex. The difference is that simple messages cannot carry capabilities or protected pointers, whereas complex ones can. Simple messages require less work on the part of the kernel and are therefore more efficient. Both message types have a system-defined structure, described below. The message size field tells how big the combined header plus body is. This information is needed both for transmission and by the receiver.

Next come two capability names (i.e., indices into the sender's capability list). The first specifies the destination port; the second can give a reply port. In client-server RPC, for example, the destination field designates the server and the reply field tells the server which port to send the response to.

The last two header fields are not used by the kernel. Higher levels of software can use them as desired. By convention, they are used to specify the kind of message and give a function code or operation code (e.g., to a server, is this request for reading or for writing?). This usage is subject to change in the future.

When a message is sent and successfully received, it is copied into the destination's address space. It can happen, however, that the destination port is already full. What happens then depends on the various options and the rights associated with the destination port. One possibility is that the sender is blocked and simply waits until space becomes available in the port. Another is that the sender times out. In some cases, it can exceed the port limit and send anyway.
A few issues concerning receiving messages are worth mentioning. For one, if an incoming message is larger than the buffer, what should be done with it? Two options are provided: throw it away or have the `mach_msg` call fail but return the size, thus allowing the caller to try again with a bigger size.

If multiple threads are blocked trying to read from the same port and a message arrives, one of them is chosen by the system to get the message. The rest remain blocked. If the port being read from is actually a port set, it is possible for the composition of the set to change while one or more threads are blocked on it. This is probably not the place to go into all the details, but suffice it to say that there are precise rules governing this and similar situations.

## Message Formats

A message body can be either simple or complex, controlled by a header bit, as mentioned above. Complex messages are structured as shown in Fig. 8-17. A complex message body consists of a sequence of (descriptor, data field) pairs. Each descriptor tells what is in the data field immediately following it. Descriptors come in two formats, differing only in how many bits each of the fields contains. The normal descriptor format is illustrated in Fig. 8-18. It specifies the type of the item that follows, how large an item is, and how many of them there are (a data field may contain multiple items of the same type). The available types include raw bits and bytes, integers of various sizes, unstructured machine words, collections of Booleans, floating-point numbers, strings, and capabilities. Armed with this information, the system can attempt to do conversions between machines when the source and destination machines have different internal representations. This conversion is not done by the kernel but by the network message server (described below). It is also done for internode transport even for simple messages (also by the network message server).

One of the more interesting items that can be contained in a data field is a capability. Using complex messages it is possible to copy or transfer a capability from one process to another. Because capabilities are protected kernel objects in Mach, a protected mechanism is needed to move them about.

![Fig. 8-18. A complex message field descriptor.](image)

This mechanism is as follows. A descriptor can specify that the word directly after it in the message contains the name for one of the sender's capabilities, and that this capability is to be passed to the receiving process and inserted in the receiver's capability list. The descriptor also specifies whether the capability is to be copied (the original is not touched) or moved (the original is deleted).
Furthermore, certain values of the *Data field type* ask the kernel to modify the capability's rights while doing the copy or move. A RECEIVE capability, for example, can be mutated into a SEND or SEND-ONCE capability, so that the receiver will have the power to send a reply to a port for which the sender has only a RECEIVE capability. In fact, the normal way to establish communication between two processes is to have one of them create a port and then send the port's RECEIVE capability to the other one, turning it into a SEND capability in flight.

To see how capability transport works, consider Fig. 8-19(a). Here we see two processes, *A* and *B*, with 3 capabilities and 1 capability, respectively. All are RECEIVE capabilities. Numbering starts at 1 since entry 0 is the null port. One of the threads in *A* is sending a message to *B* containing capability 3.

When the message arrives, the kernel inspects the header and sees that it is a complex message. It then begins processing the descriptors in the message body, one by one. In this example there is only one descriptor, for a capability, with instructions to turn it into a SEND (or maybe SEND-ONCE) capability. The kernel allocates a free slot in the receiver's capability list, slot 2 in this example, and modifies the message so that the word following the descriptor is now 2 instead of 3. When the receiver gets the message, it sees that it has a new capability, with name (index) 2. It can use this capability immediately (e.g., for sending a reply message).

![Fig. 8-19. (a) Situation just before the capability is sent. (b) Situation after it has arrived.](image)

There is one last aspect of Fig. 8-18 that we have not yet discussed: **out-of-line data**. Mach provides a way to transfer bulk data from a sender to a receiver without doing any copying (on a single machine or multiprocessor). If the out-of-line data bit is set in the descriptor, the word following the descriptor contains an address, and the size and number fields of the descriptor give a 20-bit byte count. Together these specify a region of the sender's virtual address space. For larger regions, the long form of the descriptor is used.

When the message arrives at the receiver, the kernel chooses an unallocated piece of virtual address space the same size as the out-of-line data, and maps the sender's pages into the receiver's address space, marking them copy-on-write. The address word following the descriptor is
changed to reflect the address at which the region is located in the receiver’s address space. This mechanism provides a way to move blocks of data at extremely high speed, because no copying is required except for the message header and the two-word body (the descriptor and the address). Depending on a bit in the descriptor, the region is either removed from the sender’s address space or kept there.

Although this method is highly efficient for copies between processes on a single machine (or between CPUs in a multiprocessor), it is not as useful for communication over a network because the pages must be copied if they are used, even if they are only read. Thus the ability to transmit data logically without moving physically them is lost. Copy-on-write also requires that messages be aligned on page boundaries and be an integral number of pages in length for best results. Fractional pages allow the receiver to see data before or after the out-of-line data that it should not see.

**The Network Message Server**

Everything we have said so far about communication in Mach is limited to communication within a single node, either one CPU or a multiprocessor node. Communication over the network is handled by user-level servers called network message servers, which are vaguely analogous to the external memory managers we studied earlier. Every machine in a Mach distributed system runs a network message server. The network message servers work together to handle intermachine messages, trying to simulate intramachine messages as best they can.

A network message server is a multithreaded process that performs a variety of functions. These include interfacing with local threads, forwarding messages over the network, translating data types from one machine’s representation to another’s, managing capabilities in a secure way, doing remote notification, providing a simple network-wide name lookup service, and handling authentication of other network message servers. Network message servers can speak a variety of protocols, depending on the networks to which they are attached.

The basic method by which messages are sent over the network is illustrated in Fig. 8-20. Here we have a client on machine $A$ and a server on machine $B$. Before the client can contact the server, a port must be created on $A$ to function as a proxy for the server. The network message server has the RECEIVE capability for this port. A thread inside it is constantly listening to this port (and other remote ports, which together form a port set). This port is shown as the small box in $A$’s kernel.
Fig. 8-20. Intermachine communication in Mach proceeds in five steps.

Message transport from the client to the server requires five steps, numbered 1 to 5 in Fig. 8-20. First, the client sends a message to the server's proxy port. Second, the network message server gets this message. Since this message is strictly local, out-of-line data may be sent to it and copy-on-write works in the usual way. Third, the network message server looks up the local port, 4 in this example, in a table that maps proxy ports onto network ports. Once the network port is known, the network message server looks up its location in other tables. It then constructs a network message containing the local message, plus any out-of-line data and sends it over the LAN to the network message server on the server's machine. In some cases, traffic between the network message servers has to be encrypted for security. The transport module takes care of breaking the message into packets and encapsulating them in the appropriate protocol wrappers.

When the remote network message server gets the message, it looks up the network port number contained in it and maps it onto a local port number. In step 4, it writes the message to the local port just looked up. Finally, the server reads the message from the local port and carries out the request. The reply follows the same path in the reverse direction.

Complex messages require a bit more work. For ordinary data fields, the network message server on the server's machine must perform conversion, if necessary, for example, taking account of different byte ordering on the two machines. Capabilities must also be processed. When a capability is sent over the network, it must be assigned a network port number, and both the source and destination network message servers must make entries for it in their mapping tables. If these machines do not trust each other, elaborate authentication procedures will be necessary to convince each machine of the other's true identity.

Although the idea of relaying messages from one machine to another via a user-level server offers some flexibility, a substantial price is paid in performance as compared to a pure kernel implementation, which most other distributed systems use. To solve this problem, a new version of the network communication package is being developed (the NORMA code), which runs inside the kernel and achieves faster communication. It will eventually replace the network message server.