CHAPTER 12

SECONDARY STORAGE STRUCTURE

The file system can be logically viewed as consisting of three parts. In Chapter 10, we saw the user and programmer interface to the file system. In Chapter 11, we described the internal data structures and algorithms used by the operating system to implement this interface. In this chapter we discuss the lowest level of the file system — the secondary storage structure. We first describe disk-head-scheduling algorithms. Next we discuss disk formatting, and management of boot blocks, damaged blocks, and swap space. We end with coverage of disk reliability and stable-storage.

12.1 ■ Disk Structure

Disks provide the bulk of secondary storage for modern computer systems. Magnetic tape was used as an early secondary-storage medium. Although it is relatively permanent, and can hold large numbers of data, magnetic tape is slow in comparison to the access time of main memory. Even more important, magnetic tape is limited to sequential access. Thus, it is unsuitable for providing the random access needed by files. Tapes are currently used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another.

Information on the disk is referenced by a multipart address, which includes the drive number, the surface, the track, and the sector. All the tracks on one drive that can be accessed without the heads being moved (the equivalent tracks on the different surfaces) constitute a *cylinder*.

EX-1015
U.S. Patent 8,073,681

Within a track, information is stored in sectors. IBM mainframes allow the user to select the size of the sector, but most other disks have a sector size fixed by hardware. A *sector* is the smallest unit of information that can be read from or written to the disk. Depending on the disk drive, sectors vary from 32 bytes to 4096 bytes; usually, they are 512 bytes. There are 4 to 32 sectors per track, and from 20 to 1500 tracks per disk surface. To access a sector, we must specify the surface, track, and sector. To help the disk drive to locate its sector location, the drive records sector marks between the sectors. The read–write heads are moved to the correct track (seek time) and are electronically switched to the correct surface; then, we wait (latency time) for the requested sector to rotate below the heads. Seek time is dependent on the time it takes for the disk heads to move, so the farther apart the tracks are, the longer the seek time is.

I/O transfers between memory and disk are performed in units of one or more sectors, called *blocks*, to improve I/O efficiency. Addressing a particular block requires a track (or cylinder) number, a surface number, and a sector number. Thus, the disk can be viewed as a three-dimensional array of blocks. Commonly, this array is treated by the operating system as a one-dimensional array of blocks. Typically, block addresses increase through all blocks on a track, then through all the tracks in a cylinder, and finally from cylinder 0 to the last cylinder on the disk. We use s to denote the number of sectors per track, and t to denote the number of tracks per cylinder; clearly, we can convert from a disk address of cylinder i, surface j, sector k to a one-dimensional block number b, by

$$b = k + s \times (j + i \times t).$$

Notice that, with this mapping, accessing block $b + 1$ when the last block accessed was b requires a seek only when b was the last block of one cylinder and $b + 1$ is the first block of the next cylinder. Even in this case, the head is moved only one track.

12.2 ■ Disk Scheduling

Because most jobs depend heavily on the disk for program loading and input and output files, it is important that disk service be as fast as possible. The operating system can improve on the average disk service time by scheduling the requests for disk access.

Disk speed is composed of three parts. To access a block on the disk, the system must first move the head to the appropriate track or cylinder. This head movement is called a *seek*, and the time to complete it is *seek time*. Once the head is at the right track, it must wait until the desired block rotates under the read–write head. This delay is *latency* time. Finally,

the actual transfer of data between the disk and main memory can take place. This last part is *transfer* time. The total time to service a disk request is the sum of the seek time, latency time, and transfer time.

As we discussed in Chapter 2, every I/O device, including each disk drive, has a queue of pending requests. Whenever a process needs I/O to or from the disk, it issues a system call to the operating system. The request specifies several pieces of necessary information:

- Whether this operation is input or output
- What is the disk address (a block number, translated by the file organization module into drive, cylinder, surface, and sector coordinates)
- What is the memory address to copy to or from
- What is the amount of information to be transferred (a byte count)

If the desired disk drive and controller are available, the request can be serviced immediately. However, while the drive or controller is serving one request, any additional requests, normally from other processes, will need to be queued.

For a multiprogramming system with many processes, the disk queue may often be nonempty. Thus, when a request is complete, we must pick a new request from the queue and service it. A disk service requires a move of the head to the desired track, then a wait for latency, and finally a transfer of the data.

12.2.1 FCFS Scheduling

The simplest form of disk scheduling is, of course, *first-come, first-served (FCFS)* scheduling. This algorithm is easy to program and is intrinsically fair. However, it may not provide the best (average) service. Consider, for example, an ordered disk queue with requests involving tracks

98, 183, 37, 122, 14, 124, 65, and 67,

listed first (98) to last (67). If the read–write head is initially at track 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 tracks. This schedule is diagrammed in Figure 12.1.

The problem with this schedule is illustrated by the wild swing from 122 to 14 and then back to 124. If the requests for tracks 37 and 14 could be serviced together, before or after the requests at 122 and 124, the total head movement could be decreased substantially and the average time to service each request would decrease, improving disk throughput.

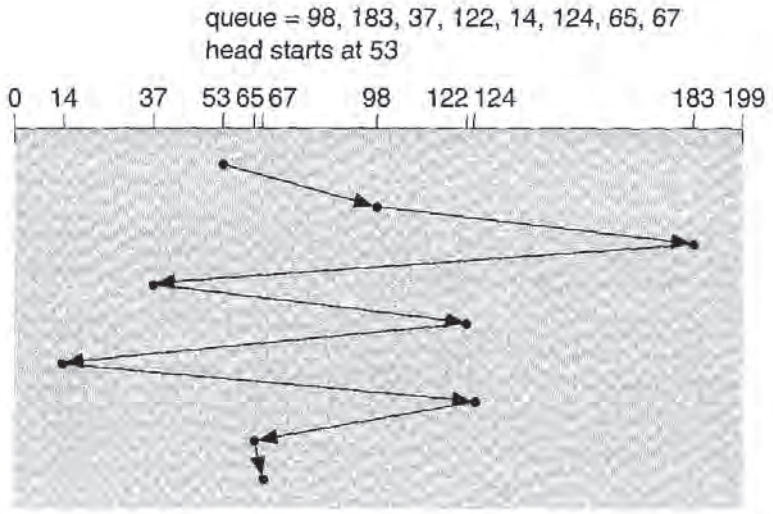


Figure 12.1 FCFS disk scheduling.

12.2.2 SSTF Scheduling

It seems reasonable to service together all requests close to the current head position, before moving the head far away to service another request. This assumption is the basis for the *shortest-seek-time-first (SSTF)* disk-scheduling algorithm. The SSTF algorithm selects the request with the minimum seek time from the current head position. Since seek time is generally proportional to the track difference between the requests, we implement this approach by moving the head to the closest track in the request queue.

For our example request queue, the closest request to the initial head position (53) is at track 65. Once we are at track 65, the next closest request is at track 67. At this point, the distance to track 37 is 30, whereas the distance to 98 is 31. Therefore, the request at track 37 is closer and is served next. Continuing, we service the request at track 14, then 98, 122, 124, and finally at 183 (Figure 12.2). This scheduling method results in a total head movement of only 236 tracks — little more than one-third of the distance needed for FCFS scheduling. This algorithm would result in a substantial improvement in average disk service.

SSTF scheduling is essentially a form of shortest-job-first (SJF) scheduling, and, like SJF scheduling, it may cause *starvation* of some requests. Remember that, in a real system, requests may arrive at any time. Assume that we have two requests in the queue, for 14 and 186. If a request near 14 arrives while we are servicing that request, it will be serviced next, making the request at 186 wait. While this request is being serviced, another request close to 14 could arrive. In theory, a continual stream of requests near one another could arrive, causing the request for

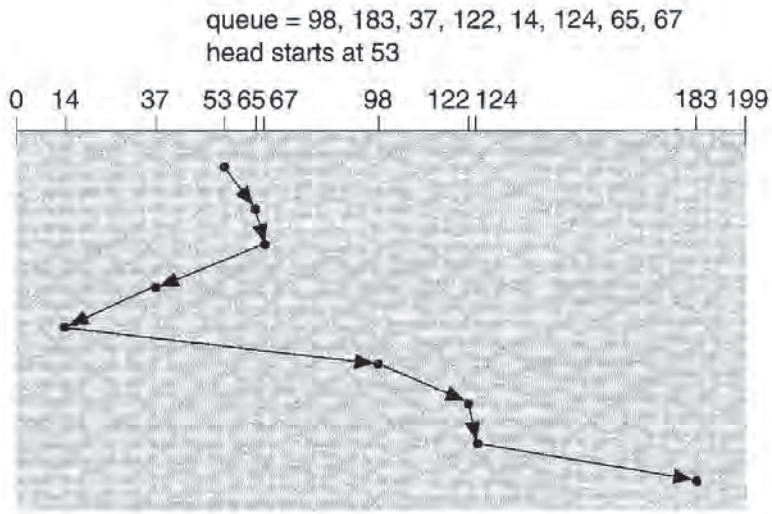


Figure 12.2 SSTF disk scheduling.

track 186 to wait indefinitely. This scenario is statistically unlikely, but is possible.

Finally, the SSTF algorithm, although a substantial improvement over the FCFS algorithm, is not optimal. For example, if we move the head from 53 to 37, even though the latter is not closest, and then to 14, before turning around to service 65, 67, 98, 122, 124, and 183, we can reduce the total head movement to 208 tracks.

12.2.3 SCAN Scheduling

Recognition of the dynamic nature of the request queue leads to the SCAN algorithm. The read–write head starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each track, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed and servicing continues. The head continuously scans the disk from end to end. We again use our example.

Before applying SCAN to scheduling,

98, 183, 37, 122, 14, 124, 65, and 67

we need to know the direction of head movement, in addition to the head's last position. If the head was moving toward 0, the head movement would service 37 and 14 as it moved to 0. At track 0, the head would reverse and move to the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183 as it moves (Figure 12.3). If a request arrives in the queue just in front of the head, it will be serviced almost immediately, whereas a request arriving just behind the head will have to wait until the

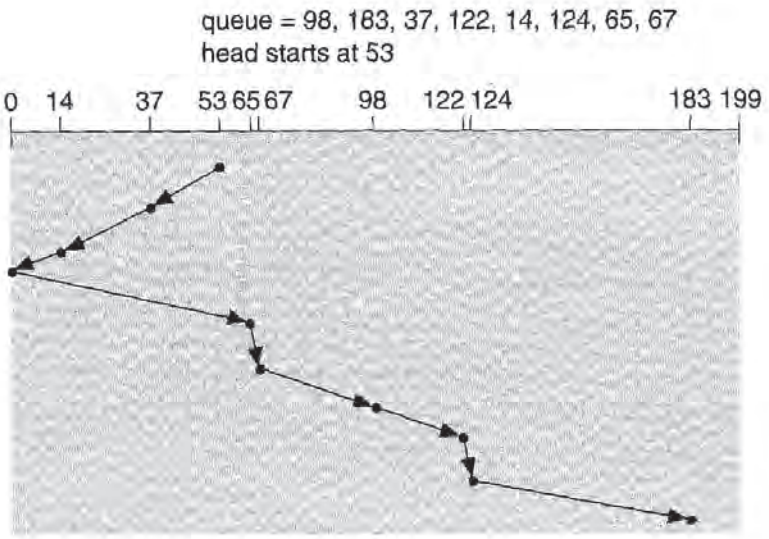


Figure 12.3 SCAN disk scheduling.

head moves to the end of the disk, reverses direction, and returns, before being serviced.

The SCAN algorithm is sometimes called the *elevator algorithm*, since it is similar to the behavior of elevators as they service requests to move from floor to floor in a building. Another analogy is that of shoveling snow from a sidewalk during a snowstorm. Starting from one end, we remove snow as we move toward the other end. As we move, new snow falls behind us. At the far end, we reverse direction and remove the newly fallen snow behind us.

Assuming a uniform distribution of requests for tracks, consider the density of requests when the head reaches one end and reverses direction. At this point, there are relatively few requests immediately behind the head, since these tracks have recently been serviced. The heaviest density of requests is at the other end of the disk. These requests have also waited the longest.

12.2.4 C-SCAN Scheduling

A variant of SCAN scheduling that is designed to provide a more uniform wait time is C-SCAN (circular SCAN) scheduling. As does SCAN scheduling, C-SCAN scheduling moves the head from one end of the disk to the other, servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip (Figure 12.4). The C-SCAN scheduling algorithm essentially treats the disk as though it were circular, with the last track adjacent to the first one.

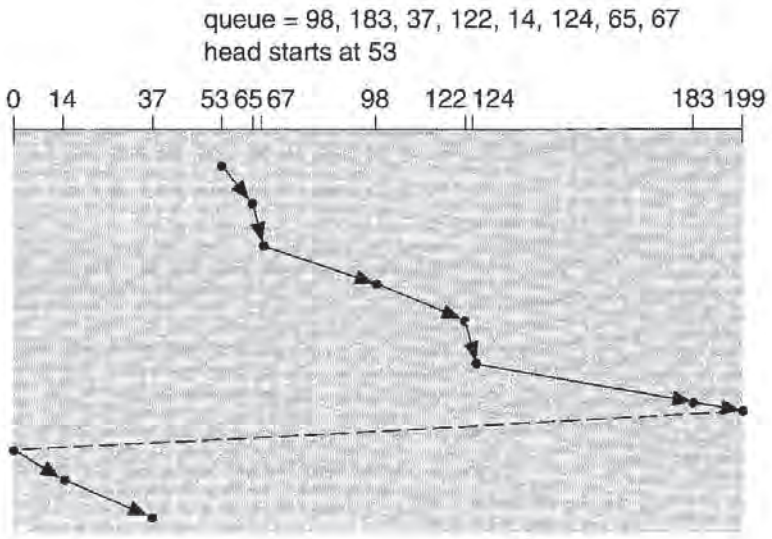


Figure 12.4 C-SCAN disk scheduling.

12.2.5 LOOK Scheduling

Notice that, as we described them, both SCAN and C-SCAN scheduling always move the head from one end of the disk to the other. In practice, neither algorithm is implemented in this way. More commonly, the head is only moved as far as the last request in each direction. As soon as there are no requests in the current direction, the head movement is reversed. These versions of SCAN and C-SCAN scheduling are called LOOK scheduling (look for a request before moving in that direction) and C-LOOK scheduling (Figure 12.5).

12.2.6 Selecting a Disk-Scheduling Algorithm

Given so many disk-scheduling algorithms, how do we choose a particular algorithm? SSTF scheduling is common and has a natural appeal. The SCAN and C-SCAN scheduling algorithms are more appropriate for systems that place a heavy load on the disk. It is possible to define an optimal algorithm, but the computation needed for an optimal schedule may not justify the savings over SSTF or SCAN scheduling.

With any scheduling algorithm, however, performance depends heavily on the number and types of requests. In particular, if the queue seldom has more than one outstanding request, then all scheduling algorithms are effectively equivalent. In this case, FCFS scheduling is also a reasonable algorithm.

Notice also that the requests for disk service can be greatly influenced by the file-allocation method. A program reading a contiguously allocated

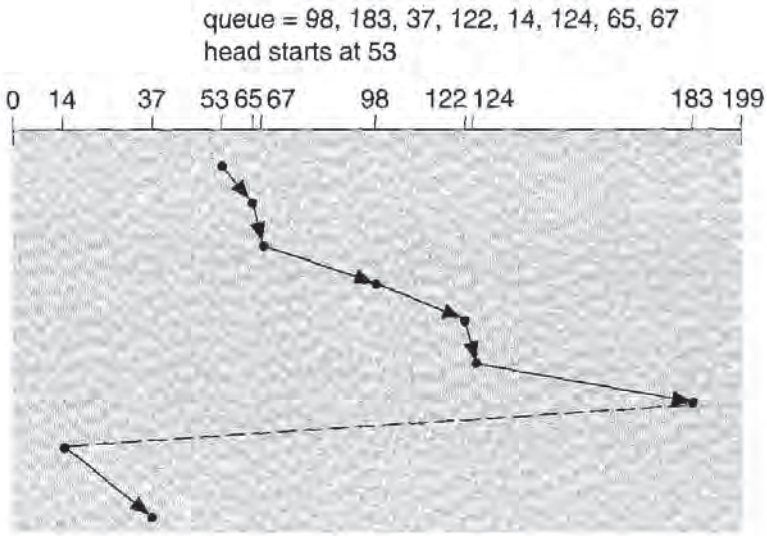


Figure 12.5 C-LOOK disk scheduling.

file will generate several requests that are close together on the disk, resulting in limited head movement. A linked or indexed file, on the other hand, may include blocks that are widely scattered on the disk, resulting in better disk-space utilization at the expense of head movement.

The location of directories and index blocks also is important. Since every file must be opened to be used, and opening a file requires searching the directory structure, the directories will be accessed frequently. Placing the directories halfway between the inner and outer edge of the disk, rather than at either end, can significantly reduce disk-head movement. For instance, if a directory entry is in the first sector and a file's data are in the last sector, then the disk head needs to move the entire width of the disk. If the directory entry is more toward the middle, the head will have to move at most one-half the width.

It should be clear that, as a result of these considerations, the disk-scheduling algorithm, like all others, should be written as a separate module of the operating system, allowing it to be removed and to be replaced with a different algorithm if necessary. Either FCFS or SSTF scheduling is a reasonable initial choice. In fact disk controller manufacturers have been helping operating system designers by moving head scheduling algorithms into the hardware itself. The operating system sends requests to the controller in FCFS order, and the controller queues them and executes them in some more optimal order. Unfortunately, if the operating system is unaware of the controller's methodology, the operating system may be ordering the requests in a way it expects to be optimal, and the controller may be rearranging them into the order it likes best. The result could be decreased, rather than increased, performance!

12.3 ■ Disk Management

There are several other aspects of disk management for which an operating system is responsible. Here we discuss disk initialization, booting from disk, and bad-block recovery.

12.3.1 Disk Formatting

When a magnetic disk is manufactured, it is essentially a blank slate, just a platter or platters of magnetically changeable medium. Before the computer can make use of the disk, the disk must be broken into the sectors that the computer can understand. This process is called *physical formatting*. When formatted, a disk consists of a set of sectors on each track the head can address. Each sector has a header containing the sector number (so the disk knows when it has found the correct sector) and space for an *error correcting code* (ECC). When data are written to the sector, the ECC is updated with a value calculated from the values of each byte written into the sector. When the sector is read, the ECC is recalculated and is compared with the stored value. If the stored and calculated numbers are different, the data are corrupted and the disk sector may be bad (see Section 12.3.3). This calculation and comparison are done automatically by the controller.

Most hard disks come physically formatted from the factory. The operating system still needs to partition and to *format logically* a disk before it can use that disk. This logical formatting writes an initial, blank directory onto the disk, and may install a FAT, inodes, free space lists, or any other information the system needs to track the disk's contents. Some database systems do not require their partitions to be logically formatted. Rather, they use the raw disk and bypass all the operating system I/O calls. This bypassing results in performance gain at the cost of duplicating some operating-system code.

12.3.2 Boot Block

For a computer to start running — for instance, when it is powered up or rebooted — it needs to have an initial program to run. This initial program, or *bootstrap program*, tends to be simple. It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and to start it executing. To accomplish this goal, the bootstrap program must locate the operating system kernel, load it into memory, and jump to an initial address.

How does the computer know the location of the bootstrap program to start it running? Some systems, such as the Apple Macintosh, store most

of the bootstrap in read-only memory (ROM). This location is convenient, because ROM needs no initialization, and so is always available. The problem is that changing the bootstrap requires changing the ROM hardware chips. For this reason, most systems store a little of the bootstrap in ROM, and store the remainder in the *boot blocks* at a fixed location on a disk. This disk is known as the *boot disk* or *system disk*; the system cannot function without it.

The code stored in ROM requests the block transfer from the disk controller (no device drivers are loaded at this point), loads data from the boot blocks into memory, and starts executing the new code. Changes to the boot program are made by rewriting of the boot blocks. The program stored in the boot blocks is more sophisticated than is the one stored in the ROM, and is able to load the entire operating system from a nonfixed location on disk, and to start the operating system running. It may be able to do so with very little code. For example, the IBM PC uses one 512-byte block for its boot program (Figure 12.6).

12.3.3 Bad Blocks

Because disks have moving parts and small tolerances (recall that the magnetic-disk read–write head floats just above the disk surface), they are prone to failure. Sometimes, the failure is complete, and the disk needs to be replaced, and its contents restored from backup media to the replacement. More frequently, one or more blocks become unreadable and unwritable. Most disks even come from the factory with *bad blocks*. Depending on the disk and controller in use, these blocks are handled in a variety of ways.

On the IBM PC family with IDE drives, bad blocks must be handled manually. The **format** command finds bad blocks when the disk is logically formatted, and writes a special value into the FAT entries to tell

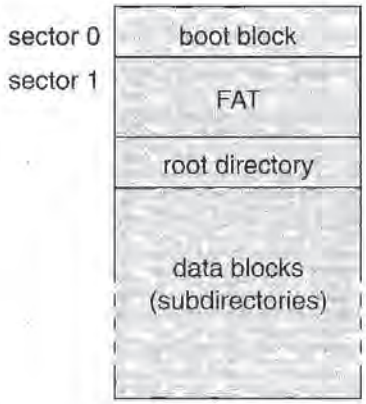


Figure 12.6 MS-DOS disk layout.

the allocation routines not to use those blocks. If blocks go bad during normal operation, a special program is run manually to search for new bad blocks and to lock them away as before. Data that resided on the bad blocks usually are lost.

The SCSI disk and controller, used in high-end PCs and most workstations, are smarter about bad-block recovery. The controller keeps a list of bad blocks on the disk, initialized during physical format and updated over the life of the disk. The controller also sets aside sectors not available to the operating system. For each bad sector, the controller can be told to replace it logically with a sector from this pool. This scheme is known as *slipping* or *forwarding* the sector. Thus, the operating system might request a write of sector 10 on track 12, but the controller would notice that sector is bad and would instead write sector 27 on track 153. A typical bad-sector transaction might then be as follows:

- The operating system requests sector 5, track 20.
- The controller calculates the ECCs and finds the sector is bad. It reports this finding to the operating system.
- The next time that the system is rebooted, a special command is run to tell the SCSI controller to replace the bad sector with a good one.
- The next time the system requests sector 5, track 20, the request is translated into the replacement sector's address by the controller.

Note that such a redirection by the controller could invalidate any optimization by the disk-scheduling algorithm! As a solution, some disks place replacement sectors in each cylinder, and then try to *slip* a sector with another from the same cylinder. Since a low percentage of sectors is bad, little performance is actually lost. An alternative is to move the disk-scheduling algorithm into the disk controller, since it knows the "real" location of all blocks on the disk.

The reason that the correction is not done totally automatically is that the data in the bad block usually are lost, and the file must be repaired either manually or through a file restoration.

12.4 ■ Swap-Space Management

Swap-space management is another low-level task of the operating system. Virtual memory requires the use of disk as an extension of memory. Unfortunately, disk access is much slower than memory access. Because swap space has a large impact on system performance, its design and implementation are focused on providing the best throughput for the virtual memory system. In this section we discuss how swap space is used, where swap space is located on disk, and how the swap space is managed.

12.4.1 Swap-Space Use

Swap space is used in various ways by different operating systems, depending upon the implemented memory management algorithms. For instance, systems which implement swapping may use swap space to hold the entire process image, including the code and data sections. Paging systems may simply store pages which have been pushed out of main memory. The amount of swap space needed on a system can therefore vary depending on the amount of memory it is backing and the way in which it is used. On personal computers, only a few megabytes of disk space may be needed. On UNIX workstations and minicomputers, tens or hundreds of megabytes may be used.

Some operating systems, such as UNIX, allow the use of multiple swap spaces. These extra swap spaces are usually put on separate disks, so the load placed on the I/O system by paging and swapping can be balanced over the system's I/O devices.

Note that it is safer to overestimate than to underestimate swap space, because if a system runs out of swap space it may be forced to abort processes or crash entirely. Overestimation results in wasted disk space which could otherwise be used for files, but does no other harm.

12.4.2 Swap-Space Location

There are two places a swap space can reside. Swap space can be carved out of the normal file system, or can be a separate disk partition. If the swap space is simply a large file within the file system (as in Microsoft Windows), normal file system routines can be used to create and destroy it, name it, and allocate its space. Adding another swap space is as easy as creating another file. This approach is therefore easy to implement. Unfortunately, it is also inefficient. Navigating the directory structure and the disk allocation data structures takes time, and (potentially) extra disk accesses. External fragmentation may also greatly increase swap space read and write times as reading an entire process image can require multiple seeks. Performance can be optimized by using caches to hold the block location information, and using special tools to allocate sequential blocks for this file, but still there is the added cost of traversing the in-cache data structures.

More commonly, swap space is created in a separate disk partition. No file system or directory structure is placed on this space. Rather, separate swap-space management algorithms are used to allocate and deallocate the blocks. The advantage of this method is its speed. The algorithms are optimized for very fast storage and retrieval of data, rather than storage efficiency. Internal fragmentation is increased, but this tradeoff is acceptable because data in swap space generally live for much shorter amounts of time than files in the file system, and this data is

accessed much more frequently than that in the file system. Unfortunately, this raw method requires that space be set aside during file system creation. Adding more swap space can only be done by destroying extant file systems or adding new disks.

Some systems are flexible and allow both raw and file-system based swap spaces. Solaris 2 is an example of this. The policy and implementation are separate, allowing the machine's administrator to decide which type to use. The tradeoff is between convenience of allocation and management, and system performance.

12.4.3 Swap-Space Management

As an example of the methods used to manage swap space, we now follow the evolution of swapping and paging in UNIX. As fully discussed in Chapter 19, UNIX started with an implementation of swapping. Entire processes were written to swap space or read into memory, contiguously, at one time. UNIX moved to a combination of swapping and paging as paging hardware became available.

In 4.3BSD, swap space is allocated to a process when the process is started. Enough space is set aside to hold the instruction pages (known as the *text pages*, or the *text segment*) and the *data segment* of the process. In this way, a process will generally not run out of swap space while it executes. When a process starts, its text is paged in from the file system. These pages are written out to swap as needed, and read back in from there, so the file system is consulted once for each text page. Pages from the data segment are read in from the file system, or created (if they are uninitialized), and are written to swap space and paged back in as needed. One optimization is that processes with the same text pages (for instance, two users running the same editor) share these pages, both in physical memory and in swap space. If one process has already paged the blocks in from the file system and out to the swap space, the other process can get them directly from the swap space.

Two per-process *swap maps* are used by the kernel to track swap space use. The text segment is a fixed size, so its swap space is allocated in 512K chunks, except for the last chunk which holds the remainder of the pages, in 1K increments (Figure 12.7).

The data segment swap map is more complicated, because the data segment can grow over time. The map is of fixed size, but contains swap addresses for blocks of varying size. Given index i, a block pointed to by swap map entry i is of size $2^i * 16K$, to a maximum of 2 megabytes. This data structure is shown in Figure 12.8. (The block size minimum and maximum are variable, and can be changed at system reboot.) If the last block is full and the process grows, a new block is allocated in swap space. This scheme results in small processes using only small blocks and

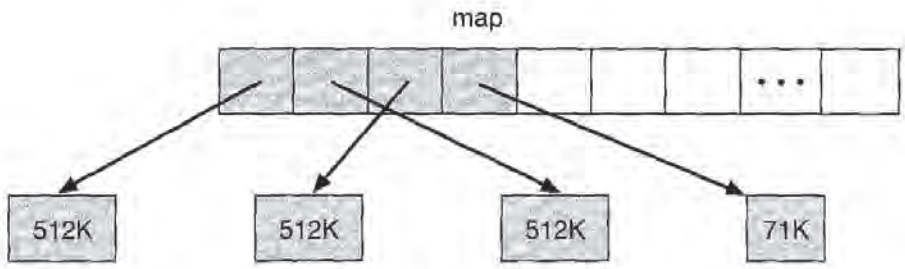


Figure 12.7 4.3BSD text segment swap map.

minimizing fragmentation. The blocks of large processes can be quickly found, and the swap map remains rather small.

In Solaris 1 (SunOS 4), some changes were made to standard UNIX methods to improve efficiency and reflect technological changes. When a process executes, text segment pages are brought in from the file system, accessed in main memory, and thrown away if selected for pageout. It is more efficient to read a page back in from the file system than to write it and then read it from swap space.

More changes were made in Solaris 2. The largest change is in the time of swap-space allocation. Solaris 2 allocates swap space only when a page is forced out of memory, not when the page is allocated. This is an improvement on modern systems which have more main memory than older systems, and tend to page less.

12.5 ■ Disk Reliability

Disks used to be the least reliable component of a system. They still have relatively high failure rates, and their failure causes a loss of data and significant downtime while the disk is replaced and data are restored. The recovery in case of a disk crash may take hours, as backup copies of the data on tape are transferred to the disk. Under normal circumstances, these restored data are not a precise image of the disk when it crashed. Backups are usually performed daily or weekly, meaning that any changes to the data since the last backup are lost if a disk crashes. Improving disk speed and reliability is therefore an important topic in current research.

Several improvements in disk-use techniques have been proposed. These methods involve the use of multiple disks working cooperatively. To improve speed, researchers have developed *disk striping* (or interleaving), a technique now in use on a few systems. A group of disks is treated as one storage unit, with each block broken into several subblocks. Each subblock is stored on a separate disk. The time required to transfer one block into memory decreases drastically, since the disks

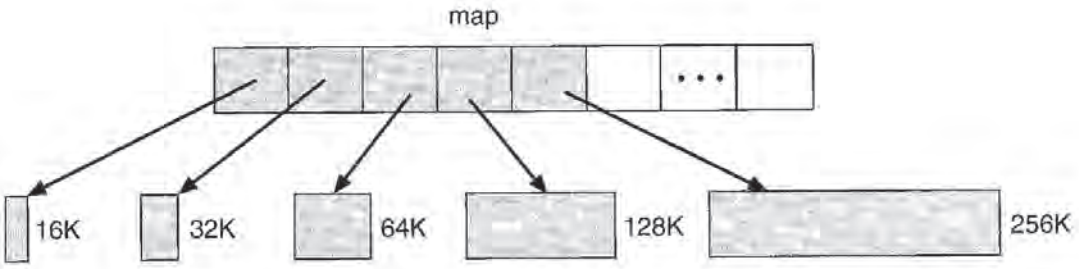


Figure 12.8 4.3BSD data segment swap map.

transfer their subblocks in parallel, fully utilizing their *I/O bandwidth*, or transfer capacity. This decrease is especially notable if the disks are *synchronized* such that there is no latency between the subblock-access on each drive. This advantage is also magnified if many small, inexpensive disks are used in place of a few, large, expensive disks. The access rate improves because seeks occur in parallel and more data can be transferred after the seek.

This idea was the basis for the development of redundant arrays of inexpensive disks (*RAID*), which improves performance, especially the price–performance ratio, and provides for the duplication of data to improve reliability. Redundancy can be organized in various ways, with differing performance and cost efficiencies.

The simplest RAID organization, called *mirroring* or *shadowing*, consists of keeping a duplicate copy of each disk. This solution is, of course, costly, because twice as many disks are used to store the same number of data. In the most complex RAID organization, *block interleaved parity*, data are written to each disk in the array in a block unit, just as in more normal configurations. However, an extra block of parity data is written. This parity block is the parity of all the equivalent blocks on each disk in the array. For instance, if there are eight disks in the array, then sector 0 of disks 1 through 7 have their parity computed and stored on disk 8. The computation takes place on a bit level for each byte, just as memory parity is computed. If one disk crashes, one of the data bits is essentially erased (an *erasure error*), but can be recomputed from the other data bits plus the parity. Thus, a single disk crash no longer results in loss of data (although multiple simultaneous crashes do have that result).

Of course, use of RAID improves speed by virtue of the use of multiple disks and controllers. This gain is less than that realized from pure disk striping, however, since the parity block must also be read or written for each block access, and the parity computation must be performed. We can decrease this overhead by distributing the parity over all disks, rather than setting aside one parity disk. Since the tradeoff between speed and reliability is a reasonable one, RAID use is becoming more common, with

many vendors either researching or producing RAID hardware and software. It has been shown that, with an array of 100 inexpensive disks and 10 parity disks, the mean time to data loss (*MTDL*) of the array is 90 years, compared to the 2 or 3 years of standard, large, expensive disks.

12.6 ■ Stable-Storage Implementation

In Chapter 6, we introduced the concept of a write-ahead log, which required the availability of stable storage. Information residing in stable storage is *never* lost. To implement such a storage, we need to replicate the needed information on more than nonvolatile storage media (usually disk) with independent failure modes. We also need to update the information in a controlled manner to ensure that failure during data transfer does not damage the needed information. For the remainder of this section, we will discuss the issue of how the storage media can be protected from failure during data transfer.

Block transfer between memory and disk storage can result in:

- **Successful completion:** The transferred information arrived safely at its destination.
- **Partial failure:** A failure occurred in the midst of transfer and the destination block has incorrect information.
- **Total failure:** The failure occurred sufficiently early during the transfer so that the destination block remains intact.

We require that if a data-transfer failure occurs, the system detects it and invokes a recovery procedure to restore the block to a consistent state. To do so, the system must maintain two physical blocks for each logical block. An output operation is executed as follows:

1. Write the information onto the first physical block.
2. When the first write successfully completes, write the same information onto the second physical block.
3. The output is complete only after the second write successfully completes.

During recovery from a failure, each pair of physical blocks is examined. If both are the same and no detectable error exists, then no further actions are necessary. If one block contains a detectable error, then we replace its content with the value of the second block. If both blocks contain no detectable error, but they differ in content, then we replace the

content of the first block with the value of the second. This recovery procedure ensures that a write to stable storage either succeeds completely or results in no change. The attempt to write to stable storage succeeds only if all copies are written.

This procedure can be extended easily to allow the use of an arbitrarily large number of copies of each block of stable storage. Although a large number of copies further reduces the probability of a failure, it is usually reasonable to simulate stable storage with only two copies.

12.7 ■ Summary

Disk systems are the major secondary-storage I/O device on most computers. Requests for disk I/O are generated both by the file system and by virtual-memory systems. Each request specifies the address on the disk to be referenced, which is in the form of a block number. The lower levels of the file-system manager convert this address into the hardware-level partition, cylinder, surface, and sector number.

Moving-head disk-scheduling algorithms are designed to minimize total head movement; they include the FCFS, SSTF, SCAN, C-SCAN, LOOK, and C-LOOK algorithms.

Disk efficiency is still an active area of research. For instance, Hewlett-Packard .s experimenting with making disk controllers smarter, and the University of California at Berkeley has a log-based file system that uses the extreme approach of treating the disk as though it were a sequential-access device, greatly reducing disk seeks and fragmentation.

In order to reduce fragmentation the system can back up and restore the entire disk or partition. The blocks are read from their scattered locations, and the restore writes them back more contiguously. Some systems (such as MS-DOS) have utilities that will scan the disk and then move blocks around to decrease the fragmentation. The performance increases realized from these techniques can be large, but the system is generally unusable while the techniques operate.

The operating system also must manage the disk blocks. First, a disk must be formatted to create the blocks on the raw hardware. Then boot blocks are added to store the system's boot-strap code. Finally, when a block becomes corrupt, the system must have a way to logically replace that block with another, good block.

Because efficient swap space use is a key to performance, systems frequently bypass the file system structure and use disk blocks more directly to store memory pages which do not fit within physical memory. Some systems just use a file within the file system for this, but there can be a performance penalty. Other systems allow the user or system administrator to make the decision by providing both options.

The write-ahead log scheme requires the availability of stable storage. To implement such a storage, we need to replicate the needed information on more than one nonvolatile storage media (usually disk) with independent failure modes. We also need to update the information in a controlled manner to ensure that failure during data transfer does not damage the needed information.

■ Exercises

12.1 All the disk-scheduling disciplines, except FCFS scheduling, are not truly fair (starvation may occur).

a. Explain why this assertion is true.

b. Describe a scheme to ensure fairness.

c. Explain why fairness is an important goal in a time-sharing system.

12.2 Suppose that the head of a moving-head disk with 200 tracks, numbered 0 to 199, is currently serving a request at track 143 and has just finished a request at track 125. The queue of requests is kept in the FIFO order:

86, 147, 91, 177, 94, 150, 102, 175, 130.

What is the total number of head movements needed to satisfy these requests for the following disk-scheduling algorithms?

a. FCFS scheduling

b. SSTF scheduling

c. SCAN scheduling

d. LOOK scheduling

e. C-SCAN scheduling

12.3 Write a monitor-type program (see Chapter 6) for disk scheduling using the SCAN and C-SCAN disk-scheduling algorithms.

12.4 When the average queue length is small, all the disk-scheduling algorithms reduce to FCFS scheduling. Explain why this assertion is true.

12.5 Compare the throughput of C-SCAN and SCAN scheduling, assuming a uniform distribution of requests.

12.6 Is disk scheduling, other than FCFS scheduling, useful in a single-user environment? Explain your answer.

12.7 SSTF scheduling tends to favor mid-range cylinders over the innermost and outermost cylinders. Explain why this assertion is true.

12.8 Requests are not usually uniformly distributed. For example, the cylinders on which the file directory structures reside are accessed more frequently than are most files. Suppose that you know that 50 percent of the requests are for a small fixed number of cylinders.

a. Which of the scheduling algorithms discussed in this chapter would be best?

b. Can you suggest a new scheduling algorithm for this case? If you can, describe your algorithm.

12.9 Why is latency optimization usually not employed in disk scheduling? How would the standard algorithms (FCFS, SSTF, SCAN, and C-SCAN scheduling) be modified to include latency optimization?

12.10 How would the use of a RAM disk affect your selection of a disk-scheduling algorithm? What factors would you need to consider?

12.11 Why is it important to try to balance file system I/O among the disks and controllers on a system in a multitasking environment?

12.12 What are the tradeoffs involved in re-reading code pages from the file system rather than using swap space to store them?

12.13 Is there any way to implement truly stable storage? Explain your answer.

Bibliographic Notes

Discussions concerning magnetic-disk technology were presented by Freedman [1983], and Harker et al. [1981]. Optical disks were covered by Kenville [1982], Fujitani [1984], O'Leary and Kitts [1985], Gait [1988], and Olsen and Kenly [1989]. Ammon et al. [1985] discussed a high-speed, large-capacity "jukebox" optical-disk system. Discussions of floppy disks were offered by Pechura and Schoeffler [1983] and Sarisky [1983]. Discussions of redundant arrays of inexpensive disks (RAID) were presented by Patterson et al. [1988] and Chen and Patterson [1990]. Disk system architectures for high performance computing are discussed by Katz et al. [1989]. Nelson and Cheng [1992] compared the performance of IPI and SCSI disks and controllers in a real system.

A complete survey of all the various disk-scheduling algorithms, and a comparative analysis of these algorithms was presented by Teorey and Pinkerton [1972]. The comparison was done using simulations, and it was

recommended that either SCAN or C-SCAN scheduling should be used, depending on the load. Wilhelm [1976] and Hofri [1980] compared the FCFS and the SSTF seek disk-scheduling algorithms. A continuum of disk-scheduling algorithms was presented by Geist and Daniel [1987].

The Loge project [English and Stepanov 1992] experimented with adding intelligence to the disk controller. The log-based file system, which works hard to make disk access sequential, was discussed in Rosenblum and Ousterhout [1991]. Both of these projects are the topics of continuing research.

PART FOUR

PROTECTION AND SECURITY

Protection mechanisms provide controlled access by limiting the types of file access that can be made by the various users. Protection must also be available to ensure that besides the files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system.

Protection is provided by a mechanism that controls the access of programs, processes, or users to the resources defined by a computer system. This mechanism must provide a means for specification of the controls to be imposed, together with a means of enforcement.

The security system prevents unauthorized access to a system, and ensuing malicious destruction or alteration of data. Security ensures the authentication of users of the system to protect the integrity of the information stored in the system (both data and code), as well as the physical resources of the computer system.

CHAPTER 13

PROTECTION

The various processes in an operating system must be protected from one another's activities. For that purpose, various mechanisms exist that can be used to ensure that the files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system.

Protection refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system. This mechanism must provide a means for specification of the controls to be imposed, together with some means of enforcement. We distinguish between protection and *security*, which is a measure of confidence that the integrity of a system and its data will be preserved. Security assurance is a much broader topic than is protection, and we address it in Chapter 14.

In this chapter, we examine the problem of protection in great detail, and develop a unifying model for implementing protection.

13.1 ■ Goals of Protection

As computer systems have become more sophisticated and pervasive in their applications, the need to protect their integrity has also grown. Protection was originally conceived as an adjunct to multiprogramming operating systems, so that untrustworthy users might safely share a common logical name space, such as a directory of files, or share a common physical name space, such as memory. Modern protection concepts have evolved to increase the reliability of any complex system that makes use of shared resources.

There are several reasons for providing protection. Most obvious is the need to prevent mischievous, intentional violation of an access restriction by a user. Of more general importance, however, is the need to ensure that each program component active in a system uses system resources only in ways consistent with the stated policies for the uses of these resources. This requirement is an absolute one for a reliable system.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by a subsystem that is malfunctioning. An unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides means to distinguish between authorized and unauthorized usage.

The role of protection in a computer system is to provide a *mechanism* for the enforcement of the *policies* governing resource use. These policies can be established in a variety of ways. Some are fixed in the design of the system, whereas others are formulated by the management of a system. Still others are defined by the individual users to protect their own files and programs. A protection system must have the flexibility to enforce a variety of policies that can be declared to it.

Policies for resource use may vary, depending on the application, and they may be subject to change over time. For these reasons, protection can no longer be considered solely as a matter of concern to the designer of an operating system. It should also be available as a tool for the applications programmer, so that resources created and supported by an applications subsystem can be guarded against misuse. In this chapter, we describe the protection mechanisms the operating system should provide, so that an application designer can use them in designing her own protection software.

One important principle is the separation of *policy* from *mechanism*. Mechanisms determine how something will be done. In contrast, policies decide *what* will be done. The separation of policy and mechanism is important for flexibility. Policies are likely to change from place to place or time to time. In the worst case, every change in policy would require a change in the underlying mechanism. General mechanisms are more desirable, because a change in a policy would then require the modification of only some system parameters or tables.

13.2 ■ Domain of Protection

A computer system is a collection of processes and objects. By *objects*, we mean both hardware objects (such as the CPU, memory segments, printers, disks, and tape drives), and software objects (such as files, programs, and semaphores). Each object has a unique name that differentiates it from all

other objects in the system, and each can be accessed only through well-defined and meaningful operations. Objects are essentially *abstract data types*.

The operations that are possible may depend on the object. For example, a CPU can only be executed on. Memory segments can be read and written, whereas a card reader can only be read. Tape drives can be read, written, and rewound. Data files can be created, opened, read, written, closed, and deleted; program files can be read, written, executed, and deleted.

Obviously, a process should be allowed to access only those resources it has been authorized to access. Furthermore, at any time, it should be able to access only those resources that it currently requires to complete its task. This requirement, commonly referred to as the *need-to-know* principle, is useful in limiting the amount of damage a faulty process can cause in the system. For example, when process p invokes procedure A, the procedure should be allowed to access only its own variables and the formal parameters passed to it; it should not be able to access all the variables of process p. Similarly, consider the case where process p invokes a compiler to compile a particular file. The compiler should not be able to access any arbitrary files, but only a well-defined subset of files (such as the source file, listing file, and so on) related to the file to be compiled. Conversely, the compiler may have private files used for accounting or optimization purposes, which process p should not be able to access.

13.2.1 Domain Structure

To facilitate this scheme, we introduce the concept of a *protection domain*. A process operates within a protection domain, which specifies the resources that the process may access. Each domain defines a set of objects and the types of operations that may be invoked on each object. The ability to execute an operation on an object is an *access right*. A *domain* is a collection of access rights, each of which is an ordered pair *<object-name, rights-set>*. For example, if domain D has the access right <file F, {read,write}>, then a process executing in domain D can both read and write file F; it cannot, however, perform any other operation on that object.

Domains do not need to be disjoint; they may share access rights. For example, in Figure 13.1, we have three domains: D_1, D_2, and D_3. The access right $<O_4$, {print}> is shared by both D_2 and D_3, implying that a process executing in either one of these two domains can print object O_4. Note that a process must be executing in domain D_1 to read and write object O_1. On the other hand, only processes in domain D_3 may execute object O_1.

The association between a process and a domain may be either static (if the set of resources available to a process is fixed throughout the latter's lifetime) or dynamic. As might be expected, the problems inherent in

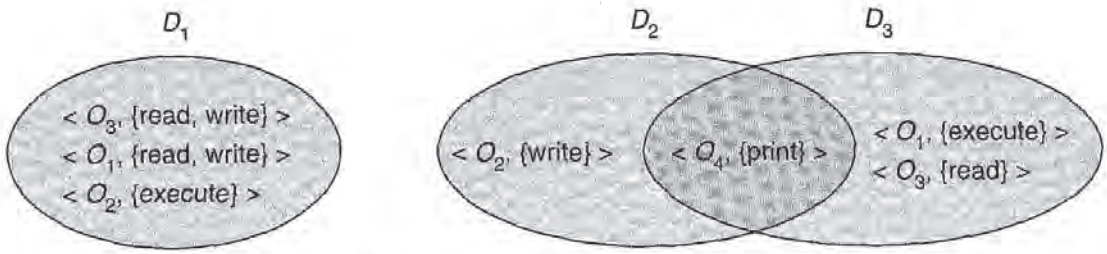


Figure 13.1 System with three protection domains.

establishing dynamic protection domains require more careful solution than do the simpler problems of the static case.

If the association between processes and domains is fixed, and we want to adhere to the need-to-know principle, then a mechanism must be available to change the content of a domain. A process may execute in two different phases. For example, it may need read access in one phase and write access in another. If a domain is static, we must define the domain to include both read and write access. However, this arrangement provides more rights than are needed in each of the two phases, since we have read access in the phase where we need only write access, and vice versa. Thus, the need-to-know principle is violated. We must allow the contents of a domain to be modified, so that it always reflects the minimum necessary access rights.

If the association is dynamic, a mechanism is available to allow a process to switch from one domain to another. We may also want to allow the content of a domain to be changed. If we cannot change the content of a domain, we can provide the same effect by creating a new domain with the changed content, and switching to that new domain when we want to change the domain content.

We note that a domain can be realized in a variety of ways:

- Each *user* may be a domain. In this case, the set of objects that can be accessed depends on the identity of the user. Domain switching occurs when the user is changed — generally when one user logs out and another user logs in.
- Each *process* may be a domain. In this case, the set of objects that can be accessed depends on the identity of the process. Domain switching corresponds to one process sending a message to another process, and then waiting for a response.
- Each *procedure* may be a domain. In this case, the set of objects that can be accessed corresponds to the local variables defined within the procedure. Domain switching occurs when a procedure call is made.

We shall discuss domain switching in greater detail later on in this chapter.

13.2.2 Examples

Consider the standard dual-mode (monitor-user mode) model of operating-system execution. When a process executes in monitor mode, it can execute privileged instructions and thus gain complete control of the computer system. On the other hand, if the process executes in user mode, it can invoke only nonprivileged instructions. Consequently, it can execute only within its predefined memory space. These two modes protect the operating system (executing in monitor domain) from the user processes (executing in user domain). In a multiprogrammed operating system, two protection domains are insufficient, since users also want to be protected from one another. Therefore, a more elaborate scheme is needed. We illustrate this scheme by examining two influential operating systems — UNIX and MULTICS — to see how these concepts have been implemented there.

13.2.2.1 UNIX

In the UNIX operating system, a domain is associated with the user. Switching the domain corresponds to changing the user identification temporarily. This change is accomplished through the file system as follows. An owner identification and a domain bit (known as the *setuid bit*) are associated with each file. When a user (with *user-id* = *A*) starts executing a file owned by *B*, whose associated domain bit is *off*, the *user-id* of the process is set to *A*. When the setuid bit is *on*, the user-id is set to that of the owner of the file: *B*. When the process exits, this temporary user-id change ends.

There are other common methods used to change domains in operating systems in which user-ids are used for domain definition, because almost all systems need to provide such a mechanism. This mechanism is used when an otherwise privileged facility needs to be made available to the general user population. For instance, it might be desirable to allow users to access a network without letting them write their own networking programs. In such a case, on a UNIX system, the setuid bit on a networking program would be set, causing the user-id to change when the program is run. The user-id would change to that of a user with network access privilege (such as "root," the most powerful user-id). One problem with this method is that if a user manages to create a file with user-id "root" and with its setuid bit on, that user can become "root" and do anything and everything on the system. The setuid mechanism is discussed further in Chapter 19.

An alternative to this method used on other operating systems is to place privileged programs in a special directory. The operating system would be designed to change the user-id of any program run from this directory, either to the equivalent of "root" or to the user-id of the owner of the directory. This eliminates the setuid problem of secret setuid

programs, because all these programs are in one location. This method is less flexible than that used in UNIX, however.

Even more restrictive, and thus more protective, are systems that simply do not allow a change of user-id. In these instances, special techniques must be used to allow users access to privileged facilities. For instance, a *daemon* process may be started at boot time and run as a special user-id. Users then run a separate program, which sends requests to this process whenever they need to use the facility. This method is used by the TOPS-20 operating system.

In any of these systems, great care must be taken in writing privileged programs. Any oversight or carelessness can result in a total lack of protection on the system. Generally, these programs are the first to be attacked by people trying to break into a system; unfortunately, the attackers are frequently successful. For instance, there are many instances where security has been breached on UNIX systems because of the setuid feature. We discuss security in Chapter 14.

13.2.2.2 MULTICS

In the MULTICS system, the protection domains are organized hierarchically into a ring structure. Each ring corresponds to a single domain (Figure 13.2). The rings are numbered from 0 to 7. Let D_i and D_j be any two domain rings. If $j < i$, then D_i is a subset of D_j. That is, a process executing in domain D_j has more privileges than does a process executing in domain D_i. A process executing in domain D_0 has the most privileges. If there are only two rings, this scheme is equivalent to the monitor-user mode of

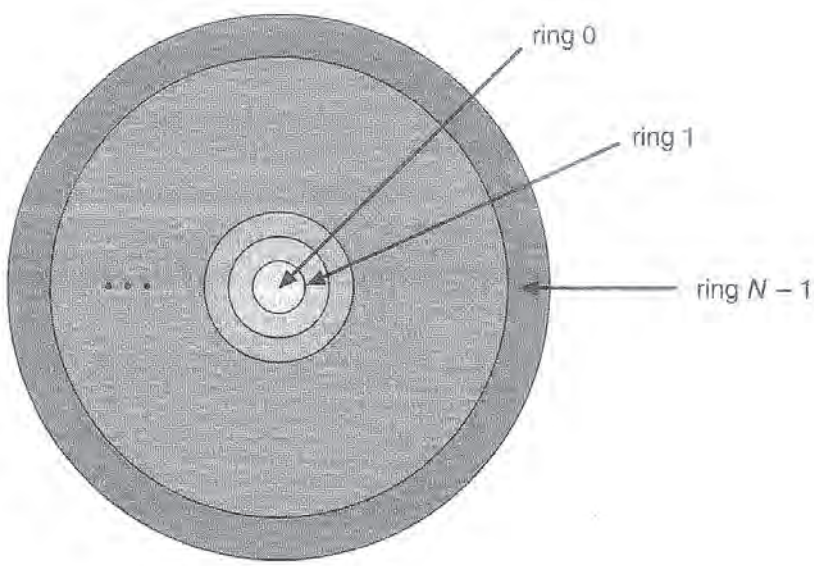


Figure 13.2 MULTICS ring structure.

execution, where monitor mode corresponds to D_0 and user mode corresponds to D_1.

MULTICS has a segmented address space; each segment is a file. Each segment is associated with one of the rings. A segment description includes an entry that identifies the ring number. In addition, it includes three access bits to control reading, writing, and execution. The association between segments and rings is a policy decision with which we are not concerned in this book. With each process, a *current-ring-number* counter is associated, identifying the ring in which the process is executing currently. When a process is executing in ring i, it cannot access a segment associated with ring j, $j < i$. It can, however, access a segment associated with ring k, $k \geq i$. The type of access, however, is restricted, according to the access bits associated with that segment.

Domain switching in MULTICS occurs when a process crosses from one ring to another by calling a procedure in a different ring. Obviously, this switch must be done in a controlled manner; otherwise, a process can start executing in ring 0, and no protection will be provided. To allow controlled domain switching, we modify the ring field of the segment descriptor to include the following:

- **Access bracket:** A pair of integers, $b1$ and $b2$, such that $b1 \leq b2$.
- **Limit:** An integer $b3$, such that $b3 > b2$.
- **List of gates:** Identifies the entry points (gates) at which the segments may be called.

If a process executing in ring i calls a procedure (segment) with access bracket $(b1,b2)$, then the call is allowed if $b1 \leq i \leq b2$, and the current ring number of the process remains i. Otherwise, a trap to the operating system occurs, and the situation is handled as follows:

- If $i < b1$, then the call is allowed to occur, because we have a transfer to a ring (domain) with fewer privileges. However, if parameters are passed that refer to segments in a lower ring (that is, segments that are not accessible to the called procedure), then these segments must be copied into an area that can be accessed by the called procedure.
- If $i > b2$, then the call is allowed to occur only if $b3$ is less than or equal to i, and the call has been directed to one of the designated entry points in the list of gates. This scheme allows processes with limited access rights to call procedures in lower rings that have more access rights, but only in a carefully controlled manner.

The main disadvantage of the ring (hierarchical) structure is that it does not allow us to enforce the need-to-know principle. In particular, if an

object must be accessible in domain D_j but not accessible in domain D_i, then we must have $j < i$. But this requirement means that every segment accessible in D_i is also accessible in D_j.

The MULTICS protection system is generally more complex and less efficient than are those used in current operating systems. If protection interferes with the ease of use of the system, or significantly decreases system performance, then its use must be weighed carefully against the purpose of the system. For instance, it would be reasonable to have a complex protection system on a computer used by a university to process students' grades, and also used by students for class work. A similar protection system would not be suited to a computer being used for number crunching in which performance is of utmost importance. It would therefore be of benefit to separate the mechanism from the protection policy, allowing the same system to have complex or simple protection depending on the needs of its users. To separate mechanism from policy, we require more general models of protection.

13.3 ■ Access Matrix

Our model of protection can be viewed abstractly as a matrix, called an *access matrix*. The rows of the access matrix represent domains, and the columns represent objects. Each entry in the matrix consists of a set of access rights. Because objects are defined explicitly by the column, we can omit the object name from the access right. The entry **access**(i,j) defines the set of operations that a process, executing in domain D_i, can invoke on object O_j.

To illustrate these concepts, we consider the access matrix shown in Figure 13.3. There are four domains and four objects: three files (F_1, F_2, F_3), and one laser printer. When a process executes in domain D_1, it can

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Figure 13.3 Access matrix.

read files F_1 and F_3. A process executing in domain D_4 has the same privileges as it does in domain D_1, but in addition, it can also write onto files F_1 and F_3. Note that the laser printer can be accessed only by a process executing in domain D_2.

The access-matrix scheme provides us with the mechanism for specifying a variety of policies. The mechanism consists of implementing the access matrix and ensuring that the semantic properties we have outlined indeed hold. More specifically, we must ensure that a process executing in domain D_i can access only those objects specified in row i, and then only as allowed by the access-matrix entries.

Policy decisions concerning protection can be implemented by the access matrix. The policy decisions involve which rights should be included in the (i,j)th entry. We must also decide the domain in which each process executes. This last policy is usually decided by the operating system.

The users normally decide the contents of the access-matrix entries. When a user creates a new object O_j, the column O_j is added to the access matrix with the appropriate initialization entries, as dictated by the creator. The user may decide to enter some rights in some entries in column j and other rights in other entries, as needed.

The access matrix provides an appropriate mechanism for defining and implementing strict control for both the static and dynamic association between processes and domains. When we switch a process from one domain to another, we are executing an operation (switch) on an object (the domain). We can control domain switching by including domains among the objects of the access matrix. Similarly, when we change the content of the access matrix, we are performing an operation on an object: the access matrix. Again, we can control these changes by including the access matrix itself as an object. Actually, since each entry in the access

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Figure 13.4 Access matrix of Figure 13.3 with domains as objects.

matrix may be modified individually, we must consider each entry in the access matrix as an object to be protected.

Now, we need to consider only the operations that are possible on these new objects (domains and the access matrix), and decide how we want processes to be able to execute these operations.

Processes should be able to switch from one domain to another. Domain switching from domain D_i to domain D_j is allowed to occur if and only if the access right *switch* $\in$ **access**(i, j). Thus, in Figure 13.4, a process executing in domain D_2 can switch to domain D_3 or to domain D_4. A process in domain D_4 can switch to D_1, and one in domain D_1 can switch to domain D_2.

Allowing controlled change to the contents of the access-matrix entries requires three additional operations: **copy**, **owner**, and **control**.

The ability to copy an access right from one domain (row) of the access matrix to another is denoted by an asterisk (*) appended to the access right. The *copy* right allows the copying of the access right only within the column (that is, for the object) for which the right is defined. For example, in Figure 13.5(a), a process executing in domain D_2 can copy the read operation into any entry associated with file F_2. Hence, the access matrix of Figure 13.5(a) can be modified to the access matrix shown in Figure 13.5(b).

domain \ object	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

domain \ object	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Figure 13.5 Access matrix with *copy* rights.

There are two variants to this scheme:

1. A right is copied from **access**(i, j) to **access**(k, j); it is then removed from **access**(i, j); this action is a *transfer* of a right, rather than a copy.
2. Propagation of the *copy* right may be limited. That is, when the right R^* is copied from **access**(i, j) to **access**(k, j), only the right R (not R^*) is created. A process executing in domain D_k cannot further copy the right R.

A system may select only one of these three *copy* rights, or it may provide all three by identifying them as separate rights: *copy*, *transfer*, and *limited copy*.

The *copy* right allows a process to copy some rights from an entry in one column to another entry in the same column. We also need a mechanism to allow addition of new rights and removal of some rights. The *owner* right controls these operations. If **access**(i, j) includes the *owner* right, then a process executing in domain D_i can add and remove any right in any entry in column j. For example, in Figure 13.6(a), domain D_1 is the

domain \ object	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write*
D_3	execute		

(a)

domain \ object	F_1	F_2	F_3
D_1	owner execute		
D_2		owner read* write*	read* owner write*
D_3		write	write

(b)

Figure 13.6 Access matrix with *owner* rights.

owner of F_1, and thus can add and delete any valid right in column F_1. Similarly, domain D_2 is the owner of F_2 and F_3, and thus can add and remove any valid right within these two columns. Thus, the access matrix of Figure 13.6(a) can be modified to the access matrix shown in Figure 13.6(b).

The *copy* and *owner* rights allow a process to change the entries in a column. A mechanism is also needed to change the entries in a row. The *control* right is applicable to only domain objects. If **access**(i, j) includes the *control* right, then a process executing in domain D_i can remove any access right from row j. For example, suppose that, in Figure 13.4, we include the *control* right in **access**(D_2, D_4). Then, a process executing in domain D_2 could modify domain D_4, as shown in Figure 13.7.

Although the *copy* and *owner* rights provide us with a mechanism to limit the propagation of access rights, they do not, however, provide us with the appropriate tools for preventing the propagation of information (that is, disclosure of information). The problem of guaranteeing that no information initially held in an object can migrate outside of its execution environment is called the *confinement problem*. This problem is in general unsolvable (see Bibliographic Notes for references).

These operations on the domains and the access matrix are not in themselves particularly important. What is more important is that they illustrate the ability of the access-matrix model to allow the implementation and control of dynamic protection requirements. New objects and new domains can be created dynamically and included in the access-matrix model. However, we have shown only that the basic mechanism is here; the policy decisions concerning which domains are to have access to which objects in which ways must be made by the system designers and users.

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Figure 13.7 Modified access matrix of Figure 13.4.

13.4 ■ Implementation of Access Matrix

How can the access matrix be implemented effectively? In general, the matrix will be sparse; that is, most of the entries will be empty. Although there are data-structure techniques available for representing sparse matrices, they are not particularly useful for this application, because of the way in which the protection facility is used.

13.4.1 Global Table

The simplest implementation of the access matrix is a global table consisting of a set of ordered triples <domain, object, rights-set>. Whenever an operation M is executed on an object O_j within domain D_i, the global table is searched for a triple $<D_i, O_j, R_k>$, where $M \in R_k$. If this triple is found, the operation is allowed to continue; otherwise, an exception (error) condition is raised. This implementation suffers from several drawbacks. The table is usually large and thus cannot be kept in main memory, so additional I/O is needed. Virtual-memory techniques are often used for managing this table. In addition, it is difficult to take advantage of special groupings of objects or domains. For example, if a particular object can be read by everyone, it must have a separate entry in every domain.

13.4.2 Access Lists for Objects

Each column in the access matrix can be implemented as an access list for one object, as described in Section 10.4.2. Obviously, the empty entries can be discarded. The resulting list for each object consists of ordered pairs <domain, rights-set>, which define all domains with a nonempty set of access rights for that object.

This approach can be extended easily to define a list plus a *default* set of access rights. When an operation M on an object O_j is attempted in domain D_i, we search the access list for object O_j, looking for an entry $<D_i, R_k>$ with $M \in R_k$. If the entry is found, we allow the operation; if it is not, we check the default set. If M is in the default set, we allow the access. Otherwise, access is denied and an exception condition occurs. Note that, for efficiency, we may check the default set first, and then search the access list.

13.4.3 Capability Lists for Domains

Rather than associating the columns of the access matrix with the objects as access lists, we can associate each row with its domain. A *capability list* for a domain is a list of objects together with the operations allowed on

those objects. An object is often represented by its physical name or address, called a *capability*. To execute operation M on object O_j, the process executes the operation M, specifying the capability (pointer) for object O_j as a parameter. Simple *possession* of the capability means that access is allowed.

The capability list is associated with a domain, but is never directly accessible to a process executing in that domain. Rather, the capability list is itself a protected object, maintained by the operating system and accessed by the user only indirectly. Capability-based protection relies on the fact that the capabilities are never allowed to migrate into any address space directly accessible by a user process (where they could be modified). If all capabilities are secure, the object they protect is also secure against unauthorized access.

Capabilities were originally proposed as a kind of secure pointer, to meet the need for resource protection that was foreseen as multiprogrammed computer systems came of age. The idea of an inherently protected pointer (from the point of view of a user of a system) provides a foundation for protection that can be extended up to the applications level.

To provide inherent protection, we must distinguish capabilities from other kinds of objects, and must interpret them by an abstract machine on which higher-level programs run. Capabilities are usually distinguished from other data in one of two ways:

- Each object has a *tag* to denote its type as either a capability or as accessible data. The tags themselves must not be directly accessible by an applications program. Hardware or firmware support may be used to enforce this restriction. Although only 1 bit is necessary to distinguish between capabilities and other objects, more bits are often used. This extension allows all objects to be tagged with their types by the hardware. Thus, the hardware can distinguish integers, floating-point numbers, pointers, Booleans, characters, instructions, capabilities, and uninitialized values by their tags.
- Alternatively, the address space associated with a program can be split into two parts. One part is accessible to the program and contains the program's normal data and instructions. The other part, containing the capability list, is accessible by only the operating system. A segmented memory space (Section 8.6) is useful to support this approach.

Several capability-based protection systems have been developed; we describe them briefly in Section 13.6. The Mach operating system also uses a version of capability-based protection; it is described in great detail in Chapter 20.

13.4.4 A Lock–Key Mechanism

The *lock–key scheme* is a compromise between access lists and capability lists. Each object has a list of unique bit patterns, called *locks*. Similarly, each domain has a list of unique bit patterns, called *keys*. A process executing in a domain can access an object only if that domain has a key that matches one of the locks of the object.

As capability lists must be, the list of keys for a domain must be managed by the operating system on behalf of the domain. Users are not allowed to examine or modify the list of keys (or locks) directly.

13.4.5 Comparison

Access lists correspond directly to the needs of the users. When a user creates an object, she can specify which domains can access the object, as well as the operations allowed. However, because access-rights information for a particular domain is not localized, determining the set of access rights for each domain is difficult. In addition, every access to the object must be checked, requiring a search of the access list. In a large system with long access lists, this search can be time-consuming.

Capability lists do not correspond directly to the needs of the users; they are useful, however, for localizing information for a particular process. The process attempting access must present a capability for that access. Then, the protection system needs only to verify that the capability is valid. Revocation of capabilities, however, may be inefficient (Section 13.5).

The lock–key mechanism is a compromise between these two schemes. The mechanism can be both effective and flexible, depending on the length of the keys. The keys can be passed freely from domain to domain. In addition, access privileges may be effectively revoked by the simple technique of changing some of the keys associated with the object (Section 13.5).

Most systems use a combination of access lists and capabilities. When a process first tries to access an object, the access list is searched. If access is denied, an exception condition occurs. Otherwise, a capability is created and is attached to the process. Additional references use the capability to demonstrate swiftly that access is allowed. After the last access, the capability is destroyed. This strategy is used in the MULTICS system and in the CAL system; these systems use both access lists and capability lists.

As an example, consider a file system. Each file has an associated access list. When a process opens a file, the directory structure is searched to find the file, access permission is checked, and buffers are allocated. All this information is recorded in a new entry in a file table associated with the process. The operation returns an index into this table for the newly

opened file. All operations on the file are made by specification of the index into the file table. The entry in the file table then points to the file and its buffers. When the file is closed, the file-table entry is deleted. Since the file table is maintained by the operating system, it cannot be corrupted by the user. Thus, the only files that the user can access are those that have been opened. Since access is checked when the file is opened, protection is ensured. This strategy is used in the UNIX system.

Note that the right to access *must* still be checked on each access, and the file-table entry has a capability only for the allowed operations. If a file is opened for reading, then a capability for read access is placed in the file-table entry. If an attempt is made to write onto the file, the system determines this protection violation by comparing the requested operation with the capability in the file-table entry.

13.5 ■ Revocation of Access Rights

In a dynamic protection system, it may sometimes be necessary to revoke access rights to objects that are shared by different users. Various questions about revocation may arise:

- **Immediate versus delayed:** Does revocation occur immediately, or is it delayed? If revocation is delayed, can we find out when it will take place?
- **Selective versus general:** When an access right to an object is revoked, does it affect *all* the users who have an access right to that object, or can we specify a select group of users whose access rights should be revoked?
- **Partial versus total:** Can a subset of the rights associated with an object be revoked, or must we revoke all access rights for this object?
- **Temporary versus permanent:** Can access be revoked permanently (that is, the revoked access right will never again be available), or can access be revoked and later be obtained again?

With an access-list scheme, revocation is easy. The access list is searched for the access right(s) to be revoked, and they are deleted from the list. Revocation is immediate, and can be general or selective, total or partial, and permanent or temporary.

Capabilities, however, present a much more difficult revocation problem. Since the capabilities are distributed throughout the system, we must find them before we can revoke them. There are several different schemes for implementing revocation for capabilities, including the following:

- **Reacquisition**. Periodically, capabilities are deleted from each domain. If a process wants to use a capability, it may find that that capability has been deleted. The process may then try to reacquire the capability. If access has been revoked, the process will not be able to reacquire the capability.

- **Back-pointers**. A list of pointers is maintained with each object, pointing to all capabilities associated with that object. When revocation is required, we can follow these pointers, changing the capabilities as necessary. This scheme has been adopted in the MULTICS system. It is quite general, although it is a costly implementation.

- **Indirection**. The capabilities do not point to the objects directly, but instead point indirectly. Each capability points to a unique entry in a global table, which in turn points to the object. We implement revocation by searching the global table for the desired entry and deleting it. When an access is attempted, the capability is found to point to an illegal table entry. Table entries can be reused for other capabilities without difficulty, since both the capability and the table entry contain the unique name of the object. The object for a capability and its table entry must match. This scheme was adopted in the CAL system. It does not allow selective revocation.

- **Keys**. A key is a unique bit pattern that can be associated with each capability. This key is defined when the capability is created, and it can be neither modified nor inspected by the process owning that capability. A *master key* associated with each object can be defined or replaced with the **set-key** operation. When a capability is created, the current value of the master key is associated with the capability. When the capability is exercised, its key is compared to the master key. If the keys match, the operation is allowed to continue; otherwise, an exception condition is raised. Revocation replaces the master key with a new value by the **set-key** operation, invalidating all previous capabilities for this object.

  Note that this scheme does not allow selective revocation, since only one master key is associated with each object. If we associate a list of keys with each object, then selective revocation can be implemented. Finally, we can group all keys into one global table of keys. A capability is valid only if its key matches some key in the global table. We implement revocation by removing the matching key from the table. With this scheme, a key can be associated with several objects, and several keys can be associated with each object, providing maximum flexibility.

  In key-based schemes, the operations of defining keys, inserting them into lists, and deleting them from lists should not be available to all users. In particular, it would be reasonable to allow only the owner

of an object to set the keys for that object. This choice, however, is a policy decision that the protection system can implement, but should not define.

13.6 ■ Capability-Based Systems

In this section, we briefly survey two capability-based protection systems. These systems vary in their complexity and in the type of policies that can be implemented on them. Neither of them is widely used, but they are interesting proving grounds for protection theories.

13.6.1 Hydra

Hydra is a capability-based protection system that provides considerable flexibility. The system provides a fixed set of possible access rights that are known to and interpreted by the system. These rights include such basic forms of access as the right to read, write, or execute a memory segment. In addition, the system provides the means for a user (of the protection system) to declare additional rights. The interpretation of user-defined rights is performed solely by the user's program, but the system provides access protection for the use of these rights, as well as for the use of system-defined rights. The facilities provided by this system are interesting, and constitute a significant development in protection technology.

Operations on objects are defined procedurally. The procedures that implement such operations are themselves a form of object, and are accessed indirectly by capabilities. The names of user-defined procedures must be identified to the protection system if it is to deal with objects of the user-defined type. When the definition of an object is made known to Hydra, the names of operations on the type become *auxiliary rights*. Auxiliary rights can be described in a capability for an instance of the type. For a process to perform an operation on a typed object, the capability it holds for that object must contain the name of the operation being invoked among its auxiliary rights. This restriction enables discrimination of access rights to be made on an instance-by-instance and process-by-process basis.

Another interesting concept is *rights amplification*. This scheme allows certification of a procedure as *trustworthy* to act on a formal parameter of a specified type, on behalf of any process that holds a right to execute the procedure. The rights held by a trustworthy procedure are independent of, and may exceed, the rights held by the calling process. However, it is necessary neither to regard such a procedure as universally trustworthy (the procedure is not allowed to act on other types, for instance), nor to extend trustworthiness to any other procedures or program segments that might be executed by a process.

Amplification is useful in allowing implementation procedures access to the representation variables of an abstract data type. If a process holds a capability to a typed object *A*, for instance, this capability may include an auxiliary right to invoke some operation *P*, but would not include any of the so-called kernel rights, such as read, write, or execute, on the segment that represents *A*. Such a capability gives a process a means of indirect access (through the operation *P*) to the representation of *A*, but only for specific purposes.

On the other hand, when a process invokes the operation *P* on an object *A*, the capability for access to *A* may be amplified as control passes to the code body of *P*. This amplification may be necessary, to allow *P* the right to access the storage segment representing *A*, to implement the operation that *P* defines on the abstract data type. The code body of *P* may be allowed to read or to write to the segment of *A* directly, even though the calling process cannot. On return from *P*, the capability for *A* is restored to its original, unamplified state. This case is a typical one in which the rights held by a process for access to a protected segment must change dynamically, depending on the task to be performed. The dynamic adjustment of rights is performed to guarantee consistency of a programmer-defined abstraction. Amplification of rights can be stated explicitly in the declaration of an abstract type to the Hydra system.

When a user passes an object as an argument to a procedure, it may be necessary to ensure that the procedure cannot modify the object. We can implement this restriction readily by passing an access right that does not have the modification (write) right. However, if amplification may occur, the right to modify may be reinstated. Thus, the user-protection requirement can be circumvented. In general, of course, a user may trust that a procedure indeed performs its task correctly. This assumption, however, is not always correct, because of hardware or software errors. Hydra solves this problem by restricting amplifications.

The procedure call mechanism of Hydra was designed as a direct solution to the *problem of mutually suspicious subsystems*. This problem is defined as follows. Suppose that a program is provided that can be invoked as a service by a number of different users (for example, a sort routine, a compiler, a game). When users invoke this service program, they take the risk that the program will malfunction and will either damage the given data, or will retain some access right to the data to be used (without authority) later. Similarly, the service program may have some private files (for accounting purposes, for example) that should not be accessed directly by the calling user program. Hydra provides mechanisms for directly dealing with this problem.

A Hydra subsystem is built on top of its protection kernel and may require protection of its own components. A subsystem interacts with the kernel through calls on a set of kernel-defined primitives that defines access rights to resources defined by the subsystem. Policies for use of

these resources by user processes can be defined by the subsystem designer, but are enforceable by use of the standard access protection afforded by the capability system.

A programmer can make direct use of the protection system, after acquainting himself with its features in the appropriate reference manual. Hydra provides a large library of system-defined procedures that can be called by user programs. A user of the Hydra system would explicitly incorporate calls on these system procedures into the code of his programs, or would use a program translator that had been interfaced to Hydra.

13.6.2 Cambridge CAP System

A different approach to capability-based protection has been taken in the design of the Cambridge CAP system. CAP's capability system is simpler and superficially less powerful than that of Hydra. However, closer examination shows that it too can be used to provide secure protection of user-defined objects. In CAP, there are two kinds of capabilities. The ordinary kind is called a *data capability*. It can be used to provide access to objects, but the only rights provided are the standard read, write, or execute of the individual storage segments associated with the object. Data capabilities are interpreted by microcode in the CAP machine.

A so-called *software capability* is protected by, but not interpreted by, the CAP microcode. It is interpreted by a *protected* (that is, a privileged) procedure, which may be written by an applications programmer as part of a subsystem. A particular kind of rights amplification is associated with a protected procedure. When executing the code body of such a procedure, a process temporarily acquires the rights to read or write the contents of a software capability itself. This specific kind of rights amplification corresponds to an implementation of the **seal** and **unseal** primitives on capabilities (see Bibliographic Notes for references). Of course, this privilege is still subject to type verification to ensure that only software capabilities for a specified abstract type are allowed to be passed to any such procedure. Universal trust is not placed in any code other than the CAP machine's microcode.

The interpretation of a software capability is left completely to the subsystem, through the protected procedures it contains. This scheme allows a variety of protection policies to be implemented. Although a programmer can define her own protected procedures (any of which might be incorrect), the security of the overall system cannot be compromised. The basic protection system will not allow an unverified, user-defined, protected procedure access to any storage segments (or capabilities) that do not belong to the protection environment in which it resides. The most serious consequence of an insecure protected procedure is a protection breakdown of the subsystem for which that procedure has responsibility.

The designers of the CAP system have noted that the use of software capabilities has allowed them to realize considerable economies in formulating and implementing protection policies commensurate with the requirements of abstract resources. However, a subsystem designer who wants to make use of this facility cannot simply study a reference manual, as is the case with Hydra. Instead, he must learn the principles and techniques of protection, since the system provides him with no library of procedures to be used.

13.7 ■ Language-Based Protection

To the degree that protection is provided in existing computer systems, it has usually been achieved through the device of an operating-system kernel, which acts as a security agent to inspect and validate each attempt to access a protected resource. Since comprehensive access validation is potentially a source of considerable overhead, either we must give it hardware support to reduce the cost of each validation, or we must accept that the system designer may be inclined to compromise the goals of protection. It is difficult to satisfy all these goals if the flexibility to implement various protection policies is restricted by the support mechanisms provided or if protection environments are made larger than necessary to secure greater operational efficiency.

As operating systems have become more complex, and particularly as they have attempted to provide higher-level user interfaces, the goals of protection have become much more refined. In this refinement, we find that the designers of protection systems have drawn heavily on ideas that originated in programming languages and especially on the concepts of abstract data types and objects. Protection systems are now concerned not only with the identity of a resource to which access is attempted but also with the functional nature of that access. In the newest protection systems, concern for the function to be invoked extends beyond a set of system-defined functions, such as standard file access methods, to include functions that may be user-defined as well.

Policies for resource use may also vary, depending on the application, and they may be subject to change over time. For these reasons, protection can no longer be considered as a matter of concern to only the designer of an operating system. It should also be available as a tool for use by the applications designer, so that resources of an applications subsystem can be guarded against tampering or the influence of an error.

At this point, programming languages enter the picture. Specifying the desired control of access to a shared resource in a system is making a declarative statement about the resource. This kind of statement can be integrated into a language by an extension of its typing facility. When

protection is declared along with data typing, the designer of each subsystem can specify its requirements for protection, as well as its need for use of other resources in a system. Such a specification should be given directly as a program is composed, and in the language in which the program itself is stated. There are several significant advantages to this approach:

1. Protection needs are simply declared, rather than programmed as a sequence of calls on procedures of an operating system.
2. Protection requirements may be stated independently of the facilities provided by a particular operating system.
3. The means for enforcement do not need to be provided by the designer of a subsystem.
4. A declarative notation is natural because access privileges are closely related to the linguistic concept of data type.

There is a variety of techniques that can be provided by a programming language implementation to enforce protection, but any of these must depend on some degree of support from an underlying machine and its operating system. For example, suppose a language were used to generate code to run on the Cambridge CAP system. On this system, every storage reference made on the underlying hardware occurs indirectly through a capability. This restriction prevents any process from accessing a resource outside of its protection environment at any time. However, a program may impose arbitrary restrictions on how a resource may be used during execution of a particular code segment by any process. We can implement such restrictions most readily by using the software capabilities provided by CAP. A language implementation might provide standard, protected procedures to interpret software capabilities that would realize the protection policies that could be specified in the language. This scheme puts policy specification at the disposal of the programmers, while freeing them from the details of implementing its enforcement.

Even if a system does not provide a protection kernel as powerful as those of Hydra or CAP, there are still mechanisms available for implementing protection specifications given in a programming language. The principal distinction is that the *security* of this protection will not be as great as that supported by a protection kernel, because the mechanism must rely on more assumptions about the operational state of the system. A compiler can separate references for which it can certify that no protection violation could occur from those for which a violation might be possible, and can treat them differently. The security provided by this form of protection rests on the assumption that the code generated by the compiler will not be modified prior to or during its execution.

What, then, are the relative merits of enforcement based solely on a kernel, as opposed to enforcement provided largely by a compiler?

- **Security:** Enforcement by a kernel provides a greater degree of security of the protection system itself than does the generation of protection-checking code by a compiler. In a compiler-supported scheme, security rests on correctness of the translator, on some underlying mechanism of storage management that protects the segments from which compiled code is executed, and, ultimately, on the security of files from which a program is loaded. Some of these same considerations also apply to a software-supported protection kernel, but to a lesser degree, since the kernel may reside in fixed physical storage segments and may be loaded from only a designated file. With a tagged capability system, in which all address computation is performed either by hardware or by a fixed microprogram, even greater security is possible. Hardware-supported protection is also relatively immune to protection violations that might occur as a result of either hardware or system software malfunction.
- **Flexibility:** There are limits to the flexibility of a protection kernel in implementing a user-defined policy, although it may supply adequate facilities for the system to provide enforcement for its own policies. With a programming language, protection policy can be declared and enforcement provided as needed by an implementation. If a language does not provide sufficient flexibility, it can be extended or replaced, with less perturbation of a system in service than would be caused by the modification of an operating-system kernel.
- **Efficiency:** The greatest efficiency is obtained when enforcement of protection is supported directly by hardware (or microcode). Insofar as software support is required, language-based enforcement has the advantage that static access enforcement can be verified off-line at compile time. Also, since the enforcement mechanism can be tailored by an intelligent compiler to meet the specified need, the fixed overhead of kernel calls can often be avoided.

In summary, the specification of protection in a programming language allows the high-level description of policies for the allocation and use of resources. A language implementation can provide software for protection enforcement when automatic hardware-supported checking is unavailable. In addition, it can interpret protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system.

One way of making protection available to the application program is through the use of *software capability* that could be used as an object of computation. Inherent in this concept is the idea that certain program

components might have the privilege of creating or examining these software capabilities. A capability-creating program would be able to execute a primitive operation that would seal a data structure, rendering the latter's contents inaccessible to any program components that did not hold either the seal or the unseal privileges. They might copy the data structure, or pass its address to other program components, but they could not gain access to its contents. The reason for introducing such software capabilities is to bring a protection mechanism into the programming language. The only problem with the concept as proposed is that the use of the **seal** and **unseal** operations takes a procedural approach to specifying protection. A nonprocedural or declarative notation seems a preferable way to make protection available to the applications programmer.

What is needed is a safe, dynamic access-control mechanism for distributing capabilities to system resources among user processes. If it is to contribute to the overall reliability of a system, the access-control mechanism should be safe to use. If it is to be useful in practice, it should also be reasonably efficient. This requirement has led to the development of a number of language constructs that allow the programmer to declare various restrictions on the use of a specific managed resource (see the Bibliographic Notes for appropriate references). These constructs provide mechanisms for three functions:

1. Distributing capabilities safely and efficiently among customer processes: In particular, mechanisms ensure that a user process will use the managed resource only if it was granted a capability to that resource.

2. Specifying the type of operations that a particular process may invoke on an allocated resource (for example, a reader of a file should be allowed only to read the file, whereas a writer should be able both to read and to write): It should not be necessary to grant the same set of rights to every user process, and it should be impossible for a process to enlarge its set of access rights, except with the authorization of the access control mechanism.

3. Specifying the order in which a particular process may invoke the various operations of a resource (for example, a file must be opened before it can be read): It should be possible to give two processes different restrictions on the order in which they can invoke the operations of the allocated resource.

The incorporation of protection concepts into programming languages, as a practical tool for system design, is at present in its infancy. It is likely that protection will become a matter of greater concern to the designers of new systems with distributed architectures and increasingly stringent

requirements on data security. Then, the importance of suitable language notations in which to express protection requirements will be recognized more widely.

13.8 ■ Summary

Computer systems contain many objects. These objects need to be protected from misuse. Objects may be hardware (such as memory, CPU time, or I/O devices) or software (such as files, programs, and abstract data types). An access right is permission to perform an operation on an object. A domain is a set of access rights. Processes execute in domains and may use any of the access rights in the domain to access and manipulate objects.

The access matrix is a general model of protection. The access matrix provides a mechanism for protection without imposing a particular protection policy on the system or its users. The separation of policy and mechanism is an important design property.

The access matrix is sparse. It is normally implemented either as access lists associated with each object, or as capability lists associated with each domain. We can include dynamic protection in the access-matrix model by considering domains and the access matrix itself as objects.

Real systems are much more limited, and tend to provide protection for only files. UNIX is representative, providing read, write, and execution protection separately for the owner, group, and general public for each file. MULTICS uses a ring structure in addition to file access. Hydra, the Cambridge CAP system, and Mach are capability systems that extend protection to user-defined software objects.

■ Exercises

13.1 What are the main differences between capability lists and access lists?

13.2 A Burroughs B7000/B6000 MCP file can be tagged as sensitive data. When such a file is deleted, its storage area is overwritten by some random bits. For what purpose would such a scheme be useful?

13.3 In a ring-protection system, level 0 has the greatest access to objects and level n (greater than zero) has fewer access rights. The access rights of a program at a particular level in the ring structure are considered as a set of capabilities. What is the relationship between the capabilities of a domain at level j and a domain at level i to an object (for $j > i$)?

13.4 Consider a computer system in which "computer games" can be played by students only between 10 P.M. and 6 A.M., by faculty members between 5 P.M. and 8 A.M., and by the computer center staff at all times. Suggest a scheme for implementing this policy efficiently.

13.5 The RC 4000 system (and other systems) have defined a tree of processes (called a process tree) such that all the descendants of a process are given resources (objects) and access rights by their ancestors only. Thus, a descendant can never have the ability to do anything that its ancestors cannot do. The root of the tree is the operating system, which has the ability to do anything. Assume the set of access rights was represented by an access matrix, A. $A(x,y)$ defines the access rights of process x to object y. If x is a descendant of z, what is the relationship between $A(x,y)$ and $A(z,y)$ for an arbitrary object y?

13.6 What hardware features are needed in a computer system for efficient capability manipulation? Can these be used for memory protection?

13.7 Consider a computing environment where a unique number is associated with each process and each object in the system. Suppose that we allow a process with number n to access an object with number m only if $n > m$. What type of protection structure do we have?

13.8 What protection problems may arise if a shared stack is used for parameter passing?

13.9 Consider a computing environment where a process is given the privilege of accessing an object only n times. Suggest a scheme for implementing this policy.

13.10 If all the access rights to an object are deleted, the object can no longer be accessed. At this point, the object should also be deleted, and the space it occupies should be returned to the system. Suggest an efficient implementation of this scheme.

13.11 What is the need-to-know principle? Why is it important for a protection system to adhere to this principle?

13.12 Why is it difficult to protect a system in which users are allowed to do their own I/O?

13.13 Capability lists are usually kept within the address space of the user. How does the system ensure that the user cannot modify the contents of the list?

Bibliographic Notes

The access-matrix model of protection between domains and objects was developed by Lampson [1969, 1971]. Popek [1974], Saltzer and Schroeder [1975] provided excellent surveys on the subject of protection. Harrison et al. [1976] used a formal version of this model to enable them to prove mathematically properties of a protection system.

The concept of a capability evolved from Iliffe's and Jodeit's *codewords*, which were implemented in the Rice University computer [Iliffe and Jodeit 1962]. The term *capability* was introduced by Dennis and Van Horn [1966].

The Hydra system was described by Wulf et al. [1981]. The CAP system was described by Needham and Walker [1977]. Organick [1972] discussed the MULTICS ring protection system.

Revocation was discussed by Redell and Fabry [1974], Cohen and Jefferson [1975], and Ekanadham and Bernstein [1979]. The principle of separation of policy and mechanism was advocated by the designer of Hydra [Levin et al. 1975]. The confinement problem was first discussed by Lampson [1973], and was further examined by Lipner [1975].

The use of higher-level languages for specifying access control was suggested first by Morris [1973], who proposed the use of the *seal* and *unseal* operation discussed in Section 13.7. Kieburtz and Silberschatz [1978, 1983], and McGraw and Andrews [1979], proposed various language constructs for dealing with general dynamic resource-management schemes. Jones and Liskov [1978] considered the problem of how a static access-control scheme can be incorporated in a programming language that supports abstract data types.

CHAPTER 14

SECURITY

Protection, as we have discussed it in Chapter 13, is strictly an *internal* problem: How do we provide controlled access to programs and data stored in a computer system? Security, on the other hand, requires not only an adequate protection system, but also consideration of the *external* environment within which the system operates. Internal protection is not useful if the operator's console is exposed to unauthorized personnel, or if files (stored, for example, on tapes and disks) can simply be removed from the computer system and taken to a system with no protection. These security problems are essentially management, not operating-system, problems.

The information stored in the system (both data and code), as well as the physical resources of the computer system, need to be protected from unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. In this chapter, we examine the ways in which information may be misused or intentionally made inconsistent. We then present mechanisms to guard against this occurrence.

14.1 ■ The Security Problem

In Chapter 13, we discussed various mechanisms that the operating system can provide (with an appropriate aid from the hardware) that will allow users to protect their resources (usually programs and data). These mechanisms work well as long as the users of the system do not try to circumvent the intended use of and access to these resources. Unfortunately, in reality, this situation is seldom realized. When it is not,

security comes into play. We say that a system is secure if its resources are utilized and accessed as intended under all circumstances. Unfortunately, it is not generally possible to achieve total security. Nonetheless, mechanisms must be available to make security breaches a rare occurrence, rather than the norm.

Security violations (misuse) of the system can be categorized as being either intentional (malicious) or accidental. It is easier to protect against accidental misuse than to protect against malicious misuse. Among the forms of malicious access are the following:

- Unauthorized reading of data (theft of information)
- Unauthorized modification of data
- Unauthorized destruction of data

Absolute protection of the system from malicious abuse is not possible, but the cost to the perpetrator can be made sufficiently high to deter most if not all attempts to access, without proper authority, the information residing in the system.

To protect the system, we must take security measures at two levels:

- **Physical:** The site or sites containing the computer systems must be physically secured against armed or surreptitious entry by intruders.
- **Human:** Users must be authorized carefully to reduce the chance of any such user giving access to an intruder in exchange for a bribe or other favors.

Security at both levels must be maintained to ensure operating system security. A weakness at a high level of security (physical or human) allows circumvention of strict low-level (operating system) security measures.

It is worthwhile, in many applications, to devote a considerable effort to the security of the computer system. Large commercial systems containing payroll or other financial data are inviting targets to thieves. Systems that contain data pertaining to corporate operations may be of interest to unscrupulous competitors. Furthermore, loss of such data, whether via accident or fraud, can seriously impair the ability of the corporation to function.

On the other hand, the system hardware must provide protection (as discussed in Chapter 13) to allow for the implementation of security features. For instance, MS-DOS and Macintosh OS provide little security because the hardware for which they were originally designed for did not provide memory or I/O protection. Now that the hardware has become sufficiently sophisticated to provide protection, the designers of these

operating systems are struggling to add security. Unfortunately, adding a feature to a functional system is a much more difficult and challenging task than is designing and implementing the feature before the system is built.

In the remainder of this chapter, we shall address security at the operating-system level. Security at the physical and human levels, although important, is far beyond the scope of this text. Security within the operating system is implemented at several levels, ranging from passwords for access to the system to the isolation of concurrent processes running within the system. The file system also provides some degree of protection.

14.2 ■ Authentication

A major security problem for operating systems is the *authentication* problem. The protection system depends on an ability to identify the programs and processes that are executing. This ability, in turn, eventually rests on our power to identify each user of the system. A user normally identifies himself. How do we determine if a user's identity is authentic? Generally, authentication is based on some combination of three sets of items: user possession (a key or card), user knowledge (a user identifier and password), and a user attribute (fingerprint, retina pattern, or signature).

The most common approach to authenticating a user identity is the use of user *passwords*. When the user identifies herself by user-id or account name, she is asked for a password. If the user-supplied password matches the password stored in the system, the system assumes that the user is legitimate.

Passwords are often used to protect objects in the computer system, in the absence of more complete protection schemes. They can be considered a special case of either keys or capabilities. For instance, a password could be associated with each resource (such as a file). Whenever a request is made to use the resource, the password must be given. If the password is correct, access is granted. Different passwords may be associated with different access rights. For example, different passwords may be used for reading, appending, and updating a file.

Although there are some problems associated with the use of passwords, they are nevertheless extremely common, because they are easy to understand and use. The problems with passwords are related to the difficulty of keeping a password secret. Passwords can be compromised by being guessed, accidentally exposed, or illegally transferred from an authorized user to an unauthorized one, as we show next.

There are two common ways to guess a password. One is for the intruder (either human or program) to know the user or have information about the user. All too frequently, people use obvious information (such

as the name of their cat or spouse) as their password. The other way is brute-force; trying all possible combinations of letters, numbers, and punctuation until the password is found. Short passwords do not leave enough choices to prevent a password from being guessed by repeated trials. For example, a four-decimal password provides only 10,000 variations. On the average, guessing 5000 times would produce the password. If a program could be written that would try a password every 1 millisecond, it would then take only about 5 seconds to guess a password. Longer passwords are less susceptible to being guessed by enumeration, and systems that allow uppercase and lowercase letters, numbers, and all punctuation characters to be used in passwords make the task of guessing the password much more difficult. Of course users must take advantage of the large password space and not just use lower-case letters.

The failure of password security due to exposure can result from visual or electronic monitoring. An intruder could look over the shoulder of a user when the user is logging in, and can learn the password easily. Alternatively, anyone with access to the network on which a computer resides could seamlessly add a network monitor, allowing her to watch all data being transferred on the network, including user-ids and passwords. Exposure is a particularly severe problem if the password is written down where it can be read or lost. As we shall see, some systems force the user to select hard-to-remember or long passwords. Taken to extreme, this can cause a user to record the password, providing much worse security than when the system allows easy passwords!

The final method of password compromise is the result of human nature. Most computer installations have the rule that users are not allowed to share accounts. This rule is sometimes implemented for accounting reasons, but often it is used to aid in security. For instance, if one user-id was shared by several users, and a security breach occurred from that user-id, then it is impossible to know who was using that user-id at the time, or even if it was one of the authorized users. With one user per user-id, the user could be questioned directly about the use of that account. Sometimes, users break account-sharing rules to help out friends or circumvent accounting, and this behavior can result in a system being accessed by unauthorized users, possibly harmful ones.

Passwords can be either system-generated or user-selected. System-generated passwords may be difficult to remember, and thus may be commonly written down. User-selected passwords, however, are often easy to guess (the user's name or favorite car, for example). At some sites, administrators occasionally check user passwords and notify the users if the password is too short or easy to guess. Some systems also *age* passwords, forcing users to change them at regular intervals (every month, for instance). This method is not foolproof either, since users may easily toggle between two passwords.

Several variants on the simple password scheme can be used. For example, the password can be changed frequently. In the extreme, the password is changed for each session. A new password is selected (either by the system or by the user) at the end of *each* session, and that password must be used for the next session. Note that, even if a password is misused, it can be used only once, and its use prevents the legitimate user from using it. Consequently, the legitimate user discovers the security violation at the next session, when he uses a now-invalid password. Steps can then be taken to repair the broached security.

Another approach is to have a set of paired passwords. When a session begins, the system randomly selects and presents one part of a password pair; the user must supply the other part. This approach can be generalized to the use of an algorithm as a password. The algorithm might be an integer function, for example. The system selects a random integer and presents it to the user. The user applies the function and replies with the result of the function. The system also applies the function. If the two results match, access is allowed.

One problem with all these approaches is the difficulty of keeping the password (or list of password pairs, or algorithms) secret. The UNIX system uses a variant of the algorithmic password to avoid the necessity of keeping its password list secret. Each user has a password. The system contains a function that is very difficult (the designers hope impossible) to invert, but simple to compute. That is, given a value x, it is easy to compute the function value $f(x)$. Given a function value $f(x)$, however, it is impossible to compute x. This function is used to encode all passwords. Only the encoded passwords are stored. When a user presents a password, it is encoded and compared against the stored encoded password. Even if the stored encoded password is seen, it cannot be decoded, so the password cannot be determined. Thus, the password file does not need to be kept secret.

The flaw in this method is that the system no longer has control over the passwords. Although the passwords are encrypted, anyone with a copy of the password file may run fast encryption routines against it, encrypting each word in a dictionary, for instance, and comparing the results against the passwords. If the user has selected a password that is also a word in the dictionary, the password is cracked. On sufficiently fast computers, or even on clusters of slow computers, such a comparison may only take a few hours. For this reason, new versions of UNIX hide the password entries. On current UNIX systems, it is a good idea for users to use combination passwords, such as two dictionary words separated by a punctuation character. Such passwords do not easily succumb to algorithmic guessing techniques because the result is a very large space (number of words in the dictionary squared) which needs to be encrypted to try to match such passwords. To avoid the dictionary encryption method, some systems disallow the use of dictionary words as passwords.

14.3 ■ Program Threats

In an environment where a program written by one user may be used by another user, there is an opportunity for misuse, which may result in unexpected behavior. Below, we describe two common methods for achieving this.

14.3.1 Trojan Horse

Many systems have mechanisms for allowing programs written by users to be executed by other users. If these programs are executed in a domain that provides the access rights of the executing user, they may misuse these rights. Inside a text-editor program, for example, there may be code to search the file to be edited for certain keywords. If any are found, the entire file may be copied to a special area accessible to the creator of the text editor. A code segment that misuses its environment is called a *Trojan horse*. The Trojan-horse problem is exacerbated by long search paths (such as are common on UNIX systems). The search path lists the set of directories to search when an ambiguous program name is given. The path is searched for a file of that name and the file is executed. All the directories in the search path must be secure, or a Trojan horse could be slipped into the user's path and executed accidentally.

For instance, consider the use of the "." character in a search path. The "." tells the shell to include the current directory in the search. Thus, if a user has "." in her search path, has set her current directory to a friend's directory, and enters the name of a normal system command, the command may be executed from the friend's directory instead. The program would run within the user's domain, allowing the program to do anything that the user is allowed to do, including deleting the user's files for instance.

14.3.2 Trap Door

The designer of a program or system might leave a hole in the software that only he is capable of using. This type of security breach was shown in the movie "War Games." For instance, the code might check for a specific user identifier or password and circumvent normal security procedures. There have been cases of programmers being arrested for embezzling from banks by including rounding errors in their code, and having the occasional half-cent credited to their accounts. This account crediting can add up to quite a large amount of money, considering the number of transactions a bank executes!

A clever trap door could be included in a compiler. The compiler could generate standard object code as well as a trap door, regardless of the source code being compiled. This activity is particularly nefarious,

since a search of the source code of the program would not reveal any problems. Only the source code of the compiler would contain the information. Trap doors pose a difficult problem because, to detect them, we would have to analyze all the source code for all components of a system. Given that software systems may consist of millions of lines of code, this analysis is not done frequently.

14.4 ■ System Threats

Most operating systems provide a means for processes to spawn other processes. In such an environment, it is possible to create a situation where operating-system resources and user files are misused. The two most common methods for doing this are worms and viruses, which are discussed below.

14.4.1 Worms

A *worm* is a process that uses the spawn mechanism to clobber system performance. The worm spawns copies of itself, using up system resources and perhaps locking out system use by all other processes. On computer networks, worms are particularly potent, since they may reproduce themselves among systems and thus shut down the entire network. Such an event occurred in 1988 to UNIX systems on the worldwide *Internet* network, causing millions of dollars of lost system and programmer time.

The Internet links thousands of government, academic, research, and industrial computers internationally and serves as the infrastructure for electronic exchange of scientific information. At the close of the work day of November 2, 1988, Robert Tappan Morris, Jr., a first-year Cornell graduate student, unleashed a worm program on one or more hosts connected to the Internet. Targeting Sun Microsystems' Sun 3 workstations and VAX computers running variants of 4 BSD UNIX, the worm quickly spread over great distances and within a few hours of its release had consumed system resources to the point of bringing down the infected machines.

Although Morris designed the self-replicating program for rapid reproduction and distribution, features of the UNIX networking environment provided the means to propagate the worm throughout the system. It is likely that Morris chose for initial infection an Internet host left open for and accessible to outside users. From there, the worm program exploited flaws in the UNIX operating system's security routines and took advantage of UNIX utilities that simplify resource sharing in local area networks to gain unauthorized access to thousands of other connected sites. Morris' methods of attack are outlined next.

The worm was made up of two programs, a *grappling hook* (also called *bootstrap* or *vector*) program and the main program. Named *l1.c*, the grappling hook consisted of 99 lines of C code compiled and run on each machine it accessed. Once established on the system under attack, the grappling hook connected to the machine where it originated and uploaded a copy of the main worm onto the "hooked" system (see Figure 14.1). The main program proceeded to search for other machines to which the newly infected system could connect easily. In these actions, Morris exploited a UNIX networking utility, *rsh*, for easy remote task execution. By setting up special files that list host–login name pairs, users can omit entering a password each time they access a remote account on the paired list. The worm searched these special files for site names that would allow remote execution without a password. Where remote shells were established, the worm program was uploaded and began executing anew.

The attack via remote access was one of three infection methods built into the worm. The other two methods involved operating system bugs in the UNIX *finger* and *sendmail* programs. The *finger* utility functions as an electronic telephone directory; the command

finger username@sitename

returns a person's real and login names, along with other information that the user may have provided, such as office and home addresses and telephone numbers, research plan, or their birthdays. *Finger* runs as a background process (daemon) at each BSD site and responds to queries throughout the Internet. The point vulnerable to malicious entry involved reading input without checking bounds for overflow. Morris' program

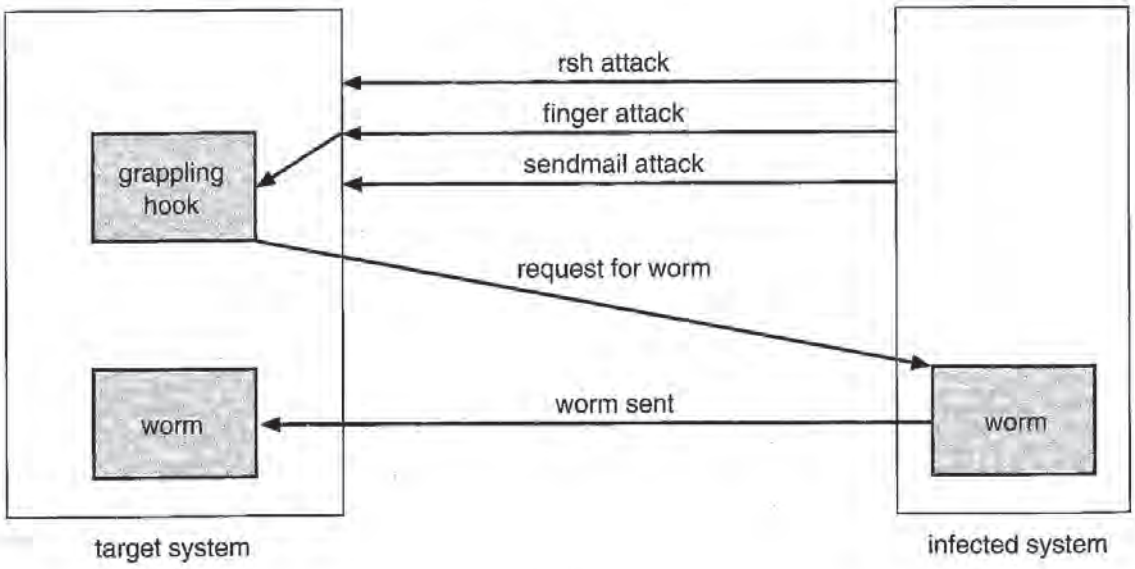


Figure 14.1 The Morris Internet worm.

queried *finger* with a 536-byte string crafted to exceed the buffer allocated for input and to overwrite the stack frame. Instead of returning to the *main* routine it was in before Morris' call, the *finger* daemon was routed to a procedure within the invading 536-byte string now residing on the stack. The new procedure executed */bin/sh*, which, if successful, provided the worm a remote shell on the machine under attack.

The bug exploited in *sendmail* also involved utilizing a daemon process for malicious entry. *Sendmail* routes electronic mail in a network environment. Debugging code in the utility permits testers to verify and display the state of the mail system. The debugging option is useful to system administrators and is often left on as a background process. Morris included in his attack arsenal a call to *debug*, which, instead of specifying a user address, as would be normal in testing, issued a set of commands that mailed and executed a copy of the grappling-hook program.

Once in place, the main worm proceeded in systematic attempts to discover user passwords. It began by trying simple cases of no password or passwords constructed of account–user name combinations, to comparisons with an internal dictionary of 432 favorite password choices, to the final stage of trying each word in the standard UNIX on-line dictionary as a possible password. This elaborate and efficient three-stage password-cracking algorithm enabled the worm to gain further access to other user accounts on the infected system. The worm then searched for *rsh* data files in these newly broken accounts. Any *rsh* entries were tried, and, as described previously, the worm could then gain access to user accounts on remote systems.

With each new access, the worm program searched for already active copies of itself. If it found one, the new copy exited, except for every seventh instance. Had the worm exited on all duplicate sightings, it might have remained undetected. Allowing every seventh duplicate to proceed (possibly to confound efforts to stop its spread by baiting with "fake" worms) created a wholesale infestation of Sun and VAX systems on the Internet.

The very features of the UNIX network environment that assisted the worm's propagation also helped to stop its advance. Ease of electronic communication, mechanisms to copy source and binary files to remote machines and access to both source code and human expertise allowed cooperative efforts to develop solutions to proceed apace. By the evening of the next day, November 3, methods of halting the invading program were circulated to system administrators via the Internet. Within days, specific software patches for the exploited security flaws were available.

One natural response is to question Morris' motives in unleashing the worm. The action has been characterized as both a harmless prank gone awry and a serious criminal offense. Based on the complexity of starting the attack, it is unlikely that the worm's release or the scope of its spread was unintentional. The worm program took elaborate steps to cover its

tracks and to repel efforts to stop its spread. Yet the program contained no code aimed at damaging or destroying the systems on which it ran. The author clearly had the expertise to include such commands; in fact, data structures were present in the bootstrap code that could have been used to transfer Trojan horse or virus programs (see Section 14.4.2). The actual behavior of the program may lead to interesting observations, but does not provide a sound basis for inferring motive. What is not open to speculation, however, is the legal outcome: A federal court convicted Morris and handed down a sentence of 3 years probation, 400 hours of community service, and a $10,000 fine. Morris' legal costs probably were in excess of $100,000.

14.4.2 Viruses

Another form of computer attack is a *virus*. Like worms, viruses are designed to spread into other programs and can wreak havoc in a system, including modifying or destroying files and causing system crashes and program malfunctions. Whereas a worm is structured as a complete, standalone program, a virus is a fragment of code embedded in a legitimate program. Viruses are a major problem for computer users, especially users of microcomputer systems. Multiuser computers, generally, are not prone to viruses because the executable programs are protected from writing by the operating system. Even if a virus does infect a program, its powers are limited because other aspects of the system are protected. Single-user systems have no such protections and, as a result, a virus has free run.

Viruses usually spread by users downloading viral programs from public bulletin boards or exchanging floppy disks containing an infection. A case from February 1992 involving two Cornell University students provides an illustration. The students had developed three Macintosh game programs with an embedded virus that they distributed to worldwide software archives via the Internet. The virus was discovered when a mathematics professor in Wales downloaded the games, and antivirus programs on his system alerted him to an infection. Some 200 other users had also downloaded the games. Although the virus was not designed to destroy data, it could spread to application files and cause such problems as long delays and program malfunctions. The authors were easy to trace, since the games had been mailed electronically from a Cornell account. New York state authorities arrested the students on misdemeanor charges of computer tampering and may have since filed additional charges.

In another incident, a programmer in California being divorced by his wife gave her a disk to load on a disputed computer. The disk contained a virus and erased all the files on the system. The husband was arrested and charged with destruction of property.

On occasion, upcoming viral infections are announced in high-profile media events. Such was the case with the Michelangelo virus, that was scheduled to erase infected hard disk files on March 6, 1992, the Renaissance artist's five hundred seventeenth birthday. Because of the extensive publicity surrounding the virus, most U.S. sites had located and destroyed the virus before it was activated, so it caused little or no damage. Such cases both alert the general public to and alarm them about the virus problem. Antivirus programs are currently very good sellers. Most commercial packages are effective against only particular, known viruses. They work by searching all the programs on a system for the specific pattern of instructions known to make up the virus. When they find a known pattern, they remove the instructions, "disinfecting" the program. These commercial packages have catalogs of hundreds of viruses for which they search.

The best protection against computer viruses is prevention, or the practice of *safe computing*. Purchasing unopened software from vendors and avoiding free or pirated copies from public sources or floppy-disk exchange is the safest route to preventing infection. However, even new copies of legitimate software applications are not immune to virus infection.

Another safeguard, while not preventing infection, does permit early detection. A user must begin by completely reformatting the hard disk, especially the boot sector, which is often targeted for viral attack. Only secure software is uploaded, and a checksum for each file is calculated. The checksum list must be then be kept free from unauthorized access. Following any system reboot, a program can recompute the checksums and compare them to the original list; any differences serve as a warning of possible infection. Because they usually work among systems, worms and viruses are generally considered to pose security, rather than protection, problems.

14.5 ■ Threat Monitoring

The security of a system can be improved by two management techniques. One is *threat monitoring*. The system can check for suspicious patterns of activity in an attempt to detect a security violation. A common example of this scheme is a time-sharing system that counts the number of incorrect passwords given when a user is trying to log in. More than a few incorrect attempts may signal an attempt to guess a password.

Another common technique is an *audit log*. An audit log simply records the time, user, and type of all accesses to an object. After security has been violated, the audit log can be used to determine how and when the problem occurred and perhaps the amount of damage done. This information can be useful, both for recovery from the violation and,

possibly, in the development of better security measures to prevent future problems. Unfortunately, logs can become large, and logging uses system resources which are then unavailable to the users.

Rather than log system activities, we can scan the system periodically for security holes. These scans can be done when the computer is relatively unused, and therefore have less effect than logging. Such a scan can check a variety of aspects of the system:

- Short or easy-to-guess passwords
- Unauthorized set-uid programs, if the system supports this mechanism
- Unauthorized programs in system directories
- Unexpected long-running processes
- Improper directory protections, on both user and system directories
- Improper protections on system data files, such as the password file, device drivers, or even the operating-system kernel itself
- Dangerous entries in the program search path (for example, Trojan horse as discussed in Section 14.3.1)
- Changes to system programs; we can detect unexpected changes by keeping a list of the checksum values of all system programs, and comparing this list against the current checksum values of the programs — checksums do not change unless the contents of the file have changed

Any problems found by a security scan can either be fixed automatically or be reported to the managers of the system.

Networked computers are much more susceptible to security attacks than are standalone systems. Rather than attacks from a known set of access points, such as directly connected terminals, we face attacks from an unknown and very large set of access points — a potentially severe security problem. To a lesser extent, systems connected to telephone lines via modems are also more exposed.

In fact, the U.S. federal government considers systems to be only as secure as is their most far-reaching connection. For instance, a top-security system may be accessed only from within a building also considered top-secure. The system loses the top-secure rating if any form of communication can occur outside that environment. Some government facilities take extreme security precautions. The connectors into which a terminal plugs to communicate with the secure computer are locked in a safe in the office when the terminal is not being used. A person needs to know a physical lock combination, as well as authentication information for the computer itself, to gain access to the computer.

14.6 ■ Encryption

The various provisions that an operating system may make for authorization may not offer sufficient protection for highly sensitive data. Moreover, as computer networks gain popularity, more sensitive (classified) information is being transmitted over channels where eavesdropping and message interception are possible. To keep such sensitive information secure, we need mechanisms to allow a user to protect data transferred over the network.

Encryption is one common method of protecting information transmitted over unreliable links. The basic mechanism works as follows:

1. The information (text) is *encrypted* (encoded) from its initial readable form (called *clear text*), to an internal form (called *cipher text*). This internal text form, although readable, does not make any sense.
2. The cipher text can be stored in a readable file, or transmitted over unprotected channels.
3. To make sense of the cipher text, the receiver must *decrypt* (decode) it back into clear text.

Even if the encrypted information is accessed by an unauthorized person, it will be useless unless it can be decoded. The main issue is the development of encryption schemes that are impossible (or at least exceedingly difficult) to break.

There is a variety of methods to accomplish this task. The most common ones provide a general encryption algorithm E, a general decryption algorithm D, and a secret key (or keys) to be supplied for each application. Let E_k and D_k denote the encryption and decryption algorithms, respectively, for a particular application with a key k. Then the encryption algorithm must satisfy the following properties for any message m:

1. $D_k(E_k(m)) = m$.
2. Both E_k and D_k can be computed efficiently.
3. The security of the system depends only on the secrecy of the key, and not on the secrecy of the algorithms E and D.

One such scheme, called the *Data Encryption Standard*, was recently adopted by the National Bureau of Standards. This scheme suffers from the *key-distribution* problem: Before communication can take place, the secret keys must be sent securely to both the sender and receiver. This task cannot be done effectively in a communication-network environment. A

solution to this problem is to use a *public key-encryption* scheme. Each user has both a public and a private key, and two users can communicate knowing only each other's public key.

An algorithm based on this concept follows. This algorithm is believed to be almost unbreakable. The public encryption key is a pair (e,n); the private key is a pair (d,n), where e, d, and n are positive integers. Each message is represented as an integer between 0 and $n - 1$. (A long message is broken into a series of smaller messages, each of which can be represented as such an integer.) The functions E and D are defined as

$$E(m) = m^e \textbf{ mod } n = C,$$
$$D(C) = C^d \textbf{ mod } n.$$

The main problem is choosing the encryption and decryption keys. The integer n is computed as the product of two large (100 or more digits) randomly chosen prime numbers p and q with

$$n = p \times q.$$

The value of d is chosen to be a large, randomly chosen integer relatively prime to $(p - 1) \times (q - 1)$. That is, d satisfies

$$\text{greatest common divisor}[d, (p - 1) \times (q - 1)] = 1.$$

Finally, the integer e is computed from p, q, and d to be the *multiplicative inverse* of d modulo $(p - 1) \times (q - 1)$. That is, e satisfies

$$e \times d \textbf{ mod } (p - 1) \times (q - 1) = 1.$$

We should point out that, although n is publicly known, p and q are not. This condition is allowed because of the well-known fact that it is difficult to factor n. Consequently, the integers d and e cannot be guessed easily.

Let us illustrate this scheme with an example. Let $p = 5$ and $q = 7$. Then, $n = 35$ and $(p - 1) \times (q - 1) = 24$. Since 11 is relatively prime to 24, we can choose $d = 11$; and since $11 \times 11 \textbf{ mod } 24 = 121 \textbf{ mod } 24 = 1$, $e = 11$. Suppose now that $m = 3$. Then,

$$C = m^e \textbf{ mod } n = 3^{11} \textbf{ mod } 35 = 12,$$

and

$$C^d \textbf{ mod } n = 12^{11} \textbf{ mod } 35 = 3 = m.$$

Thus, if we encode m using e, we can decode m using d.

14.7 ■ Summary

Protection is an internal problem. Security must consider both the computer system and the environment (people, buildings, businesses, valuable objects, and threats) within which the system is used.

The data stored in the computer system need to be protected from unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. It is easier to protect against accidental loss of data consistency than to protect against malicious access to the data. Absolute protection of the information stored in a computer system from malicious abuse is not possible, but the cost to the perpetrator can be made sufficiently high to deter most, if not all, attempts to access that information without proper authority.

The various authorization provisions in a computer system may not be sufficient protection for highly sensitive data. In such cases, data may be *encrypted*. It is not possible for encrypted data to be read unless the reader knows how to decipher (*decrypt*) the encrypted data.

■ Exercises

14.1 A password may become known to other users in a variety of ways. Is there a simple method for detecting that such an event has occurred? Explain your answer.

14.2 The list of all passwords is kept within the operating system. Thus, if a user manages to read this list, password protection is no longer provided. Suggest a scheme that will avoid this problem. (Hint: Use different internal and external representations.)

14.3 An experimental addition to UNIX allows a user to connect a *watchdog* program to a file, such that the watchdog is invoked whenever a program requests access to the file. The watchdog then either grants or denies access to the file. Discuss the pros and cons of using watchdogs for security.

14.4 The UNIX program, COPS, scans a given system for possible security holes and alerts the user to possible problems. What are the potential hazards of using such a system for security? How can these problems be limited or eliminated?

14.5 Discuss ways by which managers of systems connected to the Internet could have limited or eliminated the damage done by the worm. What are the drawbacks of making such changes to the way in which the system operates?

14.6 Argue for or against the sentence handed down against Robert Morris, Jr., for his creation and execution of the Internet worm.

14.7 Make a list of security concerns for a computer system for a bank. For each item on your list, state whether this concern relates to physical security, human security, or operating-system security.

14.8 What are the advantages of encrypting data stored in the computer system?

Bibliographic Notes

General discussions concerning security were offered by Hsiao et al. [1979], Landwehr [1981], Denning [1982], Pfleeger [1989], and Russell and Gangemi [1991]. The U.S. federal government is, of course, concerned about security. It publishes the *Orange Book*, which describes a set of security levels, and the features that an operating system must have to qualify for each security rating. Reading it is a good starting point for understanding security concerns. Also of general interest is the the text by Lobel [1986].

Issues concerning the design and verification of secure systems were discussed by Rushby [1981] and Silverman [1983]. A security kernel for a multiprocessor microcomputer was described by Schell [1983]. A distributed secure system was described by Rushby and Randell [1983].

Morris and Thompson [1979] discussed password security. Morshedian [1986] presented methods to fight password pirates. Password authentication with insecure communications was considered by Lamport [1981]. The issue of password cracking was discussed by Seely [1989]. The issue of computer breakins was discussed by Lehmann [1987] and Reid [1987].

Discussions concerning UNIX security were offered by Grampp and Morris [1984], Wood and Kochan [1985], Farrow [1986a, 1986b], Filipski and Hanko [1986], Hecht et al. [1988], Kramer [1988], and Garfinkel and Spafford [1991]. Bershad and Pinkerton [1988] presented the watchdogs extension to BSD UNIX. The COPS security-scanning package for UNIX was written by Dan Farmer at Purdue University. It is available to users on the Internet via the *ftp* program from host ftp.uu.net in directory /pub/security/cops.

Spafford [1989] presented a detailed technical discussion of the Internet worm. The Spafford article appeared with three others in a special section on the Internet worm in *Communications of the ACM*, Volume 32, Number 6, June 1989.

Diffie and Hellman [1976, 1979] were the first researchers to propose the use of the public key-encryption scheme. The algorithm presented in

Section 14.6, which is based on the public key-encryption scheme, was developed by Rivest et al. [1978]. Lempel [1979], Simmons [1979], Gifford [1982], Denning [1982], and Ahituv et al. [1987] concerned the use of cryptography in computer systems. Discussions concerning protection of digital signatures were offered by Akl [1983], Davies [1983], and Denning [1983, 1984].

PART FIVE

DISTRIBUTED SYSTEMS

A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with each other through various communication lines. The processors in a distributed system vary in size and function. They may include small microprocessors, workstations, minicomputers, and large general-purpose computer systems.

A distributed system provides the user with access to the various resources that the system maintains. Access to a shared resource allows computation speedup, and improved data availability and reliability.

A distributed file system is a file-service system whose users, servers, and storage devices are dispersed among the various sites of a distributed system. Accordingly, service activity has to be carried out across the network; instead of a single centralized data repository, there are multiple and independent storage devices.

A distributed system must provide various mechanisms for process synchronization and communication, for dealing with the deadlock problem, and for dealing with a variety of failures that are not encountered in a centralized system.

CHAPTER 15

NETWORK STRUCTURES

A recent trend in computer systems is to distribute computation among several physical processors. There are basically two schemes for building such systems. In a *multiprocessor* (tightly coupled) system, the processors share memory and a clock, and communication usually takes place through the shared memory. In a *distributed* (loosely coupled) system, the processors do not share memory or a clock. Instead, each processor has its own local memory. The processors communicate with one another through various communication networks, such as high-speed buses or telephone lines. In this chapter, we discuss the general structure of distributed systems and the networks that interconnect them. Detailed discussions are given in Chapters 16 to 18.

15.1 ■ Background

A distributed system is a collection of loosely coupled processors interconnected by a communication network. From the point of view of a specific processor in a distributed system, the rest of the processors and their respective resources are *remote*, whereas its own resources are *local*.

The processors in a distributed system may vary in size and function. They may include small microprocessors, workstations, minicomputers, and large general-purpose computer systems. These processors are referred to by a number of different names, such as *sites, nodes, computers, machines, hosts*, and so on, depending on the context in which they are mentioned. We mainly use the term *site*, to indicate a location of machines and *host* to refer to a specific system at a site. Generally, one host at one site, the

server, has a resource that another host at another site, the *client* (or user), would like to use. The purpose of the distributed system is to provide an efficient and convenient environment for this type of sharing of resources. A distributed system is shown in Figure 15.1.

A distributed operating system provides users with access to the various resources that the system provides. By *resources*, we mean both hardware (such as printers and tape drives) and software (such as files and programs). Access to these resources is controlled by the operating system. There are basically two complementary schemes for providing such a service:

- **Network operating systems:** The users are aware of the multiplicity of machines, and need to access these resources by either logging into the appropriate remote machine, or transferring data from the remote machine to their own machines.
- **Distributed operating systems:** The users do not need to be aware of the multiplicity of machines. They access remote resources in the same manner as they do local resources.

Before discussing these two types of operating systems, we shall examine why such systems are useful, and what the structure of the underlying computer network is. In Chapter 16, we present detailed discussions concerning the structure of these two types of operating systems.

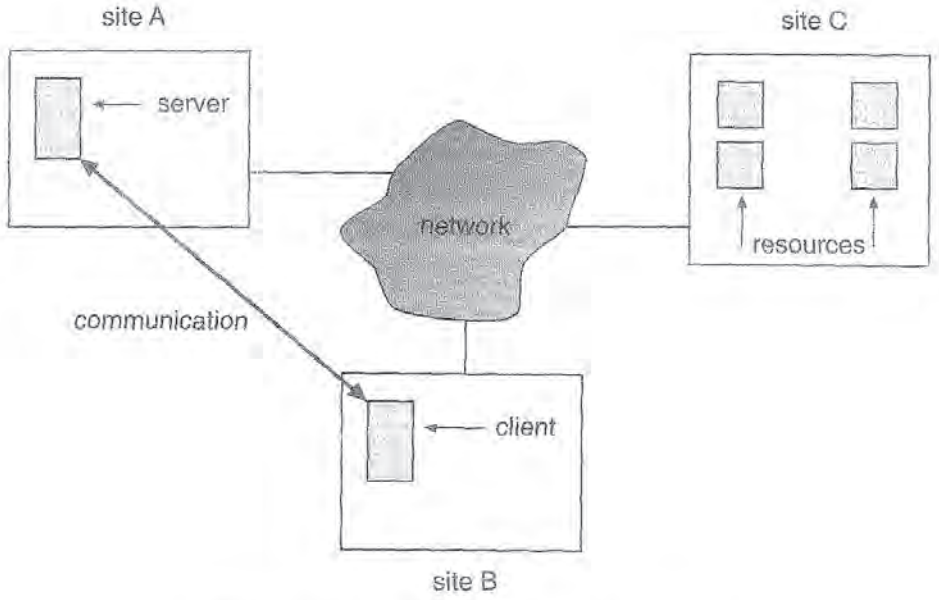


Figure 15.1 A distributed system.

15.2 ■ Motivation

There are four major reasons for building distributed systems: *resource sharing, computation speedup, reliability,* and *communication*. In this section, we briefly elaborate on each of them.

15.2.1 Resource Sharing

If a number of different sites (with different capabilities) are connected to one another, then a user at one site may be able to use the resources available at another. For example, a user at site A may be using a laser printer available at only site B. Meanwhile, a user at B may access a file that resides at A. In general, resource sharing in a distributed system provides mechanisms for sharing files at remote sites, processing information in a distributed database, printing files at remote sites, using remote specialized hardware devices (such as a high-speed array processor), and performing other operations.

15.2.2 Computation Speedup

If a particular computation can be partitioned into a number of subcomputations that can run concurrently, then the availability of a distributed system may allow us to distribute the computation among the various sites, to run the computation concurrently. In addition, if a particular site is currently overloaded with jobs, some of them may be moved to other, lightly loaded, sites. This movement of jobs is called *load sharing*. Automated load sharing, in which the distributed operating system automatically moves jobs, is still uncommon in commercial systems. It remains an active research area, however.

15.2.3 Reliability

If one site fails in a distributed system, the remaining sites can potentially continue operating. If the system is composed of a number of large autonomous installations (that is, general-purpose computers), the failure of one of them should not affect the rest. If, on the other hand, the system is composed of a number of small machines, each of which is responsible for some crucial system function (such as terminal character I/O or the file system), then a single failure may halt the operation of the whole system. In general, if enough redundancy exists in the system (in both hardware and data), the system can continue with its operation, even if some of its sites have failed.

The failure of a site must be detected by the system, and appropriate action may be needed to recover from the failure. The system must no

longer use the services of that site. In addition, if the function of the failed site can be taken over by another site, the system must ensure that the transfer of function occurs correctly. Finally, when the failed site recovers or is repaired, mechanisms must be available to integrate it back into the system smoothly. As we shall see in the following chapters, these actions present difficult issues with many possible solutions.

15.2.4 Communication

When several sites are connected to one another by a communication network, the users at different sites have the opportunity to exchange information. At a low level, *messages* are passed between systems in a manner similar to the single-computer message system discussed in Section 4.6. Given message passing, all the higher-level functionality found in standalone systems can be expanded to encompass the distributed system. Such functions include file transfer, login, mail, and remote procedure calls (RPCs).

The advantage of a distributed system is that these functions can be carried out over great distances. A project can be performed by two people at geographically separate sites. By transferring the files of the project, logging in to each other's remote systems to run programs, and exchanging mail to coordinate the work, the users are able to minimize the limitations inherent in long-distance work. In fact, this book was written in such a manner.

Taken together, the advantages of distributed systems have resulted in an industry-wide trend toward *downsizing*. Many companies are replacing their mainframes with networks of workstations or personal computers. The advantages to the companies include a better "bang for the buck" (better functionality for the cost), flexibility in locating resources and expanding facilities, better user interfaces, and easier maintenance.

Obviously, an operating system that was designed as a collection of processes that communicate through a message system (such as the Accent system) can be extended more easily to a distributed system than can a nonmessage passing system. For instance, MS-DOS is not easily integrated into a network because its kernel is interrupt based and lacks any support for message passing.

15.3 ■ Topology

The sites in the system can be connected physically in a variety of ways. Each configuration has advantages and disadvantages. We describe briefly the most common configurations implemented, and compare them with respect to the following criteria:

- **Basic cost:** How expensive is it to link the various sites in the system?
- **Communication cost:**. How long does it take to send a message from site A to site B?
- **Reliability:** If a link or a site in the system fails, can the remaining sites still communicate with one another?

The various topologies are depicted as graphs whose nodes correspond to sites. An edge from node A to node B corresponds to a direct connection between the two sites.

15.3.1 Fully Connected Networks

In a *fully connected network*, each site is directly linked with all other sites in the system (Figure 15.2). The basic cost of this configuration is high, since a direct communication line must be available between every two sites. The basic cost grows as the square of the number of sites. In this environment, however, messages between the sites can be sent fast; a message needs to use only one link to travel between any two sites. In addition, such systems are reliable, since many links must fail for the system to become partitioned. A system is *partitioned* if it has been split into two (or more) subsystems that lack any connection between them.

15.3.2 Partially Connected Networks

In a *partially connected network*, direct links exist between some, but not all, pairs of sites (Figure 15.3). Hence, the basic cost of this configuration is lower than that of the fully connected network. However, a message from one site to another may have to be sent through several intermediate sites, resulting in slower communication. For example, in the system depicted in

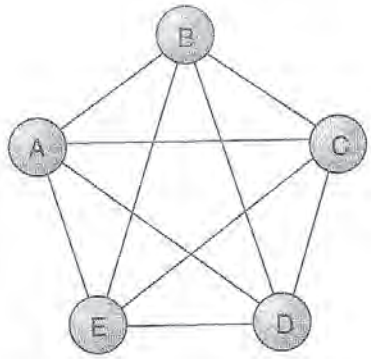


Figure 15.2 Fully connected network.

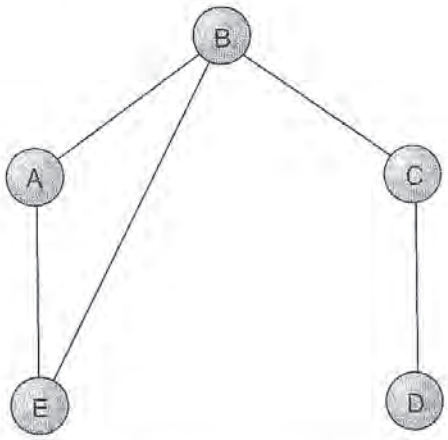


Figure 15.3 Partially connected network.

Figure 15.3, a message from site A to site D must be sent through sites B and C.

In addition, a partially connected system is not as reliable as is a fully connected network. The failure of one link may partition the network. For the example in Figure 15.3, if the link from *B* to C fails, then the network is partitioned into two subsystems. One subsystem includes sites A, B, and E; the second subsystem includes sites C and D. The sites in one partition cannot communicate with the sites in the other. So that this possibility is minimized, each site is usually linked to at least two other sites. For example, if we add a link from A to D, the failure of a single link cannot result in the partition of the network.

15.3.3 Hierarchical Networks

In a *hierarchical network*, the sites are organized as a tree (Figure 15.4). This organization is commonly used for corporate networks. Individual offices are linked to the local main office. Main offices are linked to regional offices; regional offices are linked to corporate headquarters.

Each site (except the root) has a unique parent, and some (possibly zero) number of children. The basic cost of this configuration is generally less than that of the partially connected scheme. In this environment, a parent and child communicate directly. Siblings may communicate with each other only through their common parent. A message from one sibling to another must be sent up to the parent, and then down to the sibling. Similarly, cousins can communicate with each other only through their common grandparent. This configuration matches well with the generalization that systems near each other communicate more than those that are distant. For instance, systems within a building are more likely to transfer data than those at separate installations.

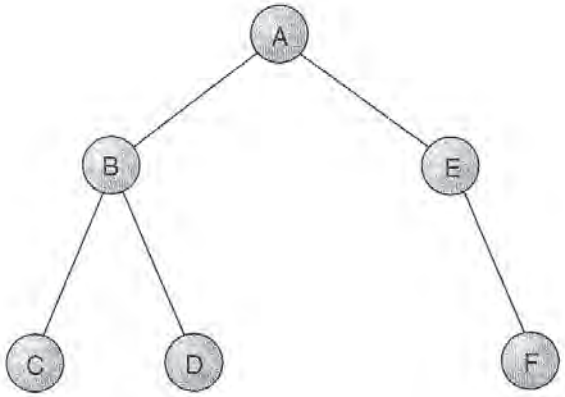


Figure 15.4 Tree-structured network.

If a parent site fails, then its children can no longer communicate with each other or with other processors. In general, the failure of any node (except a leaf) partitions the network into several disjoint subtrees.

15.3.4 Star Networks

In a *star network*, one of the sites in the system is connected to all other sites (Figure 15.5). None of the other sites are connected to any other. The basic cost of this system is linear in the number of sites. The communication cost is also low, because a message from process A to B requires at most two transfers (from A to the central site, and then from the central site B). This simple transfer scheme, however, may not ensure speed, since the central site may become a bottleneck. Consequently, though the number of message transfers needed is low, the time required

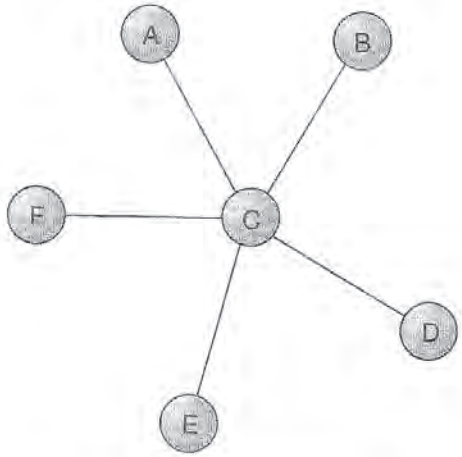


Figure 15.5 Star network.

to send these messages may be high. In many star systems, therefore, the central site is completely dedicated to the message-switching task.

If the central site fails, the network is completely partitioned.

15.3.5 Ring Networks

In a *ring network,* each site is physically connected to exactly two other sites (Figure 15.6a). The ring can be either unidirectional or bidirectional. In a unidirectional architecture, a site can transmit information to only one of its neighbors. All sites must send information in the same direction. In a bidirectional architecture, a site can transmit information to both of its neighbors. The basic cost of a ring is linear in the number of sites. However, the communication cost can be high. A message from one site to another travels around the ring until it reaches its destination. In a unidirectional ring, this process could require $n - 1$ transfers. In a bidirectional ring, at most $n/2$ transfers are needed.

In a bidirectional ring, two links must fail before the network will be partitioned. In a unidirectional ring, a single site failure (or link failure) would partition the network. One remedy is to extend the architecture by providing double links, as depicted in Figure 15.6b. The IBM token ring network is a ring network.

15.3.6 Multiaccess Bus Networks

In a *multiaccess bus network,* there is a single shared link (the bus). All the sites in the system are directly connected to that link, which may be

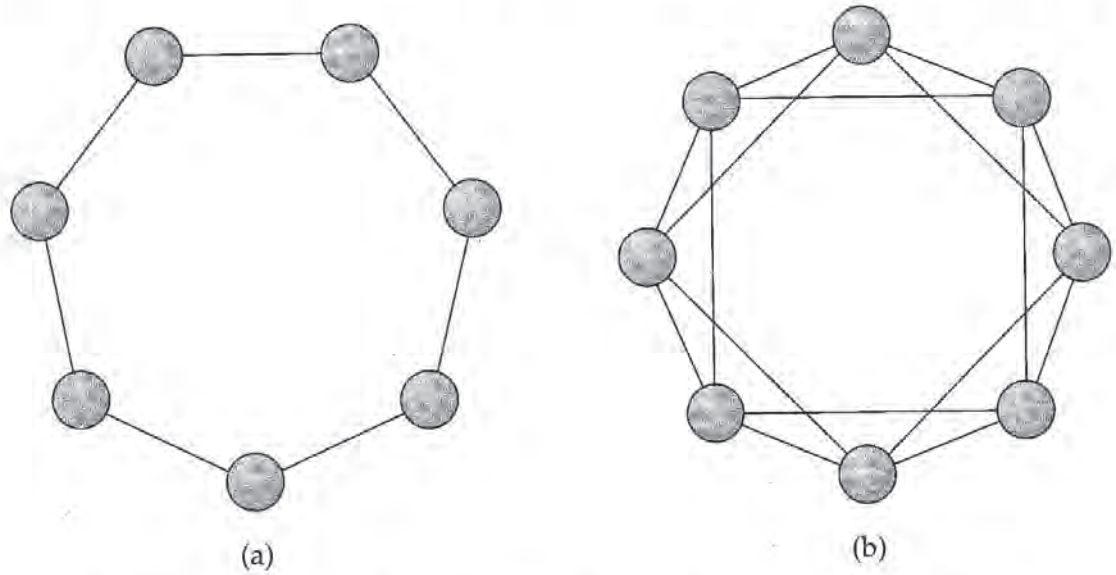


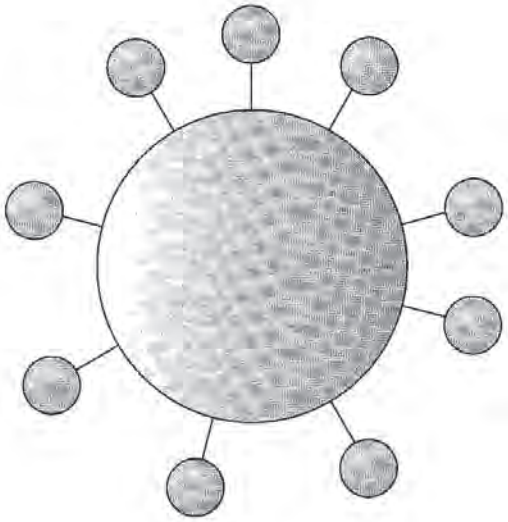
Figure 15.6 Ring networks. (a) Single links. (b) Double links.

organized as a straight line (Figure 15.7a) or as a ring (Figure 15.7b). The sites can communicate with each other directly through this link. The basic cost of the network is linear in the number of sites. The communication cost is quite low, unless the link becomes a bottleneck. Notice that this network topology is similar to that of the star network with a dedicated central site. The failure of one site does not affect communication among the rest of the sites. However, if the link fails, the network is partitioned completely. The ubiquitous Ethernet network, used by many institutions worldwide, is based on the multiaccess bus model.

15.3.7 Hybrid Networks

It is common for networks of differing types to be connected together. For example, within a site, a multiaccess bus such as Ethernet may be used, but between sites, a hierarchy may be used. Communications in such an environment can be tricky because the multiple protocols must be translated to one another and the routing of data is more complicated.

(a)



(b)

Figure 15.7 Bus network. (a) Linear bus. (b) Ring bus.

15.4 ■ Network Types

There are basically two types of networks: *local-area networks* and *wide-area networks*. The main difference between the two is the way in which they are geographically distributed. Local-area networks are composed of processors that are distributed over small geographical areas, such as a single building or a number of adjacent buildings. Wide-area networks, on the other hand, are composed of a number of autonomous processors that are distributed over a large geographical area (such as the United States). These differences imply major variations in the speed and reliability of the communications network, and are reflected in the distributed operating-system design.

15.4.1 Local-Area Networks

Local-area networks (LANs) emerged in the early 1970s, as a substitute for large mainframe computer systems. It had become apparent that, for many enterprises, it is more economical to have a number of small computers, each with its own self-contained applications, rather than a single large system. Because each small computer is likely to need a full complement of peripheral devices (such as disks and printers), and because some form of data sharing is likely to occur in a single enterprise, it was a natural step to connect these small systems into a network.

LANs are usually designed to cover a small geographical area (such as a single building, or a few adjacent buildings) and are generally used in an office environment. All the sites in such systems are close to one another, so the communication links tend to have a higher speed and lower error rate than do their counterparts in wide-area networks. So that this higher speed and reliability can be attained, high-quality (expensive) cables are needed. It is also possible to use the cable exclusively for data network traffic. Over longer distances, the cost of using high-quality cable is enormous, and the exclusive use of the cable tends to be prohibitive.

The most common links in a local-area network are twisted pair, baseband coaxial cable, broadband coaxial cable, and fiber optics. The most common configurations are multiaccess bus, ring, and star networks. Communication speeds range from 1 megabyte per second, for networks such as Appletalk and IBM's slow token ring, to 1 gigabit per second for experimental optical networks. Ten megabits per second is most common, and is the speed of Ethernet. Recently, the optical-fiber–based FDDI network has been increasing its market share. This network is token based and runs at 100 megabits per second.

A typical LAN may consist of a number of different minicomputers or workstations, various shared peripheral devices (such as laser printers or magnetic-tape units), and one or more gateways (specialized processors)

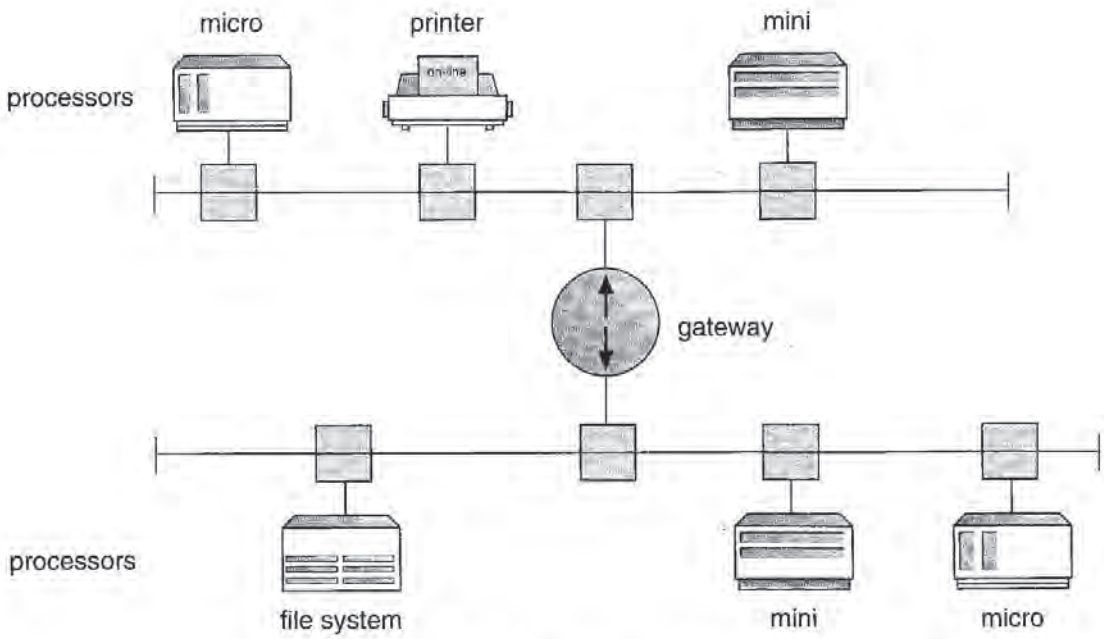


Figure 15.8 Local-area network.

that provide access to other networks (Figure 15.8). An Ethernet scheme is commonly used to construct LANs. There is no central controller in an Ethernet network, because it is a multiaccess bus, so new hosts can be added easily to the network.

15.4.2 Wide-Area Networks

Wide-area networks (WANs) emerged in the late 1960s, mainly as an academic research project to provide efficient communication between sites, allowing hardware and software to be shared conveniently and economically by a wide community of users. The first WAN to be designed and developed was the *Arpanet*. Work on the Arpanet began in 1968. The Arpanet has grown from a four-site experimental network to a worldwide network of networks, the Internet, comprising thousands of computer systems. Recently, several commercial networks have also appeared on the market. The Telenet system is available within the continental United States; the Datapac system is available in Canada. These networks provide their customers with the ability to access a wide range of hardware and software computing resources.

Because the sites in a WAN are physically distributed over a large geographical area, the communication links are by default relatively slow and unreliable. Typical links are telephone lines, microwave links, and satellite channels. These communication links are controlled by special

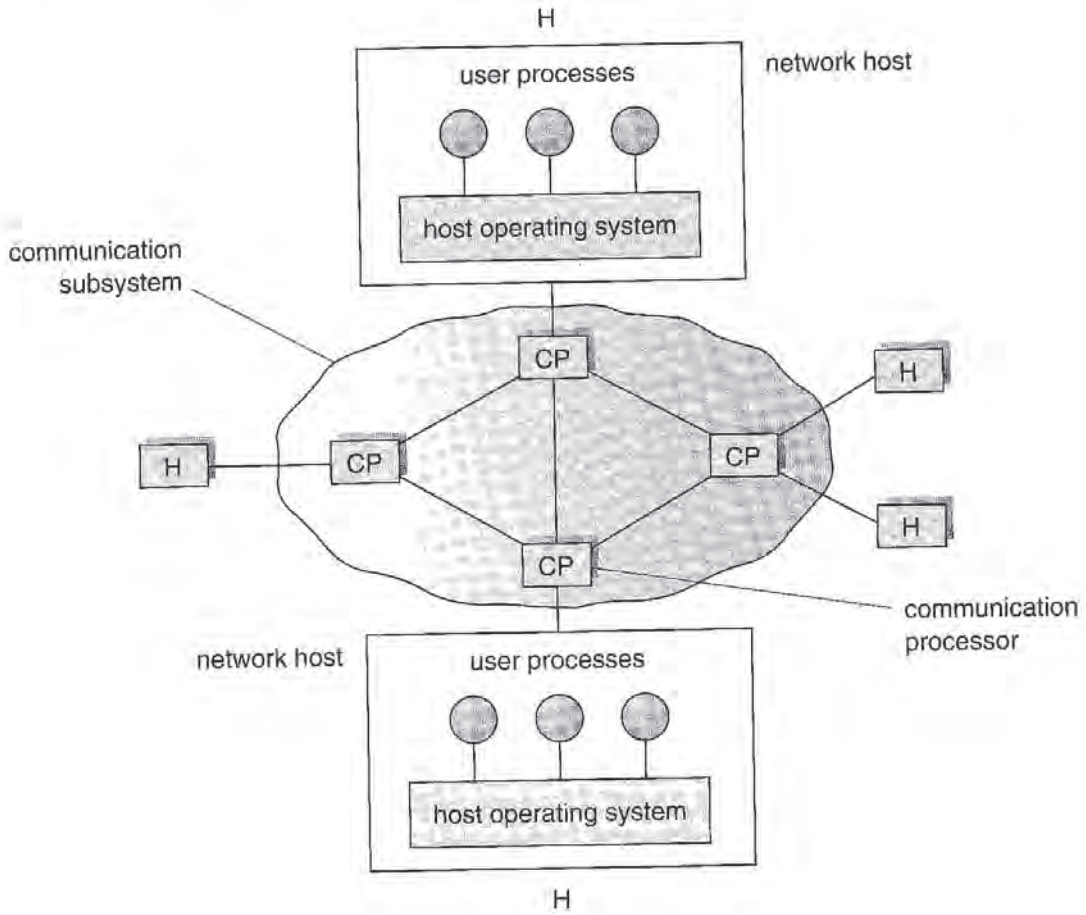


Figure 15.9 Communication processors in a wide-area network.

communication processors (Figure 15.9), which are responsible for defining the interface through which the sites communicate over the network, as well as for transferring information among the various sites.

As an example, let us consider the Internet WAN. The system provides an ability for hosts at geographically separated sites to communicate with one another. The host computers typically differ from one another in type, speed, word length, operating system, and so on. Hosts are generally on LANs, which are in turn connected to the Internet via regional networks. The regional networks, such as NSFnet in the Northeast United States, are interlinked with *routers* (described in Section 15.5.2) to form the worldwide network. Connections between networks frequently use a telephone-system service called T1, which provides a transfer rate of 1.544 megabits per second. The routers control the path each message takes through the net. This routing may be either dynamic, to increase communications efficiency, or static, to reduce security risks or to allow communications charges to be computed.

Other WANs in operation use standard telephone lines as their primary means of communication. *Modems* are the devices that accept digital data from the computer side and convert it to the analog signals that the telephone system uses. A modem at the destination site converts the analog signal back to digital and the destination receives the data. The UNIX news network, UUCP, allows systems to communicate with each other at predetermined times, via modems, to exchange messages. The messages are then routed to other nearby systems and in this way either are propagated to all hosts on the network (public messages) or are transferred to their destination (private messages). WANs are generally slower than LANs; their transmission rates range from 1200 bits per second to over 1 megabit per second.

15.5 ■ Communication

Now that we have discussed the physical aspects of networking, we turn to the internal workings. The designer of a *communication* network must address four basic issues:

- **Naming and name resolution:** How do two processes locate each other to communicate?
- **Routing strategies:** How are messages sent through the network?
- **Connection strategies:** How do two processes send a sequence of messages?
- **Contention:** The network is a shared resource, so how do we resolve conflicting demands for its use?

In Sections 15.5.1 through 15.5.4, we elaborate each of these issues.

15.5.1 Naming and Name Resolution

The first component of network communication is the naming of the systems in the network. For a process at site A to exchange information with a process at site B, they must be able to specify each other. Within a computer system, each process has a process-id, and messages may be addressed with the process-id. Because networked systems share no memory, they initially have no knowledge of the host of their target process, or even if the other process exists.

To solve this problem, processes on remote systems are generally identified by the pair <host name, identifier>, where "host name" is a name unique within the network, and "identifier" may be a process-id or

other unique number within that host. A "host name" is usually an alphanumeric identifier, rather than a number, to make it easier for users to specify. For instance, site A might have hosts named "homer," "marge," "bart," and "lisa." "Bart" is certainly easier to remember than is "12814831100."

Names are convenient for humans to use, but computers prefer numbers for speed and simplicity. For this reason, there must be a mechanism to *resolve* the host name into a host-id which describes the destination system to the networking hardware. This resolve mechanism is similar to the name-to-address binding which occurs during program compilation, linking, loading, and execution (Chapter 8). In the case of host names, there are two possibilities. First, every host may have a data file containing the names and addresses of all the other hosts reachable on the network (similar to binding at compile time). The problem with this model is that adding or removing a host from the network requires updating the data files on all the hosts. The alternative is to distribute the information among systems on the network. The network must then use a protocol to distribute and retrieve the information. This scheme is like execution-time binding. The first method was the original method used on the Internet, but as the Internet grew it became untenable, and the second method, the *domain name service* (DNS) is now in use.

DNS specifies the naming structure of the hosts, as well as name to address resolution. Hosts on the Internet are logically addressed with a multipart host. Names progress from the most specific to the most general part of the address, with periods separating the fields. For instance, "bob.cs.brown.edu" refers to host "bob" in the Department of Computer Science at Brown University. Generally, the system resolves addresses by examining the host name components in reverse order. Each component has a *name server* (simply a process on a system) that accepts a name and returns the address of the name server responsible for that name. As the final step, the name server for the host in question is contacted and a host-id is returned. For our example system, "bob.cs.brown.edu," the following steps would be taken as result of a request made by a process on system A to communicate with "bob.cs.brown.edu":

1. The kernel of system A issues a request to the name server for the "edu" domain, asking for the address of the name server for "brown.edu." The name server for the "edu" domain must be at a known address, so that it can be queried. (Other top-level domains include "com" for commercial sites, "org" for organizations, and one for each country connected to the network (for systems specified by their country rather than organization type)).

2. The "edu" name server returns the address of the host on which the "brown.edu" name server resides.

3. The kernel on system A then queries the name server at this address and asks about "cs.brown.edu."
4. An address again is returned, and a request to that address for "bob.cs.brown.edu" finally returns an *Internet address* host-id for that host (for example, 128.148.31.100).

This protocol may seem inefficient, but local caches are usually kept at each name server to speed the process. For example, the "edu" name server would have "brown.edu" in its cache, and would inform system A that it can resolve two portions of the name, returning a pointer to the "cs.brown.edu" name server. Of course, the contents of these caches must be refreshed over time in case the name server is moved or its address changes. In fact, this service is such an important one that there are many optimizations in the protocol, and many safeguards. Consider what would happen if the primary "edu" name server crashed. It is possible that no "edu" hosts would be able to have their addresses resolved, making them all unreachable! The solution is the use of secondary, backup, name servers to duplicate the contents of the primary servers.

Before the domain name service was introduced, all hosts on the Internet needed copies of a file that contained the names and addresses of each host on the network. All changes to this file had to be registered at one site (host SRI-NIC), and periodically all hosts had to copy the updated file from SRI-NIC to be able to contact new systems or find hosts whose addresses changed. Under the domain name service, each name server site is responsible for updating the host information for that domain. For instance, any host changes at Brown University are the responsibility of the name server for "brown.edu," and do not have to be reported anywhere else. DNS lookups will automatically retrieve the updated information because "brown.edu" is contacted directly. Within domains, there can be autonomous subdomains to distribute further the responsibility for host name and host-id changes.

Generally, it is the responsibility of the operating system to accept from its processes a message destined for <host name, identifier>, and to transfer that message to the appropriate host. The kernel on the destination host is then responsible for transferring the message to the process named by the identifier. This exchange is by no means trivial, and is described later in Section 15.5.3.

15.5.2 Routing Strategies

When a process at site A wants to communicate with a process at site B, how is the message sent? If there is only one physical path from A to B (such as in a star or hierarchical network), the message must be sent through that path. However, if there are multiple physical paths from A

to B, then several routing options exist. Each site has a *routing table*, indicating the alternative paths that can be used to send a message to other sites. The table may include information about the speed and cost of the various communication paths, and it may be updated as necessary, either manually or via programs that exchange routing information. The three most common routing schemes are *fixed routing*, *virtual routing*, and *dynamic routing*:

- **Fixed routing:** A path from A to B is specified in advance and does not change unless a hardware failure disables this path. Usually, the shortest path is chosen, so that communication costs are minimized.
- **Virtual circuit:** A path from A to B is fixed for the duration of one *session*. Different sessions involving messages from A to B may have different paths. A session could be as short as a file transfer or as long as a remote-login period.
- **Dynamic routing:** The path used to send a message from site A to site B is chosen only when a message is sent. Because the decision is made dynamically, separate messages may be assigned different paths. Site A will make a decision to send the message to site C; C, in turn, will decide to send it to site D, and so on. Eventually, a site will deliver the message to B. Usually a site sends a message to another site on that link that is the least used at that particular time.

There are tradeoffs among these three schemes. Fixed routing cannot adapt to link failures or load changes. In other words, if a path has been established between A and B, the messages must be sent along this path, even if the path is down or is used heavily while another possible path is used lightly. We can partially remedy this problem by using virtual circuits, and can avoid it completely by using dynamic routing. Fixed routing and virtual circuits ensure that messages from A to B will be delivered in the order in which they were sent. In dynamic routing, messages may arrive out of order. We can remedy this problem by appending a sequence number to each message.

Note that dynamic routing is the most complicated to set up and run. It is the best way to manage routing in complicated environments, however. UNIX provides both static routing for use on hosts within simple networks, and dynamic routing for complicated network environments. It is also possible to mix the two. Within a site, the hosts may just need to know how to reach the system that connects the local network to other networks (such as companywide networks or the Internet). Such a host is known as a *gateway*. These individual hosts would have a static route to the gateway. The gateway itself would use dynamic routing to be able to reach any host on the rest of the network.

A *router* is the entity within the computer system responsible for routing messages. A router can be a host computer with routing software, or a special-purpose device. Either way, a router must have at least two network connections, or else it would have nowhere to route messages. A router decides whether any given message needs to be passed from the network on which it is received to any other network connected to the router. It makes this determination by examining the destination Internet address of the message. The router examines its tables to determine the location of the destination host, or at least of the network to which to send the message toward the destination host. In the case of static routing, this table is changed only by manual update (a new file is loaded onto the router). With dynamic routing, a *routing protocol* is used between routers to inform them of network changes and to allow them to update their routing tables automatically.

15.5.3 Connection Strategies

Once messages are able to reach their destinations, processes may institute communications "sessions" to exchange information. There are a number of different ways to connect pairs of processes that want to communicate over the network. The three most common schemes are *circuit switching*, *message switching*, and *packet switching*:

- **Circuit switching:** If two processes want to communicate, a permanent physical link is established between them. This link is allocated for the duration of the communication, and no other process can use that link during this period (even if the two processes are not actively communicating for a while). This scheme is similar to that used in the telephone system. Once a communication line has been opened between two parties (that is, party A calls party B), no one else can use this circuit, until the communication is terminated explicitly (for example, when one of the parties hangs up).
- **Message switching:** If two processes want to communicate, a *temporary* link is established for the duration of one message transfer. Physical links are allocated dynamically among correspondents as needed, and are allocated for only short periods. Each message is a block of data, with system information (such as the source, the destination, and error-correction codes) that allows the communication network to deliver the message to the destination correctly. This scheme is similar to the post-office mailing system. Each letter is considered a message that contains both the destination address and source (return) address. Note that many messages (from different users) can be shipped over the same link.

- **Packet switching:** Messages are generally of variable length. To simplify the system design, we commonly implement communication with fixed-length messages called *packets, frames,* or *datagrams.* One logical message may have to be divided into a number of packets. Each packet may be sent to its destination separately, and therefore must include a source and destination address with its data. Each packet may take a different path through the network. The packets must be reassembled into messages as they arrive.

There are obvious tradeoffs among these schemes. Circuit switching requires set-up time, but incurs less overhead for shipping each message, and may waste network bandwidth. Message and packet switching, on the other hand, require less set-up time, but incur more overhead per message. Also, in packet switching, each message must be divided into packets and later reassembled. Packet switching is the most common method used on data networks because it makes the best use of network bandwidth and it is not harmful for data to be broken into packets, possibly routed separately, and reassembled at the destination. Doing the same with an audio signal (say, a telephone communication) could cause great confusion if not done carefully.

15.5.4 Contention

Depending on the network topology, a link may connect more than two sites in the computer network, so it is possible that several sites will want to transmit information over a link simultaneously. This difficulty occurs mainly in a ring or multiaccess bus network. In this case, the transmitted information may become scrambled and must be discarded. The sites must be notified about the problem, so that they can retransmit the information. If no special provisions are made, this situation may be repeated, resulting in degraded performance. Several techniques have been developed to avoid repeated collisions, including *collision detection, token passing,* and *message slots.*

- **CSMA/CD:** Before transmitting a message over a link, a site must listen to determine whether another message is currently being transmitted over that link; this technique is called *carrier sense with multiple access (CSMA).* If the link is free, the site can start transmitting. Otherwise, it must wait (and continue to listen) until the link is free. If two or more sites begin transmitting at exactly the same time (each thinking that no other site is using the link), then they will register a *collision detection (CD)* and will stop transmitting. Each site will try again after some random time interval. Note that, when site A starts transmitting over a link, it must listen continuously to detect collisions with messages from other sites. The main problem with this approach is that, when the

system is very busy, many collisions may occur, and thus performance may be degraded. Nevertheless, CSMA/CD has been used successfully in the Ethernet system, the most common network system. (The Ethernet protocol is defined by the IEEE 802.3 standard.) To limit the number of collisions, the number of hosts per Ethernet network must be limited. Adding more hosts to a congested network could result in poor network throughput. As systems get faster, they are able to send more packets per time segment. As a result, the number of systems per Ethernet segment generally is decreasing, to keep networking performance reasonable.

- **Token passing:** A unique message type, known as a *token*, continuously circulates in the system (usually a ring structure). A site that wants to transmit information must wait until the token arrives. It removes the token from the ring and begins to transmit its messages. When the site completes its round of message passing, it retransmits the token. This action, in turn, allows another site to receive and remove the token, and to start its message transmission. If the token gets lost, then the systems must detect the loss and generate a new token. They usually do that by declaring an *election*, to elect a unique site where a new token will be generated. Later, in Section 18.6, we present one election algorithm. A token-passing scheme has been adopted by the IBM and HP/Apollo systems. The benefit of a token-passing network is that performance is constant. Adding new systems to a network may lengthen the time a system waits for the token, but it will not cause a large performance decrease as may happen on Ethernet. On lightly loaded networks, however, Ethernet is more efficient, because systems may send messages at any time.

- **Message slots:** A number of fixed-length message slots continuously circulate in the system (usually a ring structure). Each slot can hold a fixed-sized message and control information (such as what the source and destination are, and whether the slot is empty or full). A site that is ready to transmit must wait until an empty slot arrives. It then inserts its message into the slot, setting the appropriate control information. The slot with its message then continues in the network. When it arrives at a site, that site inspects the control information to determine whether the slot contains a message for this site. If not, that site recirculates the slot and message. Otherwise, it removes the message, resetting the control information to indicate that the slot is empty. The site can then either use the slot to send its own message or release the slot. Because a slot can contain only fixed-sized messages, a single logical message may have to be broken down into several smaller packets, each of which is sent in a separate slot. This scheme has been adopted in the experimental Cambridge Digital Communication Ring.

15.6 ■ Design Strategies

When designing a communication network, we must deal with the inherent complexity of coordinating asynchronous operations communicating in a potentially slow and error-prone environment. It is also essential that the systems on the network agree on a protocol or a set of protocols for determining host names, locating hosts on the network, establishing connections, and so on. We can simplify the design problem (and related implementation) by partitioning the problem into multiple layers. Each layer on one system communicates with the equivalent layer on other systems. Each layer may have its own protocols, or may be a logical segmentation. The protocols may be implemented in hardware or software. For instance, Figure 15.10 shows the logical communications between two computers, with the three lowest-level layers implemented in hardware. Following the International Standards Organization (ISO), we refer to the layers with the following descriptions:

1. **Physical layer:** The physical layer is responsible for handling both the mechanical and electrical details of the physical transmission of a bit stream. At the physical layer, the communicating systems must agree on the electrical representation of a binary 0 and 1, so that when data are sent as a stream of electrical signals, the receiver is able to interpret the data properly as binary data. This layer is implemented in the hardware of the networking device.

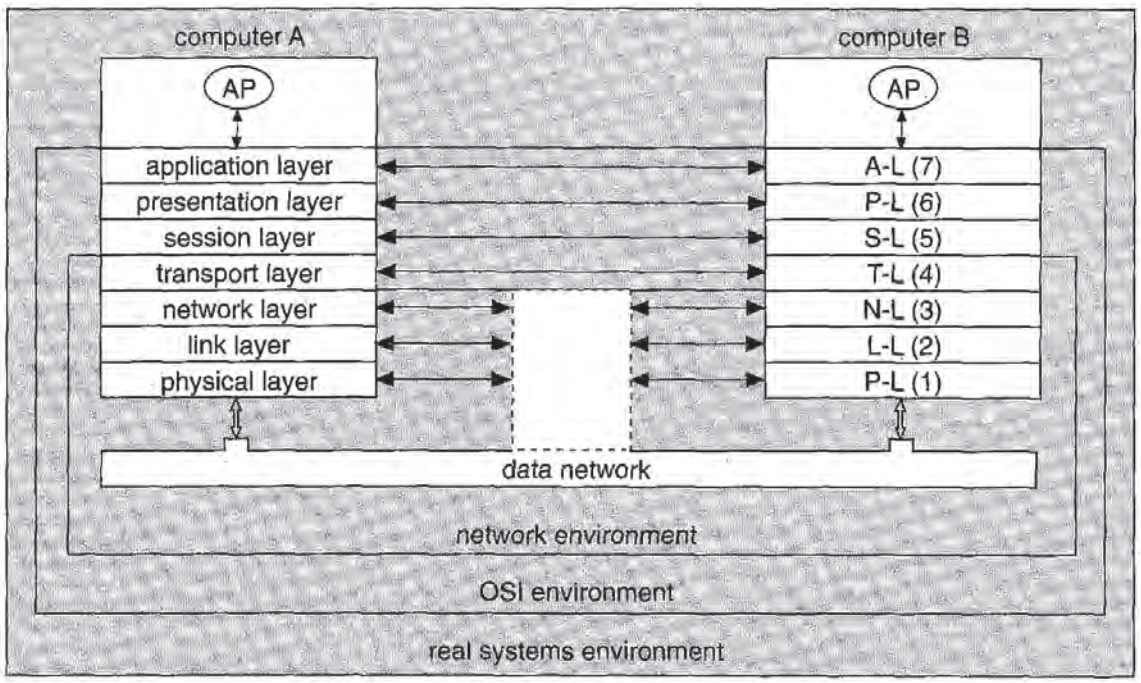


Figure 15.10 Two computers communicating via the ISO network model.

2. **Data-link layer:** The data-link layer is responsible for handling the *frames*, or fixed-length parts of packets, including any error detection and recovery that occurred in the physical layer.

3. **Network layer:** The network layer is responsible for providing connections and for routing packets in the communication network, including handling the address of outgoing packets, decoding the address of incoming packets, and maintaining routing information for proper response to changing load levels. Routers work at this layer.

4. **Transport layer:** The transport layer is responsible for low-level access to the network and for transfer of messages between the clients, including partitioning messages into packets, maintaining packet order, controlling flow, and generating physical addresses.

5. **Session layer:** The session layer is responsible for implementing sessions, or process-to-process communications protocols. Typically, these protocols are the actual communications for remote logins, and for file and mail transfers.

6. **Presentation layer:** The presentation layer is responsible for resolving the differences in formats among the various sites in the network, including character conversions, and half duplex–full duplex modes (character echoing).

7. **Application layer:** The application layer is responsible for interacting directly with the users. This layer deals with file transfer, remote-login protocols, and electronic mail, as well as with schemas for distributed databases.

Figure 15.11 summarizes the ISO *protocol stack* (a set of cooperating protocols), showing the physical flow of data. As mentioned, logically each layer of a protocol stack communicates with the equivalent layer on other systems. But physically, a message starts at or above the application layer, and is passed through each lower level in turn. Each of these layers may modify the message and include message header data for the equivalent layer on the receiving side. Finally, the message makes it to the data-network layer and is transferred as one or more packets (Figure 15.12). These data are received by the data link layer of the target system, and the message is moved up through the protocol stack, being analyzed, modified, and stripped of headers as it progresses. It finally reaches the application layer for use by the receiving process.

The ISO model formalizes some of the earlier work done in network protocols, but was developed in late 1970s and is not yet in widespread use. Part of the basis for ISO is the more timeworn and widely used protocol stack developed under UNIX for use in the Arpanet (which became the Internet.)

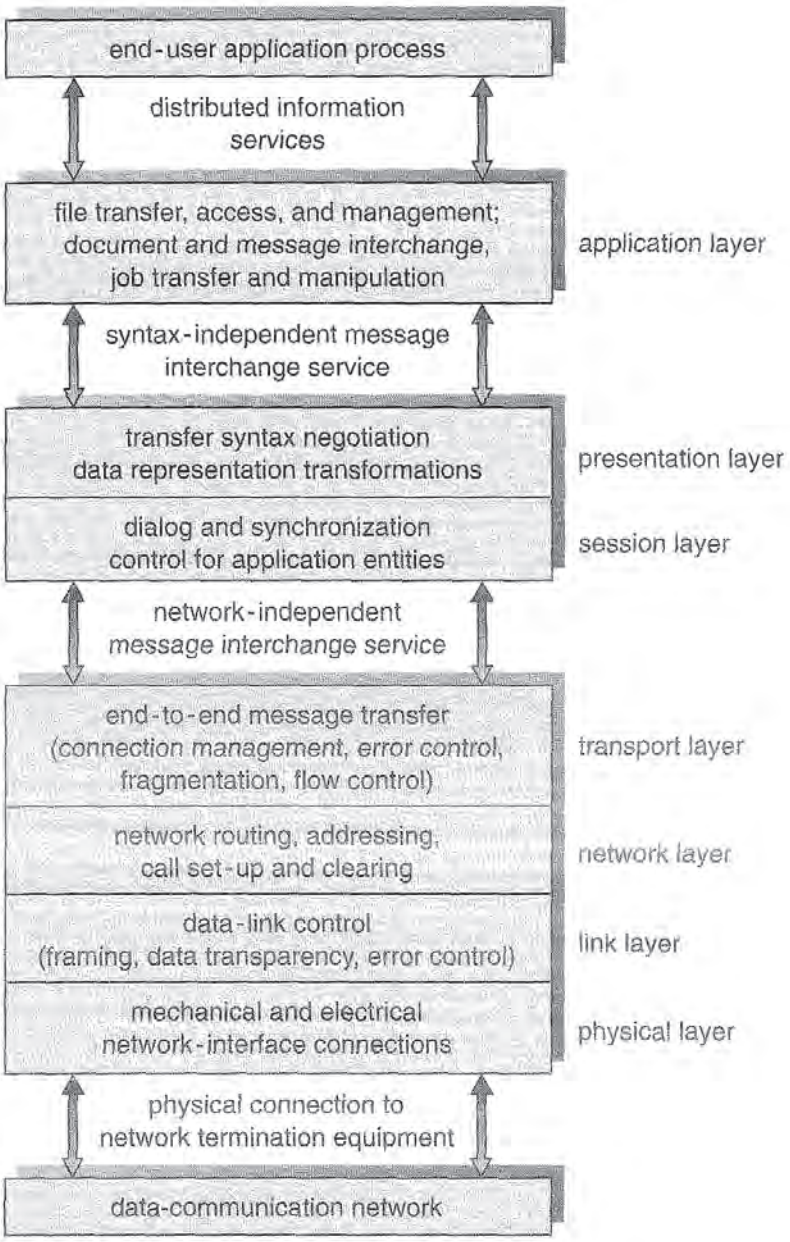


Figure 15.11 The ISO protocol layer summary.

Most Internet sites still communicate via the *Transmission Control Protocol/Internet Protocol*, commonly known as *TCP/IP*. The TCP/IP protocol stack has fewer layers than does the ISO model. Theoretically, because it combines several functions in each layer, it is more difficult to implement but more efficient than ISO networking. The TCP/IP stack (with its ISO correspondence) is shown in Figure 15.13. The IP protocol is responsible for transmitting IP *datagrams*, the basic unit of information, across a TCP/IP internet. TCP uses IP to transport a reliable stream of information between two processes. The other common internet transmission protocol is UDP/IP.

The *User Datagram Protocol (UDP)* is an unreliable, connectionless transport protocol. It uses IP to transfer the packets, but adds error correction and a protocol *port* address to specify the process on the remote system for which the packet is destined.

15.7 ■ Networking Example

We shall now return to the name-resolution issue raised in Section 15.5.1, and shall examine its operation with respect to the TCP/IP protocol stack on the Internet. We consider the processing needed to transfer a packet between hosts on different Ethernet networks.

In a TCP/IP network, every host has a name and an associated 32-bit Internet number (host-id). Both of these must be unique, and, so that the name space can be managed, they are segmented. The name is hierarchical (as explained in Section 15.5.1), describing the host name and then the organizations with which the host is associated. The host-id is split into a network number and a host number. The proportion of the split varies, depending on the size of the network. Once a network number is assigned by the Internet administrators, the site with that number is free to assign host-ids.

The sending system checks its routing tables to locate a router to send the packet on its way. The routers use the network part of the host-id to transfer the packet from its source network to the destination network.

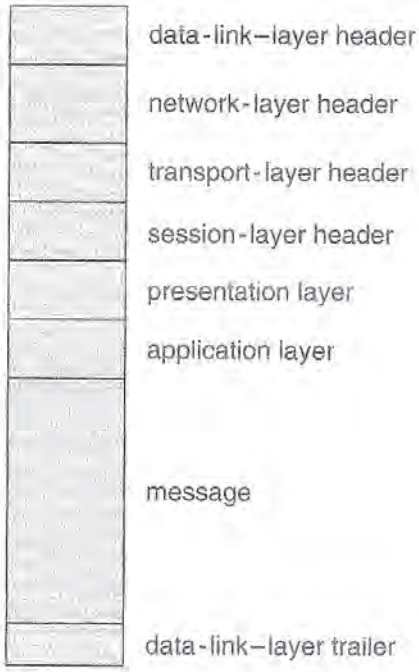


Figure 15.12 An ISO network message.

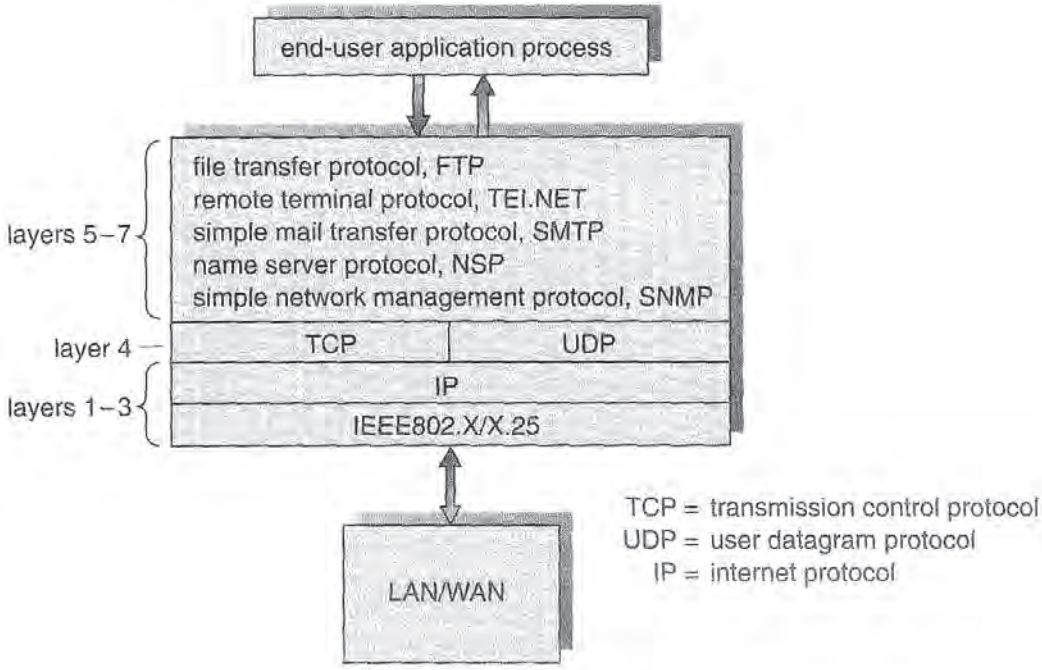


Figure 15.13 The TCP/IP protocol layer summary.

The destination system then receives the packet. The packet may be a complete message, or it may just be a component of a message, with more packets needed before the message is reassembled and passed to the TCP/UDP layer for transmission to the destination process.

Now we know how a packet moves from its source network to its destination. Within a network, how does a packet move from sender (host or router) to receiver? Every Ethernet device has a unique byte number assigned to it for addressing. Two devices communicate with each other only with this number. Periodically, the kernel generates a UDP packet containing the host-id and Ethernet number of the system. This packet is *broadcast* to all other systems on that Ethernet network. A broadcast uses a special network address (usually, the maximum address) to signal that all hosts should receive and process the packet. The broadcast is not resent by gateways, so only systems on the local network receive them. On receipt of this message, every host retrieves this id pair from the UDP packet and caches the pair in an internal table. This sequence is the *Address Resolution Protocol (ARP)*. The cache entries are *aged*, so that they eventually get removed from the cache if a renewing broadcast is not received. In this way hosts that are removed from a network are eventually "forgotten."

Once an Ethernet device has announced its host-id and address, communication can begin. A process may specify the name of a host with which to communicate. The kernel takes that name and determines the

Internet number of the target, using a DNS lookup. The message is passed from the application layer, through the software layers, and to the hardware layer. At the hardware layer, the packet (or packets) has the Ethernet address at its start, and a trailer at the end to indicate the end of the packet and containing a *checksum* for detection of packet damage (Figure 15.14). The packet is placed on the network by the Ethernet device. Note that the data section of the packet may contain some or all of the data of the original message, but may also contain some of the upper-level headers that compose the message. In other words, all parts of the original message must be sent from source to destination, and all headers above the 802.3 layer (data-link layer) are included as data in the Ethernet packets.

If the destination is on the same local network as the source, the system can look in its ARP cache, can find the Ethernet address of the host, and can place the packet on the wire. The destination Ethernet device then sees its address in the packet and reads in the packet, passing it up the protocol stack.

If the destination system is on a network different from that of the source, the source system finds an appropriate router on its network and sends the packet there. Routers then pass the packet along the WAN until it reaches its destination network. The router that connects the destination network checks its ARP cache, finds the Ethernet number of the destination, and sends the packet to that host. Through all of these

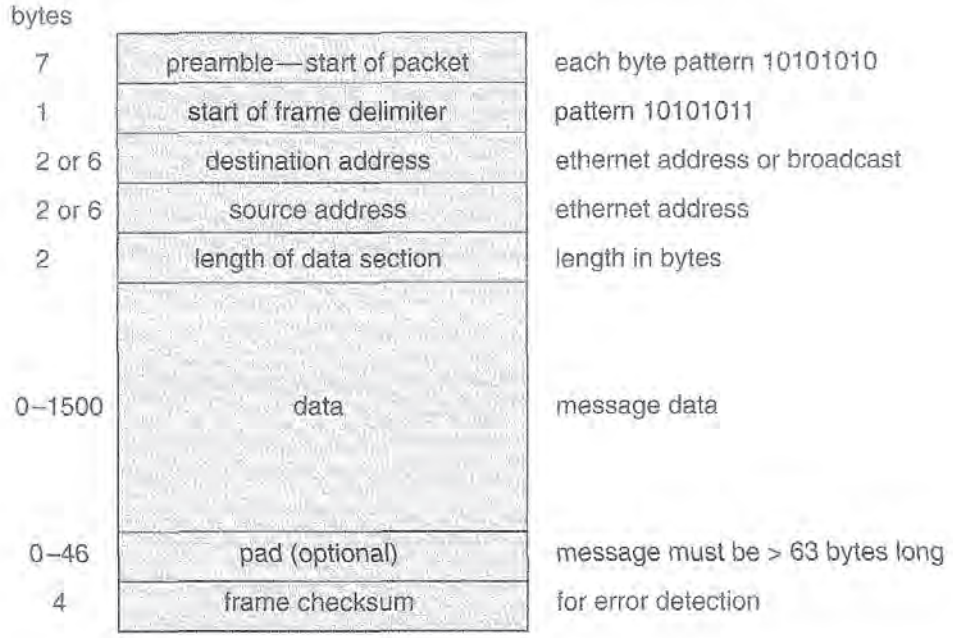


Figure 15.14 An Ethernet packet.

transfers, the data-link–layer header may change as the Ethernet address of the next router in the chain is used, but the other headers of the packet remain the same until the packet is received and they are processed by the protocol stack, finally being passed to the receiving process by the kernel.

15.8 ■ Summary

A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with one another through various communication lines, such as high-speed buses or telephone lines. The processors in a distributed system vary in size and function. They may include small microprocessors, workstations, minicomputers, and large general-purpose computer systems.

The processors in the system are connected through a communication network, which can be configured in a number of different ways. The network may be fully or partially connected. It may be a tree, a star, a ring, or a multiaccess bus. The communication-network design must include routing and connection strategies, and must solve the problems of contention and security.

Principally, there are two types of distributed systems: LANs and WANs. The main difference between the two is in the way they are distributed geographically. LANs are composed of processors that are distributed over small geographical areas, such as a single building or a few adjacent buildings. WANs are composed of autonomous processors that are distributed over a large geographical area (such as the United States).

Protocol stacks, as specified by network layering models, massage the message, adding information to it to ensure that it reaches its destination. A naming system such as DNS must be used to translate from a host name to network address, and another protocol (such as ARP) may be needed to translate the network number to a network device address (an Ethernet address, for instance). If systems are on separate networks, routers are needed to pass packets from source network to destination network.

■ Exercises

15.1 Contrast the various network topologies in terms of reliability.

15.2 Why do most WANs employ only a partially connected topology?

15.3 What are the main differences between a WAN and a LAN?

15.4 What network configuration would best suit the following environments?

a. A dormitory floor

b. A university campus

c. A state

d. A nation

15.5 Even though the ISO model of networking specifies seven layers of functionality, most computer systems use fewer layers to implement a network. Why do they use fewer layers? What problems could the use of fewer layers cause?

15.6 Explain why a doubling of the speed of the systems on an Ethernet segment may result in decreased network performance. What changes could be made to ameliorate the problem?

15.7 Under what circumstances is a token-ring network more effective than an Ethernet network?

15.8 Why would it be a bad idea for gateways to pass broadcast packets between networks? What would be the advantages of doing so?

15.9 In what ways is using a name server better than using static host tables? What are the problems and complications associated with name servers? What methods could be used to decrease the amount of traffic name servers generate to satisfy translation requests?

15.10 Of what use is an address resolution protocol? Why is the use of such a protocol better than making each host read each packet to determine to whom it is destined? Does a token-ring network need such a protocol?

Bibliographic Notes

Tanenbaum [1988] and Halsall [1992] provided general overviews of computer networks. Fortier [1989] presented a detailed discussion of networking hardware and software.

The Internet and several other networks were discussed in Quarterman and Hoskins [1986]. The Internet and its protocols was described in Comer [1991] and Comer and Stevens [1991, 1993]. UNIX network programming was described thoroughly in Stevens [1990]. The general state of networks has been given in Quarterman [1990].

Feng [1981] surveyed the various network topologies. Boorstyn and Frank [1977] and Gerla and Kleinrock [1977] discussed topology design problems. Day and Zimmerman [1983] discussed the OSI model.

A special issue of *Computer Networks* [December 1979] included nine papers on LANs covering such subjects as hardware, software, simulation, and examples. A taxonomy and extensive list of LANs were presented by Thurber and Freeman [1980]. Stallings [1984] discussed various types of ring-structured LANs.

Reliable communication in the presence of failures was discussed by Birman and Joseph [1987]. Integrating security in a large distributed system was discussed by Satyanarayanan [1989].

CHAPTER 16

DISTRIBUTED-SYSTEM STRUCTURES

In this chapter, we discuss the general structure of distributed systems. We contrast the main differences in operating-system design between these types of systems and the centralized systems with which we were concerned previously.

16.1 ■ Network-Operating Systems

A *network operating system* provides an environment where users, who are aware of the multiplicity of machines, can access remote resources by either logging into the appropriate remote machine, or transferring data from the remote machine to their own machines.

16.1.1 Remote Login

An important function of a network operating system is to allow users to log in remotely on another computer. The Internet provides the *telnet* facility for this purpose. To illustrate this facility, let us suppose that a user at Brown University wishes to compute on "cs.utexas.edu," a computer that is located at the University of Texas. To do so, the user must have a valid account on that machine. To log in remotely, the user issues the command

telnet cs.utexas.edu

This command results in a connection being formed between the local machine at Brown University and the "cs.utexas.edu" computer. After this

connection has been established, the networking software creates a transparent, bidirectional link such that all characters entered by the user are sent to a process on "cs.utexas.edu," and all the output from that process is sent back to the user. The process on the remote machine asks the user for a login name and a password. Once the correct information has been received, the process acts as a proxy for the user, who can compute on the remote machine just as any local user can.

16.1.2 Remote File Transfer

Another major function of a network operating system is to provide a mechanism for file transfer from one machine to another. In such an environment, each computer maintains its own local file system. If a user at one site (say, "cs.brown.edu") wants to access a file located on another computer (say, "cs.utexas.edu"), then the file must be copied explicitly from the computer at Texas to the computer at Brown.

The Internet provides a mechanism for such a transfer with the File Transfer Protocol (FTP) program. Suppose that a user on cs.brown.edu wants to copy a file *paper.tex* that resides on cs.utexas.edu into a local file *my-paper.tex*. The user must first invoke the FTP program, by executing

ftp cs.utexas.edu

The program then asks the user for the login name and a password. Once the correct information has been received, the user must connect to the subdirectory where the file *paper.tex* resides, and then copy the file by executing

get *paper.tex my-paper.tex*

In this scheme, the file location is not transparent to the user; users must know exactly where each file is. Moreover, there is no real file sharing, because a user can only copy a file from one site to another. Thus, several copies of the same file may exist, resulting in a waste of space. In addition, if these copies are modified, the various copies will be inconsistent.

Notice that, in our example, the user at Brown university must have login permission on "cs.utexas.edu." FTP also provides a way to allow a user who does not have an account on the Texas computer to copy files remotely. This remote copying is accomplished through the "anonymous ftp" method, which works as follows. The file to be copied (that is, *paper.tex*) must be placed in a special subdirectory (say *ftp*) with the protection set to allow the public to read the file. A user who wishes to copy the file uses the **ftp** command as before. When the user is asked for the login name, the user supplies the name "anonymous," and an arbitrary password.

Once anonymous login is accomplished, care must be taken by the system to ensure that this partially authorized user does not access inappropriate files. Generally, the user is allowed to access only those files that are in the directory tree of user "anonymous." Any files placed here are accessible to any anonymous users, subject to the usual file-protection scheme used on that machine. Anonymous users, however, cannot access files outside of this directory tree.

The FTP mechanism is implemented in a manner similar to telnet. There is a daemon on the remote site that watches for connection requests to the system's FTP port. Login authentication is accomplished, and the user is allowed to execute commands remotely. Unlike the telnet daemon, which executes any command for the user, the FTP daemon responds only to a predefined set of file-related commands. These include:

- **get:** Transfer a file from the remote machine to the local machine
- **put:** Transfer from the local machine to the remote machine
- **ls** or **dir:** List files in the current directory on the remote machine
- **cd:** Change the current directory on the remote machine

There are also various commands to change transfer modes (for binary or ASCII files) and to determine connection status.

An important point about telnet and FTP is that they require the user to change paradigms. FTP requires the user to know a command set entirely different from the normal operating-system commands. Telnet requires a smaller shift, in which the user must know appropriate commands on the remote system. For instance, a user on a UNIX machine who telnets to a VMS machine must switch to VMS commands for the duration of the telnet session. Facilities are more convenient for users if they do not require the use of a different set of commands. Distributed operating systems are designed to ameliorate this problem.

16.2 ■ Distributed-Operating Systems

In a distributed operating system, the users access remote resources in the same manner as they do local resources. Data and process migration from one site to another is under the control of the distributed operating system.

16.2.1 Data Migration

Suppose that a user on site A wants to access data (such as a file) that reside at site B. There are two basic methods for the system to transfer the data. One approach is to transfer the entire file to site A. From that point

on, all access to the file is local. When the user no longer needs access to the file, a copy of the file (if it has been modified) is sent back to site B. Even if only a modest change has been made to a large file, all the data must be transferred. This mechanism can be thought of as an automated FTP system. This approach was used in the Andrew file system, as will be discussed in Chapter 17, but it was found to be too inefficient.

The other approach is to transfer to site A only those portions of the file that are actually *necessary* for the immediate task. If another portion is required later, another transfer will take place. When the user no longer wants to access the file, any part of it that has been modified must be sent back to site B. (Note the similarity to demand paging.) The Sun Microsystems' Network File System (NFS) protocol uses this method (see Chapter 17), as do newer versions of Andrew.

Clearly, if only a small part of a large file is being accessed, the latter approach is preferable. If significant portions of the file are being accessed, it is more efficient to copy the entire file.

Note that it is not sufficient merely to transfer data from one site to another. The system must also perform various data translations if the two sites involved are not directly compatible (for instance, if they use different character-code representations or represent integers with a different number or order of bits).

16.2.2 Computation Migration

In some circumstances, it may be more efficient to transfer the computation, rather than the data, across the system. For example, consider a job that needs to access various large files that reside at different sites, to obtain a summary of those files. It would be more efficient to access the files at the sites where they reside and then to return the desired results to the site that initiated the computation. Generally, if the time to transfer the data is longer than the time to execute the remote command, the remote command should be used.

Such a computation can be carried out in a number of different ways. Suppose that process P wants to access a file at site A. Access to the file is carried out at site A, and could be initiated by a *remote procedure call*, or RPC. An RPC uses a datagram protocol (UDP on the internet) to execute a routine on a remote system (Section 16.3.1). Process P invokes a predefined procedure at site A. The procedure executes appropriately, and then returns the results to P.

Alternatively, process P can send a *message* to site A. The operating system at site A would then create a new process Q whose function is to carry out the designated task. When process Q completes its execution, it sends the needed result back to P via the message system. Note that, in this scheme, process P may execute concurrently with process Q and in fact may have several processes running concurrently on several sites.

Both methods could be used to access several files residing at various sites. One remote procedure call might result in the invocation of another remote procedure call, or even in the transfer of messages to another site. Similarly, process Q could, during the course of its execution, send a message to another site, which in turn would create another process. This process might either send a message back to Q or repeat the cycle.

16.2.3 Process Migration

A logical extension to computation migration is process migration. When a process is submitted for execution, it is not always executed at the site in which it is initiated. It may be advantageous to execute the entire process, or parts of it, at different sites. This scheme may be used for several reasons:

- **Load balancing:** The processes (or subprocesses) may be distributed across the network to even the workload.
- **Computation speedup:** If a single process can be divided into a number of subprocesses that may run concurrently on different sites, then the total process turnaround time can be reduced.
- **Hardware preference:** The process may have characteristics that make it more suitable for execution on some specialized processor (such as matrix inversion on an array processor, rather than on a microprocessor).
- **Software preference:** The process may require software that is available at only a particular site, and either the software cannot be moved, or it is less expensive to move the process.
- **Data access:** Just as in computation migration, if the data being used in the computation are numerous, it may be more efficient to have a process run remotely, rather than to transfer all the data locally.

There are basically two complementary techniques that can be used to move processes in a computer network. In the first, the system can attempt to hide the fact that the process has migrated from the client. This scheme has the advantage that the user does not need to code his program explicitly to accomplish the migration. This method is usually employed for achieving load balancing and computation speedup among homogeneous systems, as they do not need user input to help them execute programs remotely.

The other approach is to allow (or require) the user to specify explicitly how the process should migrate. This method is usually employed in a situation when the process must be moved to satisfy a hardware or software preference.

16.3 ■ Remote Services

Consider a user who needs access to data located at some other site. For example, a user may wish to find out the total number of lines, words and characters in a file located at another site A. The request for doing so is handled by a remote server at site A, who accesses the file, computes the desired result, and eventually transfers the actual data back to the user.

One way to achieve this transfer is through the *remote-service* method. Requests for accesses are delivered to the server. The server machine performs the accesses, and their results are forwarded back to the user. There is a direct correspondence between accesses and traffic to and from the server. Access requests are translated to messages for the servers, and server replies are packed as messages sent back to the users. Every access is handled by the server and results in network traffic. For example, a read results in a request message being sent to the server, and a reply to the user with the requested data.

16.3.1 Remote Procedure Calls

One of the most common forms of remote service is the *remote procedure call (RPC)* paradigm, which we discussed briefly in Section 4.6.3. The RPC was designed as a way to abstract the procedure-call mechanism for use between systems with network connections. It is similar in many respects to the IPC mechanism described in Section 4.6, and is usually built on top of such a system. Because we are dealing with an environment where the processes are executing on separate systems, we must use a *message-based* communication scheme to provide remote service. In contrast to the IPC facility, the messages exchanged for RPC communication are well structured, and are thus no longer just packets of data. They are addressed to an RPC daemon listening to a *port* on the remote system, and contain an identifier of the function to execute and the parameters to pass to that function. The function is then executed as requested, and any output is sent back to the requester in a separate message.

A *port* is simply a number included at the start of a message packet. Whereas a system normally has one network address, it can have many ports within that address to differentiate the many network services it supports. If a remote process needs a service, it addresses its messages to the proper port. For instance, if a system wished to allow other systems to be able to list the current users on it, it would have a daemon supporting such an RPC attached to a port, say port 3027. Any remote system could obtain the needed information (that is, the list of current users) by sending an RPC message to port 3027 on the server; the data would be received in a reply message.

The RPC mechanism is common on networked systems, so there are several issues that we should discuss in regard to its operation. One

important issue is that of the semantics of a call. Whereas local procedure calls fail only under extreme circumstances, RPCs can fail, or be duplicated and executed more than once, due to common network errors. Because we are dealing with message transfer over unreliable communication links, it is much easier for an operating system to ensure that a message was acted on at most once, than it is to ensure that the message was acted on exactly once. Because local procedure calls have the latter meaning, most systems attempt to duplicate that functionality. They do so by attaching to each message a timestamp. The server must keep a history of all the timestamps of messages it has already processed, or a history large enough to ensure that repeated messages are detected. Incoming messages that have a timestamp already in the history are ignored. Generation of these timestamps is discussed in Section 18.1.

Another important issue concerns the communication between server and client. With standard procedure calls, some form of binding takes place during link, load, or execution time (see Chapter 8), such that a procedure call's name is replaced by the memory address of the procedure call. The RPC scheme requires a similar binding of the client and the server port, but how does a client know the port numbers on the server? Neither system has full information about the other because they do not share memory. Two approaches are common. First, the binding information may be predecided, in the form of fixed port addresses. At compile time, an RPC call has a fixed port number associated with it. Once a program is compiled, the server cannot change the port number of the requested service. Second, binding can be done dynamically by a rendezvous mechanism. Typically, an operating system provides a rendezvous (also called a matchmaker) daemon on a fixed RPC port. A client then sends a message, containing the name of the RPC, to the rendezvous daemon requesting the port address of the RPC it needs to execute. The port number is returned, and the RPC calls may be sent to that port until the process terminates (or the server crashes). This method requires the extra overhead of the initial request, but is more flexible than the first approach. A sample interaction is shown in Figure 16.1.

The remote procedure calls scheme is useful in implementing a distributed file system (see Chapter 17). Such a system can be implemented as a set of RPC daemons and clients. The messages are addressed to the DFS port on a server on which a file operation is to take place. The message contains the disk operation to be performed. Disk operations might be **read, write, rename, delete,** or **status,** corresponding to the usual file-related system calls. The return message contains any data resulting from that call, which is executed by the DFS daemon on behalf of the client. For instance, a message might contain a request to transfer a whole file to a client, or be limited to simple block requests. In the latter case, several such requests might be needed if a whole file is to be transferred.

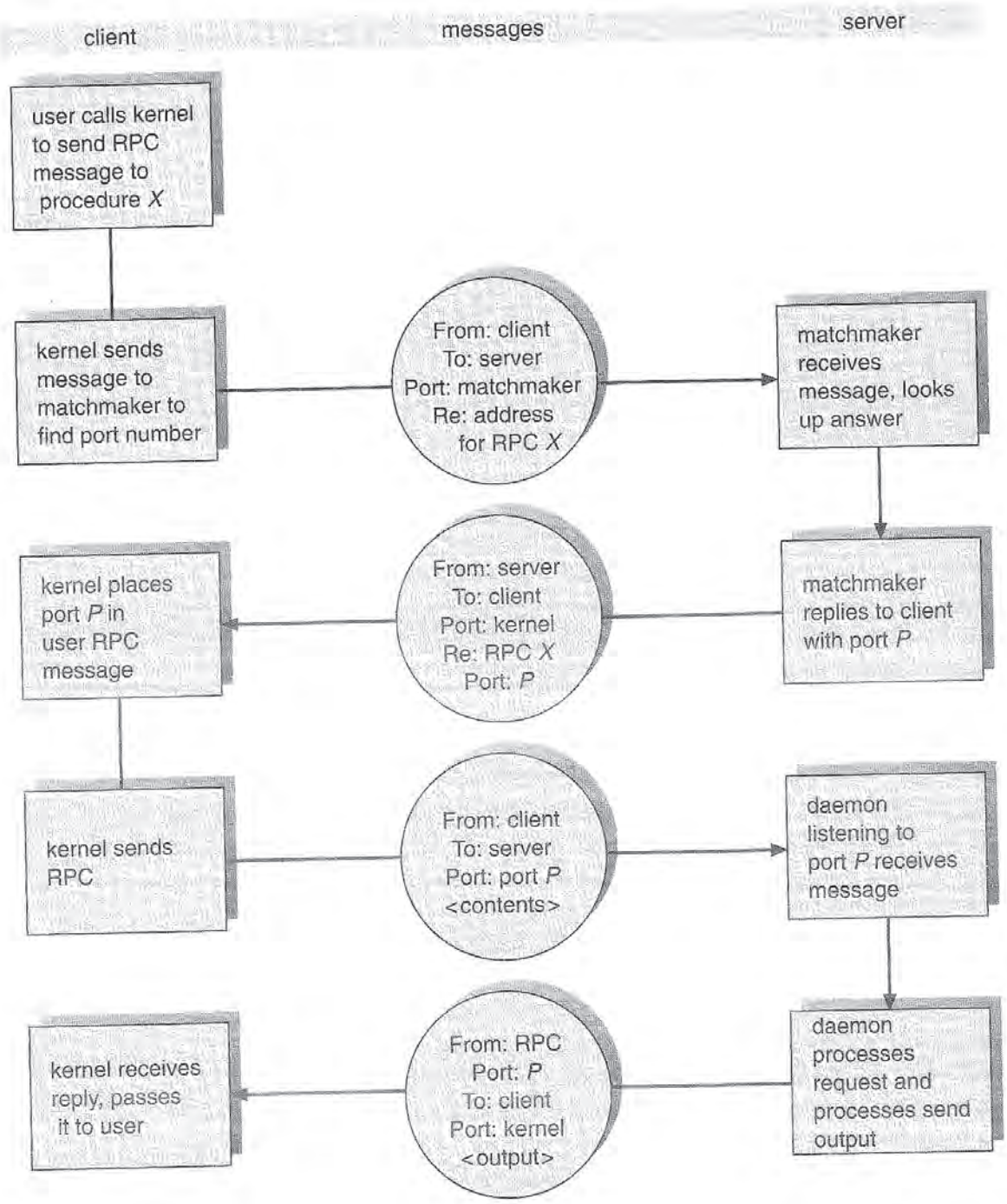


Figure 16.1 Execution of a remote procedure call (RPC).

16.3.2 Threads

Distributed systems often utilize both threads and RPCs. Threads may be used to send and receive messages while other operations within the task continue asynchronously. For instance, a daemon might have many threads waiting to receive request messages, with a single message being

picked up by a thread and processed concurrently with other threads doing the same. Researchers are studying how the use of threads can make RPCs more lightweight. One improvement over conventional RPC takes advantage of the fact that a server thread that blocks waiting for a new request has little significant context information. In a scheme sometimes referred to as *implicit receive*, a thread that has completed a job disappears. The kernel then simply creates a new thread to handle incoming requests. It also writes the message onto the server's address space and sets up the stack so that the new thread can access the message. A thread created on an "as needed" basis to respond to a new RPC can be called a *pop-up thread*. This technique improves performance because it is cheaper to start a new thread than to restore an existing one. Because no threads block waiting for new work, no context has to be saved, or restored. Additional time is saved because incoming RPCs do not have to be copied to a buffer within a server thread.

It has been shown that a significant number of RPCs in distributed systems are to processes on the same machine as the caller. RPC can be made more lightweight if highly efficient communication is enabled via shared memory between threads in different processes that are running on the same machine. On starting up, a server thread, *ST*, exports its interface to the kernel. Defined in the interface are those procedures that are callable, their parameters, and other related features. A client thread, *CT*, that imports this interface will receive a unique identifier that it will use later to make the call. To make a call to *ST*, *CT* pushes the arguments onto an *argument stack*, which is a data structure shared by both *ST* and *CT*. As part of the call, *ST* also puts the unique identifier in a register. Once the kernel observes this placement, it knows that the call is not remote, but rather is local. The kernel then puts the client in the server's address space and initiates *CT*'s execution of *ST*'s procedure. Because the arguments are already in place and do not have to be copied, local RPCs handled in this manner show an improved performance time over conventional RPCs.

Let us illustrate the use of threads in a distributed environment by examining a specific threads package available through the Open Software Foundation's Distributed Computing Environment (DCE). DCE is important because almost all major UNIX vendors have agreed to add it to their UNIX implementations as a way of standardizing the network functionality and protocols and allow better interoperability between systems. It will also be used by Microsoft's Windows/NT. The DCE package provides, for user convenience, a large number of calls. In this chapter, we consider the most significant; we group them into the following five categories:

1. Thread-management calls: **create, exit, join, detach**
2. Synchronization calls: **mutex_init, mutex_destroy, mutex_lock, mutex_trylock, mutex_unlock**

3. Condition-variable calls: **cond_init, cond_destroy, cond_wait, cond_signal, cond_broadcast**
4. Scheduling calls: **setscheduler, getscheduler, setprio, getprio**
5. Kill-thread calls: **cancel, setcancel**

We now briefly discuss each of these.

The four thread-management calls create new threads and allow them to exit on completing a request. The call **join,** which is similar to the UNIX system call **wait,** permits a parent to wait for its child. If a parent thread need not wait for the child, it may decline to do so with **detach.**

Access to shared data can be synchronized through the use of *mutex,* which is a variant of a binary semaphore (see Chapter 6). Mutexes in DCE can be created (initiated) or destroyed dynamically. A mutex is either in a locked or unlocked state. Three operations are defined on mutexes, **mutex_lock, mutex_trylock,** and **mutex_unlock.** A lock attempt succeeds on only an unlocked mutex; once locked, the mutex is confined to a single atomic action. Conversely, a mutex is unlocked by **mutex_unlock;** if several threads are waiting on the mutex, only a single thread is released. The **mutex_trylock** call attempts to lock a mutex. If that mutex is already locked, then this call returns a status code indicating that it did not lock, and the thread issuing the call is not blocked. If that mutex is unlocked, then the call returns a success status code.

DCE also provides the condition variable feature found in most thread packages. This feature functions in much the same way as do the condition variables in monitors (Chapter 6). A condition variable works in conjunction with a mutex but has a significantly different use. To illustrate, let us suppose that a thread A has locked a mutex m1 to gain entry to a critical region. Once there, the thread discovers that it cannot continue executing until another thread B has executed and has modified data in that critical region. Thread A would normally lock a second mutex m2 for that purpose. This second locking, however, would result in a deadlock, because thread B cannot enter the critical region to change the data for which A is waiting.

Rather than locking mutex m2, thread A could lock a condition variable. This condition-variable locking would automatically unlock mutex m1 if the setting of the lock by thread A results in a wait. When thread B changes the desired data, it issues a wakeup call that activates the waiting thread A.

Like mutexes, condition variables can also be created and deleted dynamically. The capacity to wait on a condition variable is provided by **cond_wait.** DCE provides two types of wakeup operations: **cond_signal** resumes the execution of only a single waiting thread, whereas **cond_broadcast** resumes the execution of all threads that are waiting on the condition variable.

Another category of calls we consider comprises scheduling calls. The set of scheduling calls in the DCE package allow the scheduling algorithms and priorities to be set and read. A number of preemptive and nonpreemptive scheduling algorithms are available, including round robin and FIFO.

The threads package provides calls for killing threads. The system call **cancel** enacts an attempt to kill another thread. That thread can use the **setcancel** call either to allow or to disallow the kill attempt.

16.4 ■ Robustness

A distributed system may suffer from various types of hardware failure. The failure of a link, the failure of a site, and the loss of a message are the most common failures. To ensure that the system is robust, we must *detect* any of these failures, *reconfigure* the system so that computation may continue, and *recover* when a site or a link is repaired.

16.4.1 Failure Detection

In an environment with no shared memory, it is generally not possible to differentiate among link failure, site failure, and message loss. We can usually detect that one of these failures has occurred, but we may not be able to identify what kind of failure it is. Once a failure has been detected, appropriate action must be taken, depending on the particular application.

To detect link and site failure, we use a *handshaking* procedure. Suppose that sites A and B have a direct physical link between them. At fixed intervals, both sites send each other an *I-am-up* message. If site A does not receive this message within a predetermined time period, it can assume that site B has failed, that the link between A and B has failed, or that the message from B has been lost. At this point, site A has two choices. It can wait for another time period to receive an *I-am-up* message from B, or it can send an *Are-you-up?* message to B.

If site A does not receive an *I-am-up* message or a reply to its inquiry, the procedure can be repeated. The only conclusion that site A can draw safely is that some type of failure has occurred.

Site A can try to differentiate between link failure and site failure by sending an *Are-you-up?* message to B by another route (if one exists). If and when B receives this message, it immediately replies positively. This positive reply tells A that B is up, and that the failure is in the direct link between them. Since it is not known in advance how long it will take the message to travel from A to B and back, a *time-out* scheme must be used. At the time A sends the *Are-you-up?* message, it specifies a time interval during which it is willing to wait for the reply from B. If A receives the reply message within that time interval, then it can safely conclude that B

is up. If, however, it does not receive the reply message within the time interval (that is, a time-out occurs), then A may conclude only that one or more of the following situations has occurred:

1. Site B is down.
2. The direct link (if one exists) from A to B is down.
3. The alternative path from A to B is down.
4. The message has been lost.

Site A cannot, however, decide which of these events has indeed occurred.

16.4.2 Reconfiguration

Suppose that site A has discovered, through the mechanism described in the previous section, that a failure has occurred. It must then initiate a procedure that will allow the system to reconfigure and to continue its normal mode of operation.

- If a direct link from A to B has failed, this information must be broadcast to every site in the system, so that the various routing tables can be updated accordingly.
- If the system believes that a site has failed (because that site can be reached no longer), then every site in the system must be so notified, so that they will no longer attempt to use the services of the failed site. The failure of a site that serves as a central coordinator for some activity (such as deadlock detection) requires the election of a new coordinator. Similarly, if the failed site is part of a logical ring, then a new logical ring must be constructed. Note that, if the site has not failed (that is, if it is up but cannot be reached), then we may have the undesirable situation where two sites serve as the coordinator. When the network is partitioned, the two coordinators (each for its own partition) may initiate conflicting actions. For example, if the coordinators are responsible for implementing mutual exclusion, we may have a situation where two processes may be executing simultaneously in their critical sections.

16.4.3 Recovery from Failure

When a failed link or site is repaired, it must be integrated into the system gracefully and smoothly.

- Suppose that a link between A and B has failed. When it is repaired, both A and B must be notified. We can accomplish this notification by

continuously repeating the handshaking procedure, described in Section 16.4.1.

- Suppose that site B has failed. When it recovers, it must notify all other sites that it is up again. Site B then may have to receive from the other sites various information to update its local tables; for example, it may need routing table information, a list of sites that are down, or undelivered messages and mail. Note that, if the site has not failed, but simply could not be reached, then this information is still required.

16.5 ■ Design Issues

It has been the challenge of many designers to make the multiplicity of processors and storage devices transparent to the users. Ideally, a distributed system should look to its users like a conventional, centralized system. The user interface of a transparent distributed system should not distinguish between local and remote resources. That is, users should be able to access remote distributed systems as though the latter were local, and it should be the responsibility of the distributed system to locate the resources and to arrange for the appropriate interaction.

Another aspect of transparency is user mobility. It would be convenient to allow users to log in to any machine in the system, and not to force them to use a specific machine. A transparent distributed system facilitates user mobility by bringing over the user's environment (for example, home directory) to wherever she logs in. Both the Andrew file system from CMU and Project Athena from MIT provide this functionality on a large scale. NFS can provide this transparency on a smaller scale.

We use the term *fault tolerance* in a broad sense. Communication faults, machine failures (of type fail-stop), storage-device crashes, and decays of storage media are all considered to be faults that should be tolerated to some extent. A fault-tolerant system should continue to function, perhaps in a degraded form, when faced with these failures. The degradation can be in performance, in functionality, or in both. It should be, however, proportional, in some sense, to the failures that cause it. A system that grinds to a halt when only a few of its components fail is certainly not fault tolerant. Unfortunately, fault tolerance is difficult to implement. Most commercial systems provide only limited tolerance. For instance, the DEC VAXcluster allows multiple computers to share a set of disks. If a system crashes, users may still access their information from another system. Of course, if a disk fails, all the systems will lose access. But in this case, RAID can be used to ensure continued access to the data even in the event of a failure (Section 12.5).

The capability of a system to adapt to increased service load is called *scalability*. Systems have bounded resources and can become completely

saturated under increased load. For example, regarding a file system, saturation occurs either when a server's CPU runs at a high utilization rate, or when disks are almost full. Scalability is a relative property, but it can be measured accurately. A scalable system should react more gracefully to increased load than does a nonscalable one. First, its performance should degrade more moderately than that of a nonscalable system. Second, its resources should reach a saturated state later, when compared with a nonscalable system. Even perfect design cannot accommodate an ever growing load. Adding new resources might solve the problem, but it might generate additional indirect load on other resources (for example, adding machines to a distributed system can clog the network and increase service loads). Even worse, expanding the system can incur expensive design modifications. A scalable system should have the potential to grow without these problems. In a distributed system, the ability to scale up gracefully is of special importance, since expanding the network by adding new machines or interconnecting two networks is commonplace. In short, a scalable design should withstand high service load, accommodate growth of the user community, and enable simple integration of added resources.

Fault tolerance and scalability are related to each other. A heavily loaded component can become paralyzed and behave like a faulty component. Also, shifting the load from a faulty component to that component's backup can saturate the latter. Generally, having spare resources is essential for ensuring reliability as well as for handling peak loads gracefully. An inherent advantage that a distributed system has is a potential for fault tolerance and scalability because of the multiplicity of resources. However, inappropriate design can obscure this potential. Fault-tolerance and scalability considerations call for a design demonstrating distribution of control and data.

Very large-scale distributed systems, to a great extent, are still only theoretical. There are no magic guidelines to ensure the scalability of a system. It is easier to point out why current designs are not scalable. We shall discuss several designs that pose problems, and shall propose possible solutions, all in the context of scalability.

One principle for designing very large-scale systems is the principle that the service demand from any component of the system should be bounded by a constant that is independent of the number of nodes in the system. Any service mechanism whose load demand is proportional to the size of the system is destined to become clogged once the system grows beyond a certain size. Adding more resources would not alleviate such a problem. The capacity of this mechanism simply limits the growth of the system.

Central control schemes and central resources should not be used to build scalable (and fault-tolerant) systems. Examples of centralized entities are central authentication servers, central naming servers, and central file servers. Centralization is a form of functional asymmetry among machines

constituting the system. The ideal alternative is a configuration that is functionally symmetric; that is, all the component machines have an equal role in the operation of the system, and hence each machine has some degree of autonomy. Practically, it is virtually impossible to comply with such a principle. For instance, incorporating diskless machines violates functional symmetry, since the workstations depend on a central disk. However, autonomy and symmetry are important goals to which we should aspire.

The practical approximation to symmetric and autonomous configuration is *clustering*. The system is partitioned into a collection of semiautonomous clusters. A cluster consists of a set of machines and a dedicated cluster server. So that cross-cluster resource references will be relatively infrequent, each machine's requests should be satisfied by its own cluster server most of the time. Of course, this scheme depends on the ability to localize resource references and to place the component units appropriately. If the cluster is well balanced — that is, if the server in charge suffices to satisfy all the cluster demands — it can be used as a modular building block to scale up the system.

A major problem in the design of any service is the process structure of the server. Servers are supposed to operate efficiently in peak periods, when hundreds of active clients need to be served simultaneously. A single-process server is certainly not a good choice, since whenever a request necessitates disk I/O, the whole service will be blocked. Assigning a process for each client is a better choice; however, the expense of frequent context switches between the processes must be considered. A related problem occurs because all the server processes need to share information. It appears that one of the best solutions for the server architecture is the use of lightweight processes or threads, which we discussed in Section 4.5. The abstraction presented by a group of lightweight processes is that of multiple threads of control associated with some shared resources. Usually, a lightweight process is not bound to a particular client. Instead, it serves single requests of different clients. Scheduling threads can be preemptive or nonpreemptive. If threads are allowed to run to completion (nonpreemptive), then their shared data do not need to be protected explicitly. Otherwise, some explicit locking mechanism must be used. It is clear that some form of lightweight-processes scheme is essential for servers to be scalable.

16.6 ■ Summary

A distributed system provides the user with access to the various resources the system provides. Access to a shared resource can be provided by data migration, computation migration, or job migration. A distributed file system must address two major issues: transparency (does a user access all

files in the same manner regardless of where they are in the network?) and locality (where do files reside in the system?).

A distributed file system is built on top of lower-level networking functions. The most common lower-level function is the remote procedure call (RPC) mechanism. An RPC is a structured message addressed to an RPC daemon listening to a *port* on the remote system, and contains an identifier of the function to execute and the parameters to pass to that function. The function is then executed as requested, and any output is sent back to the requester in a separate message. Threads can be used to make the destination process easier to write and more efficient, as one thread can process a request from beginning to end while its fellow threads are doing the same for other requests.

A distributed system may suffer from various types of hardware failure. For the system to be fault tolerant, it must detect failures and reconfigure the system. When the failure is repaired, the system must be reconfigured again.

■ Exercises

16.1 What are the advantages and disadvantages of making the computer network transparent to the user?

16.2 What are the formidable problems that designers must solve to implement a network transparent system?

16.3 Process migration within a heterogeneous network is usually impossible, given the differences in architectures and operating systems. Describe a method for process migration across different architectures running:

a. The same operating system

b. Different operating systems

16.4 To build a robust distributed system, you must know what kinds of failures can occur.

a. List possible types of failure in a distributed system.

b. Specify which items in your list also are applicable to a centralized system.

16.5 Is it always crucial to know that the message you have sent has arrived at its destination safely? If your answer is "yes," explain why. If your answer is "no," give appropriate examples.

16.6 Present an algorithm for reconstructing a logical ring after a process in the ring fails.

16.7 Consider a distributed system with two sites, A and B. Consider whether site A can distinguish among the following:

a. B goes down.

b. The link between A and B goes down.

c. B is extremely overloaded and response time is 100 times longer than normal.

What implications does your answer have for recovery in distributed systems?

Bibliographic Notes

Forsdick et al. [1978] and Donnelley [1979] discussed operating systems for computer networks. A survey of distributed operating systems was offered by Tanenbaum and Van Renesse [1985].

Discussions concerning distributed operating-system structures have been offered by Popek and Walker [1985] (the Locus system), Cheriton and Zwaenepoel [1983] (the V kernel), Ousterhout et al. [1988] (the Sprite network operating system), Balkovich et al. [1985] (Project Athena) and Tanenbaum et al. [1990] and Mullender et al. [1990] (the Amoeba distributed operating system). A comparison of Amoeba and Sprite is offered by Douglis et al. [1991].

Discussions concerning load balancing and load sharing were presented by Chow and Abraham [1982], Eager et al. [1986], and Ferguson et al. [1988]. Discussions concerning process migration were presented by Eager et al. [1986], Zayas [1987], Smith [1988], Jul et al. [1988], Artsy [1989a], Douglis and Ousterhout [1987, 1989], and Eskicioglu [1990]. A special issue on process migration was edited by Artsy [1989b].

Schemes for sharing idle workstations in a distributed shared computing environment are presented by Nichols [1987], Mutka and Livny [1987], and Litzkow et al. [1988].

Reliable communication in the presense of failures was discussed by Birman and Joseph [1987]. The principle that the service demand from any component of the system should be bounded by a constant that is independent of the number of nodes in the system was first advanced by Barak and Kornatzky [1987].

CHAPTER 17

DISTRIBUTED FILE SYSTEMS

In the previous chapter, we discussed network construction and the low-level protocols needed for messages to be transferred between systems. Now we discuss one use of this infrastructure. A *distributed file system* (DFS) is a distributed implementation of the classical time-sharing model of a file system, where multiple users share files and storage resources (Chapter 10). The purpose of a DFS is to support the same kind of sharing when the files are physically dispersed among the various sites of a distributed system.

In this chapter, we discuss the various ways a DFS can be designed and implemented. First, we discuss common concepts on which DFSs are based. Then, we illustrate our concepts by examining the UNIX United, NFS, Andrew, Sprite, and Locus DFSs. We take this approach to the presentation because distributed systems is an active research area, and the many design tradeoffs we shall illuminate are still being examined. By exploring these example systems, we hope to provide a sense of the considerations involved in designing an operating system, and also to indicate current areas of operating-system research.

17.1 ■ Background

A distributed system is a collection of loosely coupled machines interconnected by a communication network. We use the term *machine* to denote either a mainframe or a workstation. From the point of view of a specific machine in a distributed system, the rest of the machines and their

respective resources are *remote*, whereas the machine's own resources are referred to as *local*.

To explain the structure of a DFS, we need to define the terms *service*, *server*, and *client*. A *service* is a software entity running on one or more machines and providing a particular type of function to a priori unknown clients. A *server* is the service software running on a single machine. A *client* is a process that can invoke a service using a set of operations that forms its *client interface*. Sometimes, a lower-level interface is defined for the actual cross-machine interaction, which we refer to as the *intermachine interface*.

Using this terminology, we say that a file system provides file services to clients. A client interface for a file service is formed by a set of primitive *file operations*, such as create a file, delete a file, read from a file, and write to a file. The primary hardware component that a file server controls is a set of local secondary-storage devices (usually, magnetic disks), on which files are stored, and from which they are retrieved according to the client requests.

A DFS is a file system whose clients, servers, and storage devices are dispersed among the machines of a distributed system. Accordingly, service activity has to be carried out across the network, and instead of a single centralized data repository, there are multiple and independent storage devices. As will become evident, the concrete configuration and implementation of a DFS may vary. There are configurations where servers run on dedicated machines, as well as configurations where a machine can be both a server and a client. A DFS can be implemented as part of a distributed operating system, or alternatively by a software layer whose task is to manage the communication between conventional operating systems and file systems. The distinctive features of a DFS are the multiplicity and autonomy of clients and servers in the system.

Ideally, a DFS should look to its clients like a conventional, centralized file system. The multiplicity and dispersion of its servers and storage devices should be made transparent. That is, the client interface of a DFS should not distinguish between local and remote files. It is up to the DFS to locate the files and to arrange for the transport of the data. A transparent DFS facilitates user mobility by bringing over the user's environment (that is, home directory) to wherever a user logs in.

The most important *performance* measurement of a DFS is the amount of time needed to satisfy various service requests. In conventional systems, this time consists of disk access time and a small amount of CPU processing time. In a DFS, however, a remote access has the additional overhead attributed to the distributed structure. This overhead includes the time needed to deliver the request to a server, as well as the time for getting the response across the network back to the client. For each direction, in addition to the actual transfer of the information, there is the CPU overhead of running the communication protocol software. The performance of a

DFS can be viewed as another dimension of the DFS's transparency. That is, the performance of an ideal DFS would be comparable to that of a conventional file system.

The fact that a DFS manages a set of dispersed storage devices is the DFS's key distinguishing feature. The overall storage space managed by a DFS is composed of different, and remotely located, smaller storage spaces. Usually, there is correspondence between these constituent storage spaces and sets of files. We use the term *component unit* to denote the smallest set of files that can be stored on a single machine, independently from other units. All files belonging to the same component unit must reside in the same location.

17.2 ■ Naming and Transparency

Naming is a mapping between logical and physical objects. For instance, users deal with logical data objects represented by file names, whereas the system manipulates physical blocks of data, stored on disk tracks. Usually, a user refers to a file by a textual name. The latter is mapped to a lower-level numerical identifier that in turn is mapped to disk blocks. This multilevel mapping provides users with an abstraction of a file that hides the details of how and where on the disk the file is actually stored.

In a *transparent* DFS, a new dimension is added to the abstraction: that of hiding where in the network the file is located. In a conventional file system, the range of the naming mapping is an address within a disk. In a DFS, this range is augmented to include the specific machine on whose disk the file is stored. Going one step further with the concept of treating files as abstractions leads to the possibility of *file replication*. Given a file name, the mapping returns a set of the locations of this file's replicas. In this abstraction, both the existence of multiple copies and their location are hidden.

17.2.1 Naming Structures

There are two related notions regarding name mappings in a DFS that need to be differentiated:

- **Location transparency:** The name of a file does not reveal any hint of the file's physical storage location.
- **Location independence:** The name of a file does not need to be changed when the file's physical storage location changes.

Both definitions are relative to the level of naming discussed previously, since files have different names at different levels (that is, user-level textual names, and system-level numerical identifiers). A location-independent

naming scheme is a dynamic mapping, since it can map the same file name to different locations at two different times. Therefore, location independence is a stronger property than is location transparency.

In practice, most of the current DFSs provide a static, location-transparent mapping for user-level names. These systems, however, do not support file *migration;* that is, changing the location of a file automatically is impossible. Hence, the notion of location independence is quite irrelevant for these systems. Files are associated permanently with a specific set of disk blocks. Files and disks can be moved between machines manually, but file migration implies an automatic, operating-system initiated action. Only Andrew (Section 17.6.3) and some experimental file systems support location independence and file mobility. Andrew supports file mobility mainly for administrative purposes. A protocol provides migration of Andrew's component units to satisfy high-level user requests, without changing either the user-level names, or the low-level names of the corresponding files.

There are a few aspects that can further differentiate location independence and static location transparency:

- Divorcing data from location, as exhibited by location independence, provides better abstraction for files. A file name should denote the file's most significant attributes, which are its contents, rather than its location. Location-independent files can be viewed as logical data containers that are not attached to a specific storage location. If only static location transparency is supported, the file name still denotes a specific, although hidden, set of physical disk blocks.

- Static location transparency provides users with a convenient way to share data. Users can share remote files by simply naming the files in a location-transparent manner, as though the files were local. Nevertheless, sharing the storage space is cumbersome, because logical names are still statically attached to physical storage devices. Location independence promotes sharing the storage space itself, as well as the data objects. When files can be mobilized, the overall, system-wide storage space looks like a single, virtual resource. A possible benefit of such a view is the ability to balance the utilization of disks across the system.

- Location independence separates the naming hierarchy from the storage-devices hierarchy and from the intercomputer structure. By contrast, if static location transparency is used (although names are transparent), we can easily expose the correspondence between component units and machines. The machines are configured in a pattern similar to the naming structure. This may restrict the architecture of the system unnecessarily and conflict with other considerations. A server in charge of a root directory is an example of a

structure that is dictated by the naming hierarchy and contradicts decentralization guidelines.

Once the separation of name and location has been completed, files residing on remote server systems may be accessed by various clients. In fact, these clients may be *diskless* and rely on servers to provide all files, including the operating-system kernel. Special protocols are needed for the boot sequence, however. Consider the problem of getting the kernel to a diskless workstation. The diskless workstation has no kernel, so it cannot use the DFS code to retrieve the kernel. Instead, a special boot protocol, stored in read-only memory (ROM) on the client, is invoked. It enables networking and retrieves only one special file (the kernel or boot code) from a fixed location. Once the kernel is copied over the network and loaded, its DFS makes all the other operating-system files available. The advantages of diskless clients are many, including lower cost (because no disk is needed on each machine) and greater convenience (when an operating-system upgrade occurs, only the server copy needs to be modified, rather than all the clients as well). The disadvantages are the added complexity of the boot protocols and the performance loss resulting from the use of a network rather than of a local disk.

The current trend is toward clients with local disks. Disk drives are increasing in capacity and decreasing in cost rapidly, with new generations appearing every year or so. The same cannot be said for networks, which evolve every 5 to 10 years. Overall, systems are growing more quickly than are networks, so extra efforts are needed to limit network access to improve system throughput.

17.2.2 Naming Schemes

There are three main approaches to naming schemes in a DFS. In the simplest approach, files are named by some combination of their host name and local name, which guarantees a unique systemwide name. In Ibis, for instance, a file is identified uniquely by the name *host:local-name*, where *local-name* is a UNIX-like path. This naming scheme is neither location transparent nor location independent. Nevertheless, the same file operations can be used for both local and remote files. The structure of the DFS is a collection of isolated component units that are entire conventional file systems. In this first approach, component units remained isolated, although means are provided to refer to a remote file. We do not consider this scheme any further in this text.

The second approach was popularized by Sun's Network File System (NFS). NFS is the file system component of ONC+, a networking package which will be supported by many UNIX vendors. NFS provides means to attach remote directories to local directories, thus giving the appearance of a coherent directory tree. Early NFS versions only allowed previously

mounted remote directories to be accessed transparently. With the advent of the *automount* feature, mounts occur on-demand based on a table of mount points and file structure names. There is also some integration of components to support transparent sharing. This integration, however, is limited and is not uniform, because each machine may attach different remote directories to its tree. The resulting structure is versatile. Usually, it is a forest of UNIX trees with shared subtrees.

Total integration of the component file systems is achieved using the third approach. A single global name structure spans all the files in the system. Ideally, the composed file-system structure should be isomorphic to the structure of a conventional file system. In practice, however, there are many special files (for example, UNIX device files and machine-specific binary directories) that make this goal difficult to attain. We shall examine different variations of this approach in our discussions of UNIX United, Locus, Sprite, and Andrew in Section 17.6.

An important criterion for evaluation of the naming structures is their administrative complexity. The most complex and most difficult structure to maintain is the NFS structure. Because any remote directory can be attached anywhere onto the local directory tree, the resulting hierarchy can be highly unstructured. The effect of a server becoming unavailable is that some arbitrary set of directories on different machines becomes unavailable. In addition, a separate accreditation mechanism is used to control which machine is allowed to attach which directory to its tree. This mechanism can lead to the situation in which, on one client, a user can access a remote directory tree, whereas on another client, access is denied to that user.

17.2.3 Implementation Techniques

Implementation of transparent naming requires a provision for the mapping of a file name to the associated location. Keeping this mapping manageable calls for aggregating sets of files into component units, and providing the mapping on a component unit basis rather than on a single-file basis. This aggregation serves administrative purposes as well. UNIX-like systems use the hierarchical directory tree to provide name-to-location mapping, and to aggregate files recursively into directories.

To enhance the availability of the crucial mapping information, we can use methods such as replication, local caching, or both. As we already noted, location independence means that the mapping changes over time; hence, replicating the mapping renders a simple yet consistent update of this information impossible. A technique to overcome this obstacle is to introduce low-level, *location-independent file identifiers*. Textual file names are mapped to lower-level file identifiers that indicate to which component unit the file belongs. These identifiers are still location independent. They can be replicated and cached freely without being invalidated by migration

of component units. A second level of mapping, which maps component units to locations and needs a simple yet consistent update mechanism, is the inevitable price. Implementing UNIX-like directory trees using these low-level, location-independent identifiers makes the whole hierarchy invariant under component unit migration. The only aspect that does change is the component unit-location mapping.

A common way to implement these low-level identifiers is to use structured names. These names are bit strings that usually have two parts. The first part identifies the component unit to which the file belongs; the second part identifies the particular file within the unit. Variants with more parts are possible. The invariant of structured names, however, is that individual parts of the name are unique at all times only within the context of the rest of the parts. We can obtain uniqueness at all times by taking care not to reuse a name that is still used, by adding sufficiently more bits (this method is used in Andrew), or by using a timestamp as one of the parts of the name (as done in Apollo Domain). Another way to view this process is that we are taking a location-transparent system, such as Ibis, and adding another level of abstraction to produce a location-independent naming scheme.

The use of the techniques of aggregation of files into component units, and of lower-level location-independent file identifiers, is exemplified in Andrew and Locus.

17.3 ■ Remote File Access

Consider a user who requests access to a remote file. Assuming that the server storing the file was located by the naming scheme, the actual data transfer to satisfy the user request for the remote access must take place.

One way to achieve this transfer is through a *remote-service* mechanism, where requests for accesses are delivered to the server, the server machine performs the accesses, and their results are forwarded back to the user. One of the most common ways of implementing remote service is the *remote procedure call (RPC)* paradigm, which we discussed in Section 16.3.1. We note that there is a direct analogy between disk-access methods in conventional file systems and the remote-service method in a DFS. The remote-service method is analogous to performing a disk access for each access request.

To ensure reasonable performance of a remote-service mechanism, we can use a form of *caching*. In conventional file systems, the rationale for caching is to reduce disk I/O (thereby increasing performance), whereas in DFSs, the goal is to reduce both network traffic and disk I/O. In the following, we discuss various issues concerning the implementation of caching in a DFS, and contrast the latter with the basic remote-service paradigm.

17.3.1 Basic Caching Scheme

The concept of caching is simple. If the data needed to satisfy the access request are not already cached, then a copy of those data is brought from the server to the client system. Accesses are performed on the cached copy. The idea is to retain recently accessed disk blocks in the cache, so that repeated accesses to the same information can be handled locally, without additional network traffic. A replacement policy (for example, least recently used) is used to keep the cache size bounded. There is no direct correspondence between accesses and traffic to the server. Files are still identified with one master copy residing at the server machine, but copies of (parts of) the file are scattered in different caches. When a cached copy is modified, the changes need to be reflected on the master copy to preserve the relevant consistency semantics. The problem of keeping the cached copies consistent with the master file is the *cache-consistency problem*, which will be discussed in Section 17.3.4. Observant readers will realize that DFS caching could just as easily be called *network virtual memory:* It acts similarly to demand-paged virtual memory, except that the backing store is not a local disk, but rather is a remote server.

The granularity of the cached data can vary from blocks of a file to an entire file. Usually, more data are cached than are needed to satisfy a single access, so that many accesses can be served by the cached data. This procedure is much like disk read-ahead (Section 11.5.2). The Andrew system caches files in large chunks (64K). The other systems discussed in this chapter support caching of individual blocks driven by clients' demands. Increasing the caching unit increases the hit ratio, but also increases the miss penalty because each miss requires more data to be transferred. It also increases the potential for consistency problems. Selecting the unit of caching involves considering parameters such as the network transfer unit and the RPC protocol service unit (in case an RPC protocol is used). The network transfer unit (for Ethernet, a packet) is about 1.5K, so larger units of cached data need to be disassembled for delivery and reassembled on reception.

Block size and the total cache size are obviously of importance for block-caching schemes. In UNIX-like systems, common block sizes are 4K or 8K. For large caches (over 1 megabyte), large block sizes (over 8K) are beneficial. For smaller caches, large block sizes are less beneficial because they result in fewer blocks in the cache.

17.3.2 Cache Location

Now we turn to the issue of where the cached data should be stored. Disk caches have one clear advantage over main memory cache — reliability. Modifications to cached data are lost in a crash if the cache is kept in volatile memory. Moreover, if the cached data are kept on disk, they are

still there during recovery and there is no need to fetch them again. On the other hand, main-memory caches have several advantages of their own:

- Main-memory caches permit workstations to be diskless.
- Data can be accessed more quickly from a cache in main memory than from one on a disk.
- The current technology trend is toward bigger and less expensive memories. The achieved performance speedup is predicted to outweigh the advantages of disk caches.
- The server caches (the ones used to speed up disk I/O) will be in main memory regardless of where user caches are located; by using main-memory caches on the user machine too, we can build a single caching mechanism for use by both servers and users (as is done in Sprite).

Many remote-access implementations can be thought of as a hybrid of caching and remote service. In NFS and Locus, for instance, the implementation is based on remote service but is augmented with caching for performance. On the other hand, Sprite's implementation is based on caching, but under certain circumstances a remote service method is adopted. Thus, when we evaluate the two methods, we actually evaluate to what degree one method is emphasized over the other.

17.3.3 Cache Update Policy

The policy used to write modified data blocks back to the server's master copy has critical effect on the system's performance and reliability. The simplest policy is to write data through to disk as soon as they are placed on any cache. The advantage of *write-through* is its reliability. Little information is lost when a client system crashes. However, this policy requires each write access to wait until the information is sent to the server, which results in poor write performance. Caching with write-through is equivalent to using remote service for write accesses and exploiting caching only for read accesses. NFS provides write-through access.

An alternate write policy is to delay updates to the master copy. Modifications are written to the cache and then are written through to the server at a later time. This policy has two advantages over write-through. First, because writes are to the cache, write accesses complete much more quickly. Second, data may be overwritten before they are written back, in which case all but the last update never needs to be written at all. Unfortunately, *delayed-write* schemes introduce reliability problems, since unwritten data will be lost whenever a user machine crashes.

There are several variations of the delayed-write policy that differ in when modified data blocks are flushed to the server. One alternative is to flush a block when it is about to be ejected from the client's cache. This option can result in good performance, but some blocks can reside in the client's cache a long time before they are written back to the server. A compromise between this alternative and the write-through policy is to scan the cache at regular intervals and to flush blocks that have been modified since the last scan, just as UNIX scans its local cache. Sprite uses this policy with a 30-second interval.

Yet another variation on delayed-write is to write data back to the server when the file is closed. This policy, *write-on-close*, is used in the Andrew system. In the case of files that are open for short periods or are modified rarely, this policy does not significantly reduce network traffic. In addition, the write-on-close policy requires the closing process to delay while the file is written through, which reduces the performance advantages of delayed writes. The performance advantages of this policy over delayed-write with more frequent flushing are apparent for files that are open for long periods and are modified frequently.

17.3.4 Consistency

A client machine is faced with the problem of deciding whether or not a locally cached copy of the data is consistent with the master copy (and hence can be used). If the client machine determines that its cached data are out of date, accesses can no longer be served by those cached data. An up-to-date copy of the data needs to be cached. There are two approaches to verify the validity of cached data:

- **Client-initiated approach:** The client initiates a validity check in which it contacts the server and checks whether the local data are consistent with the master copy. The frequency of the validity check is the crux of this approach and determines the resulting consistency semantics. It can range from a check before every access, to a check on only first access to a file (on file open, basically). Every access that is coupled with a validity check is delayed, compared with an access served immediately by the cache. Alternatively, a check can be initiated every fixed interval of time. Depending on its frequency, the validity check can load both the network and the server.

- **Server-initiated approach:** The server records, for each client, the (parts of) files that it caches. When the server detects a potential inconsistency, it must react. A potential for inconsistency occurs when a file is cached by two different clients in conflicting modes. If session semantics (Section 10.5.2) are implemented, then, whenever a server

receives a request to close a file that has been modified, it should react by notifying the clients to consider the cached data invalid and discard that data. Clients having this file open at that time discard their copy when the current session is over. Other clients discard their copy at once. Under session semantics, the server does not need to be informed about opens of already cached files. The server's reaction is triggered only by the close of a writing session, and hence only this kind of session is delayed. In Andrew, session semantics are implemented, and a server-initiated method, called *callback*, is employed (Section 17.6.3).

On the other hand, if a more restrictive consistency semantics, such as UNIX semantics (Section 10.5.1), is implemented, the server must play a more active role. The server must be notified whenever a file is opened and the intended mode (read or write mode) must be indicated for every open. Assuming such notification, the server can act when it detects a file that is opened simultaneously in conflicting modes by disabling caching for that particular file (as done in Sprite). Actually, disabling caching results in switching to a remote-service mode of operation.

17.3.5 A Comparison of Caching and Remote Service

Essentially, the choice between caching and remote service trades off a potentially increased performance with decreased simplicity. We evaluate this tradeoff by listing the advantages and disadvantages of the two methods:

- A substantial number of the remote accesses can be handled efficiently by the local cache when caching is used. Capitalizing on locality in file-access patterns makes caching even more attractive. Thus, most of the remote accesses will be served as fast as will local ones. Moreover, servers are contacted only occasionally, rather than for each access. Consequently, server load and network traffic are reduced, and the potential for scalability is enhanced. By contrast, every remote access is handled across the network when the remote-service method is used. The penalty in network traffic, server load, and performance is obvious.
- Total network overhead in transmitting big chunks of data (as is done in caching) is lower than when series of responses to specific requests are transmitted (as in the remote-service method).
- Disk-access routines on the server may be better optimized if it is known that requests are always for large, contiguous segments of data, rather than for random disk blocks.

- The cache-consistency problem is the major drawback of caching. In access patterns that exhibit infrequent writes, caching is superior. However, when writes are frequent, the mechanisms employed to overcome the consistency problem incur substantial overhead in terms of performance, network traffic, and server load.
- So that caching will confer a benefit, execution should be carried out on machines that have either local disks or large main memories. Remote access on diskless, small-memory-capacity machines should be done through the remote-service method.
- In caching, since data are transferred en masse between the server and client, rather than in response to the specific needs of a file operation, the lower intermachine interface is quite different from the upper-user interface. The remote-service paradigm, on the other hand, is just an extension of the local file-system interface across the network. Thus, the intermachine interface mirrors the local user-file-system interface.

17.4 ■ Stateful versus Stateless Service

There are two approaches to server-side information. Either the server tracks each file being accessed by each client, or it simply provides blocks as they are requested by the client without knowledge of the blocks' usage.

The typical scenario of a *stateful file service* is as follows. A client must perform an open on a file before accessing that file. The server fetches some information about the file from its disk, stores it in its memory, and gives the client a connection identifier that is unique to the client and the open file. (In UNIX terms, the server fetches the inode and gives the client a file descriptor, which serves as an index to an in-core table of inodes.) This identifier is used for subsequent accesses until the session ends. A stateful service is characterized as a connection between the client and the server during a session. Either on closing the file, or by a garbage-collection mechanism, the server must reclaim the main-memory space used by clients who are no longer active.

The advantage of stateful service is increased performance. File information is cached in main memory and can be accessed easily via the connection identifier, thereby saving disk accesses. In addition, a stateful server would know whether a file were open for sequential access and could therefore read ahead the next blocks. Stateless servers cannot do so, since they have no knowledge of the purpose of the client's requests. The key point regarding fault tolerance in a stateful service approach is that main-memory information is kept by the server about its clients.

A *stateless file server* avoids this state information by making each request self-contained. That is, each request identifies the file and the position in the file (for read and write accesses) in full. The server does not

need to keep a table of open files in main memory, although it usually does so for efficiency reasons. Moreover, there is no need to establish and terminate a connection by open and close operations. They are totally redundant, since each file operation stands on its own and is not considered as part of a session. A client process would open a file, and that open would not result in a remote message being sent. Reads and writes would, of course, take place as remote messages (or cache lookups). The final close by the client would again result in only a local operation.

The distinction between stateful and stateless service becomes evident when we consider the effects of a crash occurring during a service activity. A stateful server loses all its volatile state in a crash. Ensuring the graceful recovery of such a server involves restoring this state — usually by a recovery protocol based on a dialog with clients. Less graceful recovery requires that the operations that were underway when the crash occurred, be aborted. A different problem is caused by client failures. The server needs to become aware of such failures, so that it can reclaim space allocated to record the state of crashed client processes. This phenomenon is sometimes referred to as *orphan detection and elimination*.

A stateless server avoids these problems, since a newly reincarnated server can respond to a self-contained request without any difficulty. Therefore, the effects of server failures and recovery are almost unnoticeable. There is no difference between a slow server and a recovering server from a client's point of view. The client keeps retransmitting its request if it receives no response.

The penalty for using the robust stateless service is longer request messages, and slower processing of requests, since there is no in-core information to speed the processing. In addition, stateless service imposes additional constraints on the design of the DFS. First, since each request identifies the target file, a uniform, systemwide, low-level naming scheme should be used. Translating remote to local names for each request would cause even slower processing of the requests. Second, since clients retransmit requests for file operations, these operations must be *idempotent;* that is, each operation must have the same effect and return the same output if executed several times consecutively. Self-contained read and write accesses are idempotent, as long as they use an absolute byte count to indicate the position within the file they access and do not rely on an incremental offset (as done in UNIX **read** and **write** system calls). However, we must be careful when implementing destructive operations (such as delete a file) to make them idempotent too.

In some environments, a stateful service is a necessity. If the server employs the server-initiated method for cache validation, it cannot provide stateless service, since it maintains a record of which files are cached by which clients.

The way UNIX uses file descriptors and implicit offsets is inherently stateful. Servers must maintain tables to map the file descriptors to inodes,

and store the current offset within a file. This is why NFS, which employs a stateless service, does not use file descriptors, and does include an explicit offset in every access.

17.5 ■ File Replication

Replication of files on different machines is a useful redundancy for improving availability. Multimachine replication can benefit performance too, since selecting a nearby replica to serve an access request results in shorter service time.

The basic requirement of a replication scheme is that different replicas of the same file reside on failure-independent machines. That is, the availability of one replica is not affected by the availability of the rest of the replicas. This obvious requirement implies that replication management is inherently a location-opaque activity. Provisions for placing a replica on a particular machine must be available.

It is desirable to hide the details of replication from users. It is the task of the naming scheme to map a replicated file name to a particular replica. The existence of replicas should be invisible to higher levels. However, the replicas must be distinguished from one another by different lower-level names. Another transparency issue is providing replication control at higher levels. Replication control includes determination of the degree of replication and of the placement of replicas. Under certain circumstances, it is desirable to expose these details to users. Locus, for instance, provides users and system administrators with mechanisms to control the replication scheme.

The main problem associated with replicas is their update. From a user's point of view, replicas of a file denote the same logical entity, and thus an update to any replica must be reflected on all other replicas. More precisely, the relevant consistency semantics must be preserved when accesses to replicas are viewed as virtual accesses to the replicas' logical files. If consistency is not of primary importance, it can be sacrificed for availability and performance. This option is an incarnation of a fundamental tradeoff in the area of fault tolerance. The choice is between preserving consistency at all costs, thereby creating a potential for indefinite blocking, and sacrificing consistency under some (we hope rare) circumstance of catastrophic failures for the sake of guaranteed progress. Among the surveyed systems, Locus employs replication extensively and sacrifices consistency in the case of network partition, for the sake of availability of files for both read and write accesses (see Section 17.6.5 for details).

As an illustration of these concepts, we describe the replication scheme of Ibis, which uses a variation of the primary-copy approach. The domain of the name mapping is a pair <primary-replica-identifier, local-replica-

identifier>. Since there may be no replica locally, a special value is used in that case. Thus, the mapping is relative to a machine. If the local replica is the primary one, the pair contains two identical identifiers. Ibis supports demand replication, which is an automatic replication-control policy (similar to whole-file caching). Under demand replication, reading a nonlocal replica causes it to be cached locally, thereby generating a new nonprimary replica. Updates are performed on only the primary copy and cause all other replicas to be invalidated by sending appropriate messages. Atomic and serialized invalidation of all nonprimary replicas is not guaranteed. Hence, a stale replica may be considered valid. To satisfy remote write accesses, we migrate the primary copy to the requesting machine.

17.6 ■ Example Systems

In this section, we illustrate the common concepts on which DFSs are based by examining five different and interesting DFSs: UNIX United, Sun NFS, Andrew, Sprite, and Locus.

17.6.1 UNIX United

The *UNIX United* project from the University of Newcastle upon Tyne, England, is one of the earliest attempts to scale up the UNIX file system to a distributed one without modifying the UNIX kernel. In UNIX United, a software subsystem is added to each of a set of interconnected UNIX systems (referred to as *component* or *constituent* systems), to construct a distributed system that is functionally indistinguishable from a conventional centralized UNIX system. The system is presented in two levels of detail. First, an overview of UNIX United is given. Then, the implementation, the Newcastle Connection layer, is examined and issues regarding networking and internetworking are discussed.

17.6.1.1 Overview

Any number of interlinked UNIX systems can be joined to compose a UNIX United system. Their naming structures (for files, devices, directories, and commands) are joined together into a single naming structure, in which each component system is, for all intents and purposes, just a directory. Ignoring for the moment questions regarding accreditation and access control, the resulting system is one where each user can read or write any file, use any device, execute any command, or inspect any directory, regardless of the system to which it belongs.

The component unit is a complete UNIX directory tree belonging to a certain machine. The position of these component units in the naming hierarchy is arbitrary. They can appear in the naming structure in positions

subservient to other component units (directly or through intermediary directories).

Roots of component units are assigned names so that they become accessible and distinguishable externally. A file system's own root is still referred to as "/" and still serves as the starting point of all path names starting with a "/". However, a subservient file system can access its superior system by referring to its own root parent (that is, "/.."). Therefore, there is only one root that is its own parent and that is not assigned a string name — namely, the root of the composite name structure, which is just a virtual node needed to make the whole structure a single tree. Under this convention, there is no notion of absolute path name. Each path name is relative to some context, either to the current working directory or to the current component unit.

In Figure 17.1, *unix1*, *unix2*, *unix3*, and *unix4* are names of component systems. As an illustration of the relative path names, note that within the *unix1* system, file *f2* on the system *unix2* is referred to as */../unix2/f2*. Within the *unix3* system, this file is referred to as */../../unix2/f2*. Now, suppose that the current root ("/") is as shown by the arrow. Then, file *f3* is referred to as */f3*, file *f1* is referred to as */../f1*, file *f2* is referred to as */../../unix2/f2*, and finally file *f4* is referred to as */../../unix2/dir/unix4/f4*.

Observe that users are aware of the upward boundaries of their current component unit, since they must use the "../" syntax whenever they wish to ascend outside of their current machine. Hence, UNIX United does not provide complete location transparency.

The traditional root directories (for example, */dev*, */temp*) are maintained for each machine separately. Because of the relative naming scheme they are named, from within a component system, in exactly the same manner as in a conventional UNIX.

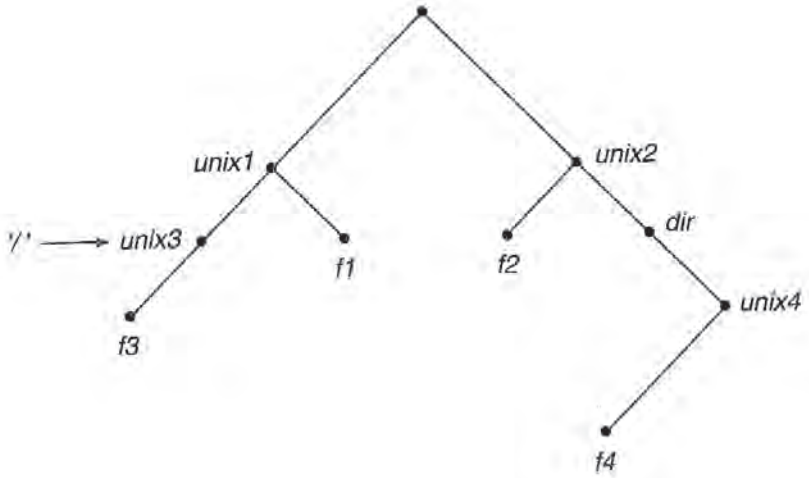


Figure 17.1 Example of a UNIX United directory structure.

Each component system has its own set of named users and its own administrator (superuser). The latter is responsible for the accreditation for users of his own system, as well as for that of remote users. A remote user's identifier is prefixed with the name of the user's original system for uniqueness. Accesses are governed by the standard UNIX file-protection mechanisms, even if they cross components' boundaries. That is, there is no need for users to log in separately, or to provide passwords, when they access remote files. However, users wishing to access files in a remote system must arrange with the specific system administrator separately.

It is often convenient to set the naming structure so as to reflect organizational hierarchy of the environment in which the system exists.

17.6.1.2 The Newcastle Connection

The Newcastle Connection is a (user-level) software layer incorporated in each component system. This connection layer separates the UNIX kernel on one hand, and the user-level programs on the other hand (see Figure 17.2). It intercepts all system calls concerning files, and filters out those that have to be redirected to remote systems. Also, the connection layer accepts system calls that have been directed to it from other systems. Remote layers communicate by the means of an RPC protocol.

The connection layer preserves the same UNIX system-call interface as that of the UNIX kernel, in spite of the extensive remote activity that the system carries out. The penalty of preserving the kernel intact is that the service is implemented as user-level daemon processes, which slow down remote operation.

Each connection layer stores a partial skeleton of the overall naming structure. Obviously, each system stores locally its own file system. In addition, each system stores fragments of the overall name structure that relate it to its neighboring systems in the naming structure (that is, systems that can be reached via traversal of the naming tree without passing through another system). In Figure 17.3, we show the partial skeletons of the hierarchy of the file systems of Figure 17.1 as maintained by the systems *unix1*, *unix2*, and *unix3*, respectively (only the relevant parts are shown).

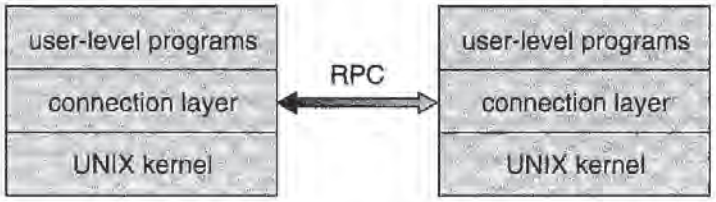


Figure 17.2 Schematic view of UNIX United architecture.

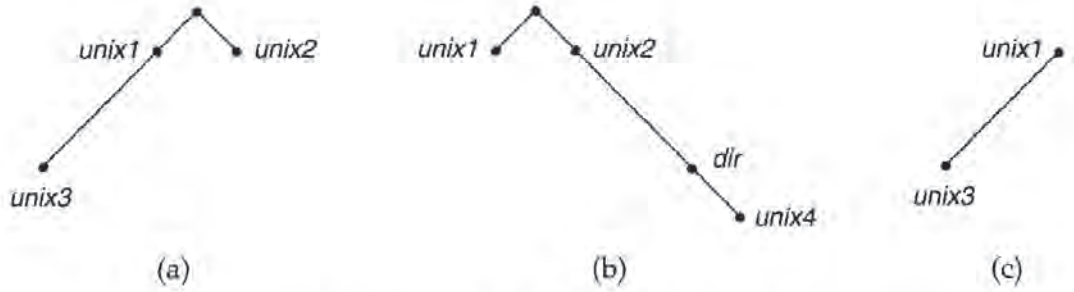


Figure 17.3 The file system of (a) *unix1*, (b) *unix2*, and (c) *unix3*.

The fragments maintained by different systems overlap and hence must remain consistent, a requirement that makes changing the overall structure an infrequent event. Some leaves of the partial structure stored locally correspond to remote roots of other parts of the global file system. These leaves are specially marked, and contain addresses of the appropriate storage sites of the descending file systems. Path-name traversals have to be continued remotely when such marked leaves are encountered, and, in fact, can span several systems until the target file is located. Once a name is resolved and the file is opened, that file is accessed using file descriptors. The connection layer marks descriptors that refer to remote files, and keeps network addresses and routing information for them in a per-process table.

The actual remote file accesses are carried out by a set of file-server processes on the target system. Each client has its own file server process with which it communicates directly. The initial connection is established with the aid of a *spawner* process that has a standard fixed name that makes it callable from any external process. This spawner process performs the remote access-rights checks according to a machine-user identification pair. Also, it converts this identification to a valid local name. So that UNIX semantics will be preserved, once a client process forks, its file service process forks as well. This service scheme does not excel in terms of robustness. Special recovery actions have to be taken in case of simultaneous server and client failures. However, the connection layer attempts to mask and isolate failures resulting from the fact that the system is a distributed one.

17.6.2 The Sun Network File System

The Network File System (NFS) is both an implementation and a specification of a software system for accessing remote files across LANs (or even WANs). NFS is part of ONC+, which most UNIX vendors are supporting. The implementation is part of the Solaris operating system, which is a modified version of UNIX SVR4, running on Sun workstations and

other hardware. It uses the unreliable datagram protocol (UDP/IP protocol) and Ethernet (or another networking system). The specification and the implementation are intertwined in our description of NFS. Whenever a level of detail is needed, we refer to the Sun implementation; whenever the description is general enough, it applies to the specification also.

17.6.2.1 Overview

NFS views a set of interconnected workstations as a set of independent machines with independent file systems. The goal is to allow some degree of sharing among these file systems (on explicit request) in a transparent manner. Sharing is based on server–client relationship. A machine may be, and often is, both a client and a server. Sharing is allowed between any pair of machines, rather than with only dedicated server machines. To ensure machine independence, sharing of a remote file system affects only the client machine and no other machine.

So that a remote directory will be accessible in a transparent manner from a particular machine — say, from *M1* — a client of that machine has to carry out a mount operation first. The semantics of the operation are that a remote directory is mounted over a directory of a local file system. Once the mount operation is completed, the mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory. The local directory becomes the name of the root of the newly mounted directory. Specification of the remote directory as an argument for the mount operation is done in a nontransparent manner; the location (that is, host name) of the remote directory has to be provided. However, from then on, users on machine *M1* can access files in the remote directory in a totally transparent manner.

To illustrate file mounting, we consider the file system depicted in Figure 17.4, where the triangles represent subtrees of directories that are of interest. In this figure, three independent file systems of machines named *U*, *S1*, and *S2* are shown. At this point, at each machine, only the local files can be accessed. In Figure 17.5(a), the effects of the mounting of *S1:/usr/shared* over *U:/usr/local* are shown. This figure depicts the view users

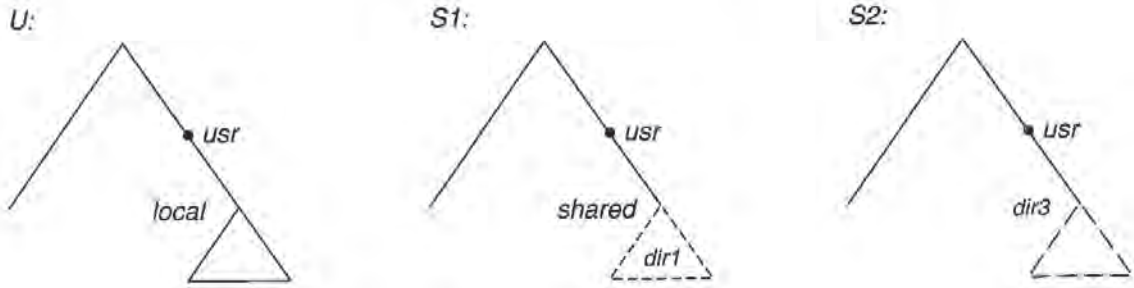


Figure 17.4 Three independent file systems.

on *U* have of their file system. Observe that they can access any file within the *dir1* directory, for instance, using the prefix */usr/local/dir1* on *U* after the mount is complete. The original directory */usr/local* on that machine is no longer visible.

Subject to access-rights accreditation, potentially any file system, or a directory within a file system, can be mounted remotely on top of any local directory. Diskless workstations can even mount their own roots from servers.

Cascading mounts are also permitted. That is, a file system can be mounted over another file system that is not a local one, but rather is a remotely mounted one. However, a machine is affected by only those mounts that it has itself invoked.

By mounting a remote file system, the client does not gain access to other file systems that were, by chance, mounted over the former file system. Thus, the mount mechanism does not exhibit a transitivity property. In Figure 17.5(b), we illustrate cascading mounts by continuing with our previous example. The figure shows the result of mounting *S2:/dir2/dir3* over *U:/usr/local/dir1*, which is already remotely mounted from *S1*. Users can access files within *dir3* on *U* using the prefix */usr/local/dir1*. If a shared file system is mounted over a user's home directories on all machines in a network, a user can log in to any workstation and get his home environment. This property is referred to as *user mobility*.

One of the design goals of NFS was to operate in a heterogeneous environment of different machines, operating systems, and network architectures. The NFS specification is independent of these media and thus encourages other implementations. This independence is achieved through the use of RPC primitives built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces. Hence, if the system consists of heterogeneous machines and file systems that are properly interfaced to NFS, file systems of different types can be mounted both locally and remotely.

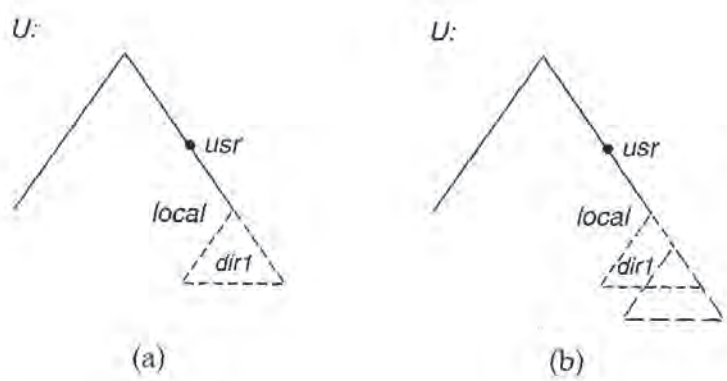


Figure 17.5 Mounting in NFS. (a) Mounts. (b) Cascading mounts.

The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services. Accordingly, two separate protocols are specified for these services; a *mount protocol*, and a protocol for remote file accesses called the *NFS protocol*. The protocols are specified as sets of RPCs. These RPCs are the building blocks used to implement transparent remote file access.

17.6.2.2 The Mount Protocol

The mount protocol is used to establish the initial logical connection between a server and a client. In Sun's implementation, each machine has a server process, outside the kernel, performing the protocol functions.

A mount operation includes the name of the remote directory to be mounted and the name of the server machine storing it. The mount request is mapped to the corresponding RPC and is forwarded to the mount server running on the specific server machine. The server maintains an *export list* (the */etc/exports* in UNIX, which can be edited by only a superuser), which specifies local file systems that it exports for mounting, along with names of machines that are permitted to mount them. Unfortunately, this list has a maximum length, so NFS is limited in scalability. Recall that any directory within an exported file system can be mounted remotely by an accredited machine. Hence, a component unit is such a directory. When the server receives a mount request that conforms to its export list, it returns to the client a *file handle* that serves as the key for further accesses to files within the mounted file system. The file handle contains all the information that the server needs to distinguish an individual file it stores. In UNIX terms, the file handle consists of a file-system identifier, and an inode number to identify the exact mounted directory within the exported file system.

The server also maintains a list of the client machines and the corresponding currently mounted directories. This list is used mainly for administrative purposes — for instance, for notifying all clients that the server is going down. Adding and deleting an entry in this list is the only way that the server state can be affected by the mount protocol.

Usually, a system has a static mounting preconfiguration that is established at boot time (*/etc/fstab* in UNIX); however, this layout can be modified. Besides the actual mount procedure, the mount protocol includes several other procedures, such as unmount and return export list.

17.6.2.3 The NFS Protocol

The NFS protocol provides a set of RPCs for remote file operations. The procedures support the following operations:

- Searching for a file within a directory

- Reading a set of directory entries
- Manipulating links and directories
- Accessing file attributes
- Reading and writing files

These procedures can be invoked only after a file handle for the remotely mounted directory has been established.

The omission of open and close operations is intentional. A prominent feature of NFS servers is that they are *stateless*. Servers do not maintain information about their clients from one access to another access. There are no parallels to UNIX's open-files table or file structures on the server side. Consequently, each request has to provide a full set of arguments, including a unique file identifier and an absolute offset inside the file for the appropriate operations. The resulting design is robust; no special measures need to be taken to recover a server after a crash. File operations need to be idempotent for this purpose.

Maintaining the list of clients mentioned above seems to violate the statelessness of the server. However, it is not essential for the correct operation of the client or the server, and hence this list does not need to be restored after a server crash. Consequently, it might include inconsistent data and is treated only as a hint.

A further implication of the stateless-server philosophy and a result of the synchrony of an RPC is that modified data (including indirection and status blocks) must be committed to the server's disk before results are returned to the client. That is, a client can cache write blocks, but when it flushes them to the server, it assumes that they have reached the server's disks. Thus, a server crash and recover will be invisible to a client; all blocks that the server is managing for the client will be intact. The consequent performance penalty can be large, because the advantages of caching are lost. In fact, there are several products now on the market that specifically address this NFS problem by providing fast stable storage (usually memory with battery backup) in which to store blocks written by NFS. These blocks remain intact even after system crash, and are written from this stable storage to disk periodically.

A single NFS write procedure call is guaranteed to be atomic, and also is not intermixed with other write calls to the same file. The NFS protocol, however, does not provide concurrency-control mechanisms. A **write** system call may be broken down into several RPC writes, because each NFS write or read call can contain up to 8K of data and UDP packets are limited to 1500 bytes. As a result, two users writing to the same remote file may get their data intermixed. The claim is that, because lock management is inherently stateful, a service outside the NFS should provide locking (and Solaris does). Users are advised to coordinate access to shared files using mechanisms outside the scope of NFS.

17.6.2.4 The NFS Architecture

The NFS architecture consists of three major layers; it is depicted schematically in Figure 17.6. The first layer is the UNIX file-system interface, based on the **open, read, write,** and **close** calls, and file descriptors.

The second layer is called *Virtual File System (VFS)* layer; it serves two important functions:

- It separates file-system generic operations from their implementation by defining a clean VFS interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems mounted locally.
- The VFS is based on a file representation structure called a *vnode* that contains a numerical designator for a file that is networkwide unique. (Recall that UNIX inodes are unique within only a single file system.) The kernel maintains one vnode structure for each active node (file or directory).

Thus, the VFS distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.

Similarly to standard UNIX, the kernel maintains a table (*/etc/mtab* in UNIX) recording the details of the mounts in which it took part as a client. Further, the vnodes for each directory that is mounted over are kept in

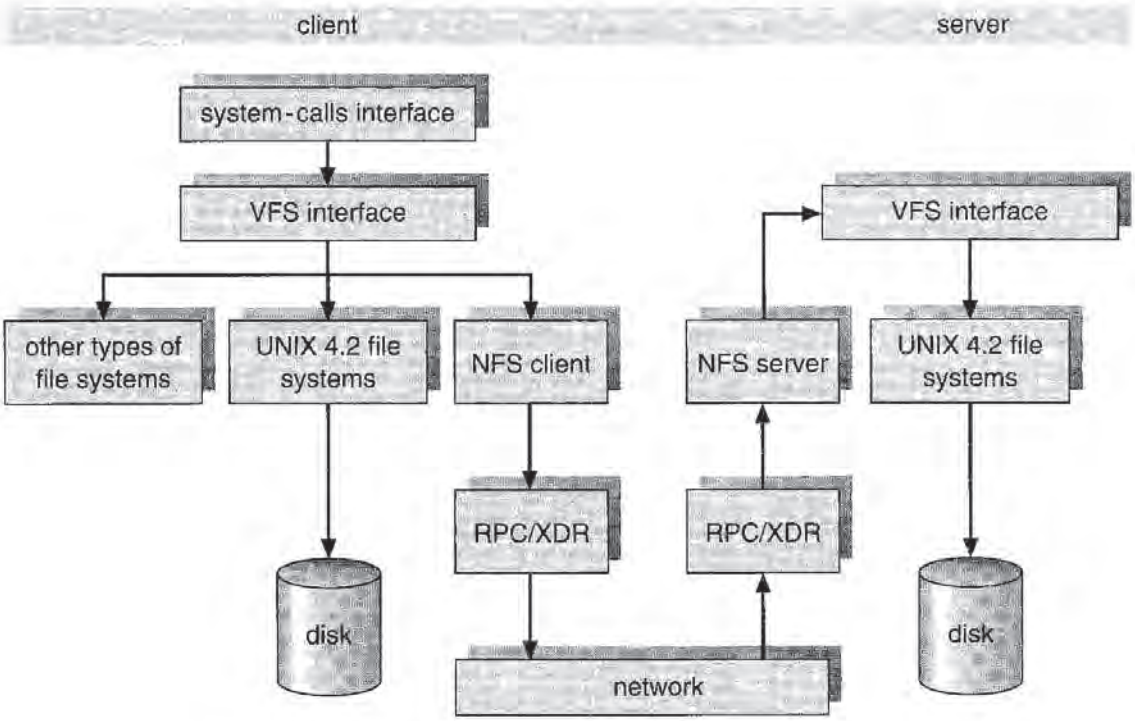


Figure 17.6 Schematic view of the NFS architecture.

memory at all times and are marked, so that requests concerning such directories will be redirected to the corresponding mounted file systems via the mount table. Essentially, the vnode structures, complemented by the mount table, provide a pointer for every file to its parent file system, as well as to the file system over which it is mounted.

The VFS activates file-system–specific operations to handle local requests according to their file-system types, and calls the NFS protocol procedures for remote requests. File handles are constructed from the relevant vnodes and are passed as arguments to these procedures. The layer implementing the NFS protocol is the bottom layer of the architecture and is called the NFS service layer.

As an illustration of the architecture, let us trace how an operation on an already-open remote file is handled (follow the example on Figure 17.6). The client initiates the operation by a regular system call. The operating-system layer maps this call to a VFS operation on the appropriate vnode. The VFS layer identifies the file as a remote one and invokes the appropriate NFS procedure. An RPC call is made to the NFS service layer at the remote server. This call is reinjected to the VFS layer on the remote system, which finds that it is local and invokes the appropriate file-system operation. This path is retraced to return the result. An advantage of this architecture is that the client and the server are identical; thus, a machine may be a client, or a server, or both.

The actual service on each server is performed by several kernel processes that provide a temporary substitute to a lightweight process (threads) facility.

17.6.2.5 Path-Name Translation

We do path-name translation by breaking the path into component names and performing a separate NFS *lookup* call for every pair of component name and directory vnode. Once a mount point is crossed, every component lookup causes a separate RPC to the server (see Figure 17.7). This expensive path-name–traversal scheme is needed, since each client has a unique layout of its logical name space, dictated by the mounts it performed. It would have been much more efficient to hand a server a path name and to receive a target vnode once a mount point was encountered. At any point, however, there can be another mount point for the particular client of which the stateless server is unaware.

So that lookup is faster, a directory name lookup cache on the client's side holds the vnodes for remote directory names. This cache speeds up references to files with the same initial path name. The directory cache is discarded when attributes returned from the server do not match the attributes of the cached vnode.

Recall that mounting a remote file system on top of another already-mounted remote file system (cascading mount) is allowed in NFS. However,

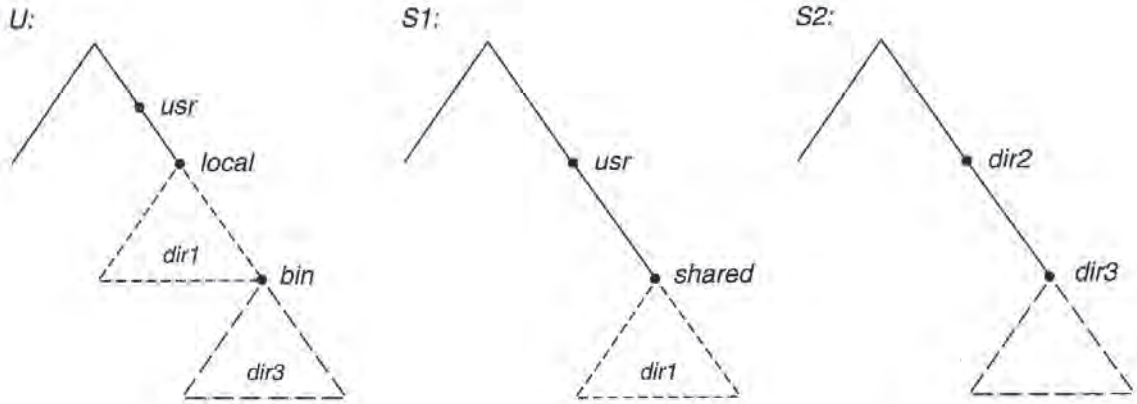


Figure 17.7 Path-name translation.

a server cannot act as an intermediary between a client and another server. Instead, a client must establish a direct server–client connection with the second server by directly mounting the desired directory. When a client has a cascading mount, more than one server can be involved in a path-name traversal. However, each component lookup is performed between the original client and some server. Therefore, when a client does a lookup on a directory on which the server has mounted a file system, the client sees the underlying directory, instead of the mounted directory.

17.6.2.6 Remote Operations

With the exception of opening and closing files, there is almost one-to-one correspondence between the regular UNIX system calls for file operations and the NFS protocol RPCs. Thus, a remote file operation can be translated directly to the corresponding RPC. Conceptually, NFS adheres to the remote-service paradigm, but in practice buffering and caching techniques are employed for the sake of performance. There is no direct correspondence between a remote operation and an RPC. Instead, file blocks and file attributes are fetched by the RPCs and are cached locally. Future remote operations use the cached data, subject to consistency constraints.

There are two caches: file-blocks cache and file-attribute (inode-information) cache. On a file open, the kernel checks with the remote server whether to fetch or revalidate the cached attributes. The cached file blocks are used only if the corresponding cached attributes are up to date. The attribute cache is updated whenever new attributes arrive from the server. Cached attributes are (by default) discarded after 60 seconds. Both read-ahead and delayed-write techniques are used between the server and the client. Clients do not free delayed-write blocks until the server confirms that the data have been written to disk. In contrast to the system

used in Sprite, delayed-write is retained even when a file is opened concurrently, in conflicting modes. Hence, UNIX semantics are not preserved.

Tuning the system for performance makes it difficult to characterize the consistency semantics of NFS. New files created on a machine may not be visible elsewhere for 30 seconds. It is indeterminate whether writes to a file at one site are visible to other sites that have this file open for reading. New opens of that file observe only the changes that have already been flushed to the server. Thus, NFS provides neither strict emulation of UNIX semantics, nor the session semantics of Andrew. In spite of these drawbacks, the utility and high performance of the mechanism makes it the most widely used, multivendor distributed system in operation.

17.6.3 Andrew

Andrew is a distributed computing environment that has been under development since 1983 at Carnegie Mellon University. As such, it is one of the newest DFSs. The Andrew file system (AFS) constitutes the underlying information-sharing mechanism among clients of the environment. A commercial implementation of AFS, known as DFS, is part of the DCE distributed computing environment from the OSF organization. Many UNIX vendors, as well as Microsoft, have announced support for this system. One of the most formidable attributes of Andrew is scalability: The Andrew system is targeted to span over 5000 workstations.

17.6.3.1 Overview

Andrew distinguishes between *client machines* (sometimes referred to as workstations) and dedicated *server machines*. Servers and clients alike run the 4.2BSD UNIX operating system and are interconnected by an internet of LANs.

Clients are presented with a partitioned space of file names: a *local name space* and a *shared name space*. Dedicated servers, collectively called *Vice*, after the name of the software they run, present the shared name space to the clients as a homogeneous, identical, and location transparent file hierarchy. The local name space is the root file system of a workstation, from which the shared name space descends. Workstations run the *Virtue* protocol to communicate with Vice, and are required to have local disks where they store their local name space. Servers collectively are responsible for the storage and management of the shared name space. The local name space is small, is distinct for each workstation, and contains system programs essential for autonomous operation and better performance. Also local are temporary files and files that the workstation owner, for privacy reasons, explicitly wants to store locally.

Viewed at a finer granularity, clients and servers are structured in clusters interconnected by a WAN. Each cluster consists of a collection of workstations on a LAN, and a representative of Vice called a *cluster server*, and is connected to the WAN by a *router*. The decomposition into clusters is done primarily to address the problem of scale. For optimal performance, workstations should use the server on their own cluster most of the time, thereby making cross-cluster file references relatively infrequent.

The file-system architecture was based on consideration of scale, too. The basic heuristic was to offload work from the servers to the clients, in light of experience indicating that server CPU speed is the system's bottleneck. Following this heuristic, the key mechanism selected for remote file operations is to cache files in large chunks (64K). This feature reduces file-open latency, and allows reads and writes to be directed to the cached copy without frequently involving the servers.

There are additional issues in Andrew's design that we shall not discuss here; briefly, they are these:

- **Client mobility:** Clients are able to access any file in the shared name space from any workstation. The only effect clients may notice when accessing files from other than their usual workstations is some initial performance degradation due to the caching of files.
- **Security:** The Vice interface is considered as the boundary of trustworthiness, because no client programs are executed on Vice machines. Authentication and secure-transmission functions are provided as part of a connection-based communication package, based on the RPC paradigm. After mutual authentication, a Vice server and a client communicate via encrypted messages. Encryption is performed by hardware devices or (more slowly) in software. Information about clients and groups is stored in a protection database that is replicated at each server.
- **Protection:** Andrew provides *access lists* for protecting directories and the regular UNIX bits for file protection. The access list may contain information about those users that are allowed to access a directory, as well as information about those users that are *not* allowed to access a directory. Thus, with this scheme, it is simple to specify that everyone except, say, Jim can access a directory. Andrew supports the access types read, write, lookup, insert, administer, lock, and delete.
- **Heterogeneity:** Defining a clear interface to Vice is a key for integration of diverse workstation hardware and operating system. So that heterogeneity is facilitated, some files in the local */bin* directory are symbolic links pointing to machine-specific executable files residing in Vice.

17.6.3.2 The Shared Name Space

Andrew's shared name space is constituted of component units called *volumes*. Andrew's volumes are unusually small component units. Typically, they are associated with the files of a single client. Few volumes reside within a single disk partition, and they may grow (up to a quota) and shrink in size. Conceptually, volumes are glued together by a mechanism similar to the UNIX mount mechanism. However, the granularity difference is significant, since in UNIX only an entire disk partition (containing a file system) can be mounted. Volumes are a key administrative unit and play a vital role in identifying and locating an individual file.

A Vice file or directory is identified by a low-level identifier called a *fid*. Each Andrew directory entry maps a path-name component to a fid. A fid is 96 bits long and has three equal-length components: a *volume number*, a *vnode number*, and a *uniquifier*. The vnode number is used as an index into an array containing the inodes of files in a single volume. The uniquifier allows reuse of vnode numbers, thereby keeping certain data structures compact. Fids are location transparent; therefore, file movements from server to server do not invalidate cached directory contents.

Location information is kept on a volume basis in a *volume-location database* replicated on each server. A client can identify the location of every volume in the system by querying this database. It is the aggregation of files into volumes that makes it possible to keep the location database at a manageable size.

To balance the available disk space and utilization of servers, volumes need to be migrated among disk partitions and servers. When a volume is shipped to its new location, its original server is left with temporary forwarding information, so that the location database does not need to be updated synchronously. While the volume is being transferred, the original server still can handle updates, which are shipped later to the new server. At some point, the volume is briefly disabled so that the recent modifications can be processed; then, the new volume becomes available again at the new site. The volume-movement operation is atomic; if either server crashes, the operation is aborted.

Read-only replication at the granularity of an entire volume is supported for system-executable files and for seldom-updated files in the upper levels of the Vice name space. The volume-location database specifies the server containing the only read–write copy of a volume and a list of read-only replication sites.

17.6.3.3 File Operations and Consistency Semantics

The fundamental architectural principle in Andrew is the caching of entire files from servers. Accordingly, a client workstation interacts with Vice servers only during opening and closing of files, and even this interaction

is not always necessary. No remote interaction is caused by reading or writing files (in contrast to the remote-service method). This key distinction has far-reaching ramifications for performance, as well as for semantics of file operations.

The operating system on each workstation intercepts file-system calls and forwards them to a client-level process on that workstation. This process, called *Venus*, caches files from Vice when they are opened, and stores modified copies of files back on the servers from which they came when they are closed. Venus may contact Vice only when a file is opened or closed; reading and writing of individual bytes of a file are performed directly on the cached copy and bypass Venus. As a result, writes at some sites are not visible immediately at other sites.

Caching is further exploited for future opens of the cached file. Venus assumes that cached entries (files or directories) are valid unless notified otherwise. Therefore, Venus does not need to contact Vice on a file open to validate the cached copy. The mechanism to support this policy, called *callback*, dramatically reduces the number of cache-validation requests received by servers. It works as follows. When a client caches a file or a directory, the server updates its state information recording this caching. We say that the client has a callback on that file. The server notifies the client before allowing a modification to the file by another client. In such a case, we say that the server removes the callback on the file for the former client. A client can use a cached file for open purposes only when the file has a callback. If a client closes a file after modifying it, all other clients caching this file lose their callbacks. Therefore, when these clients open the file later, they have to get the new version from the server.

Reading and writing bytes of a file are done directly by the kernel without Venus intervention on the cached copy. Venus regains control when the file is closed and, if the file has been modified locally, Venus updates the file on the appropriate server. Thus, the only occasions in which Venus contacts Vice servers are on opens of files that either are not in the cache or have had their callback revoked, and on closes of locally modified files.

Basically, Andrew implements session semantics. The only exceptions are file operations other than the primitive read and write (such as protection changes at the directory level), which are visible everywhere on the network immediately after the operation completes.

In spite of the callback mechanism, a small amount of cached validation traffic is still present, usually to replace callbacks lost because of machine or network failures. When a workstation is rebooted, Venus considers all cached files and directories suspect, and generates a cache-validation request for the first use of each such entry.

The callback mechanism forces each server to maintain callback information and each client to maintain validity information. If the amount of callback information maintained by a server is excessive, the server can

break callbacks and reclaim some storage by unilaterally notifying clients and revoking the validity of their cached files. There is a potential for inconsistency if the callback state maintained by Venus gets out of sync with the corresponding state maintained by the servers.

Venus also caches contents of directories and symbolic links, for path-name translation. Each component in the path name is fetched, and a callback is established for it if it is not already cached, or if the client does not have a callback on it. Lookups are done locally by Venus on the fetched directories using fids. There is no forwarding of requests from one server to another. At the end of a path-name traversal, all the intermediate directories and the target file are in the cache with callbacks on them. Future open calls to this file will involve no network communication at all, unless a callback is broken on a component of the path name.

The only exceptions to the caching policy are modifications to directories that are made directly on the server responsible for that directory for reasons of integrity. There are well-defined operations in the Vice interface for such purposes. Venus reflects the changes in its cached copy to avoid refetching the directory.

17.6.3.4 Implementation

Client processes are interfaced to a UNIX kernel with the usual set of system calls. The kernel is modified slightly to detect references to Vice files in the relevant operations and to forward the requests to the client-level Venus process at the workstation.

Venus carries out path-name translation component by component, as was described above. It has a mapping cache that associates volumes to server locations in order to avoid server interrogation for an already-known volume location. If a volume is not present in this cache, Venus contacts any server to which it already has a connection, requests the location information, and enters that information into the mapping cache. Unless Venus already has a connection to the server, it establishes a new connection. It then uses this connection to fetch the file or directory. Connection establishment is needed for authentication and security purposes. When a target file is found and cached, a copy is created on the local disk. Venus then returns to the kernel, which opens the cached copy and returns its handle to the client process.

The UNIX file system is used as a low-level storage system for both servers and clients. The client cache is a local directory on the workstation's disk. Within this directory are files whose names are placeholders for cache entries. Both Venus and server processes access UNIX files directly by the latter's inodes to avoid the expensive path-name–to-inode translation routine (*namei*). Because the internal inode interface is not visible to client-level processes (both Venus and server

processes are client-level processes), an appropriate set of additional system calls was added.

Venus manages two separate caches: one for status and the other for data. It uses a simple least recently used (LRU) algorithm to keep each of them bounded in size. When a file is flushed from the cache, Venus notifies the appropriate server to remove the callback for this file. The status cache is kept in virtual memory to allow rapid servicing of **stat** (file status returning) system calls. The data cache is resident on the local disk, but the UNIX I/O buffering mechanism does some caching of disk blocks in memory that is transparent to Venus.

A single client-level process on each file server services all file requests from clients. This process uses a lightweight-process package with nonpreemptable scheduling to service many client requests concurrently. The RPC package is integrated with the lightweight-process package, thereby allowing the file server to be concurrently making or servicing one RPC per lightweight process. RPC is built on top of a low-level datagram abstraction. Whole file transfer is implemented as a side effect of these RPC calls. There is one RPC connection per client, but there is no a priori binding of lightweight processes to these connections. Instead, a pool of lightweight processes services client requests on all connections. The use of a single multithreaded server process allows the caching of data structures needed to service requests. On the other hand, a crash of a single server process has the disastrous effect of paralyzing this particular server.

17.6.4 Sprite

Sprite is an experimental distributed operating system under development at the University of California at Berkeley. Its main research thrusts have been in the areas of network file systems, process migration, and high-performance file systems. Sprite runs on Sun and DEC workstations and is used for day-to-day computing by a few dozen students, faculty, and staff.

17.6.4.1 Overview

Sprite designers envision the next generation of workstations as powerful machines with vast physical memory. The configuration for which Sprite is targeted is large and fast disks concentrated on a few server machines servicing the storage needs of hundreds of diskless workstations. The workstations are interconnected by several LANs. Because file caching is used, the large physical memories will compensate for the lack of local disks.

The interface that Sprite provides in general, and to the file system in particular, is much like the one provided by UNIX. The file system appears as a single UNIX tree encompassing all files and devices in the network,

making them equally and transparently accessible from every workstation. The location transparency in Sprite is complete; there is no way to discern a file's network location from that file's name.

Unlike NFS, Sprite enforces consistency of shared files. Each **read** system call is guaranteed to return the most up-to-date data for a file, even if it is being opened concurrently by several remote processes. Thus, Sprite emulates a single time-sharing UNIX system in a distributed environment.

A unique feature of the Sprite file system is its interplay with the virtual-memory system. Most versions of UNIX use a special disk partition as a swapping area for virtual-memory purposes. In contrast, Sprite uses ordinary files, called *backing files*, to store the data and stacks of running processes. The basis for this design is that it simplifies process migration and enables flexibility and sharing of the space allocated for swapping. Backing files are cached in the main memories of servers, just like any other file. The designers claim that clients should be able to read random pages from server's (physical) cache faster than they can from local disks, which means that a server with a large cache may provide better paging performance than will local disk.

The virtual memory and file system share the same cache and negotiate on how to divide it according to their conflicting needs. Sprite allows the file cache on each machine to grow and shrink in response to changing demands of the machine's virtual memory and file system. This scheme is similar to Apollo's Domain operating system, which has a dynamically sized swap space.

We briefly mention a few other features of Sprite. In contrast to UNIX, where only code can be shared among processes, Sprite provides a mechanism for sharing an address space between client processes on a single workstation. A process-migration facility, which is transparent to clients as well as to the migrated process, is also provided.

17.6.4.2 Prefix Tables

Sprite presents its client with a single file-system hierarchy. The hierarchy is composed of several subtrees called *domains* (the Sprite term for component units), with each server providing storage for one or more domains. Each machine maintains a server map called a *prefix table*, whose function is to map domains to servers. The mapping is built and updated dynamically by a broadcast protocol that places a message on the network for all other network members to read. We first describe how the tables are used during name lookups, and later describe how the tables change dynamically.

Each entry in a prefix table corresponds to one of the domains. It contains the name of the topmost directory in the domain (called the prefix for the domain), the network address of the server storing the domain, and a numeric designator identifying the domain's root directory for the

storing server. Typically, this designator is an index into the server table of open files; it saves repeating expensive name translation.

Every lookup operation for an absolute path name starts with the client searching its prefix table for the longest prefix matching the given file name. The client strips the matching prefix from the file name and sends the remainder of the name to the selected server along with the designator from the prefix-table entry. The server uses this designator to locate the root directory of the domain, and then proceeds by usual UNIX path-name translation for the remainder of the file name. If the server succeeds in completing the translation, it replies with a designator for the open file.

There are several cases where the server does not complete the lookup operation:

- When the server encounters an absolute path name in a symbolic link, it immediately returns to the client the absolute path name. The client looks up the new name in its prefix table and initiates another lookup with a new server.
- A path name can ascend past the root of a domain (because of a parent ".." component). In such a case, the server returns the remainder of the path name to the client. The latter combines the remainder with the prefix of the domain that was just exited to form a new absolute path name.
- A path name can also descend into a new domain. This descent can happen when an entry for a domain is absent from the table, and as a result the prefix of the domain above the missing domain is the longest matching prefix. The selected server cannot complete the path-name traversal, since the latter descends outside its domain. Alternatively, when a root of a domain is beneath a working directory and a file in that domain is referred to with a relative path name, the server also cannot complete the translation. The solution to these situations is to place a marker to indicate domain boundaries (a mount point, in NFS terms). The marker is a special kind of file called a *remote link*. Similar to a symbolic link, its content is a file name — its own name in this case. When a server encounters a remote link, it returns the file name to the client.

Relative path names are treated much as they are in conventional UNIX. When a process specifies a new working directory, the prefix mechanism is used to open the working directory and both its server address and designator are saved in the process's state. When a lookup operation detects a relative path name, it sends the path name directly to the server for the current working directory, along with the directory's designator. Hence, from the server's point of view, there is no difference between relative and absolute name lookups.

So far, the key difference from mappings based on the UNIX mount mechanism was the initial step of matching the file name against the prefix table, instead of looking it up component by component. Systems (such as NFS and conventional UNIX) that employ a name-lookup cache create a similar effect of avoiding the component-by-component lookup once the cache holds the appropriate information.

Prefix tables are a unique mechanism mainly because of the way they evolve and change. When a remote link is encountered by the server, this indicates to the server that the client lacks an entry for a domain — the domain whose remote link was encountered. To obtain the missing prefix information, the client must broadcast a file name. A *broadcast* is a network message that is seen by all the computer systems on the network. The server storing that file responds with the prefix-table entry for this file, including the string to use as a prefix, the server's address, and the descriptor corresponding to the domain's root. The client then can fill in the details in its prefix table.

Initially, each client starts with an empty prefix table. The broadcast protocol is invoked to find the entry for the root domain. More entries are added gradually as needed; a domain that has never been accessed will not appear in the table.

The server locations kept in the prefix table are hints that are corrected when found to be wrong. Hence, if a client tries to open a file and gets no response from the server, it invalidates the prefix-table entry and attempts a broadcast query. If the server has become available again, it responds to the broadcast and the prefix-table entry is reestablished. This same mechanism also works if the server reboots at a different network address, or if its domains are moved to other servers.

The prefix mechanism ensures that, whenever a server storing a domain is functioning, the domain's files can be opened and accessed from any machine regardless of the status of the servers of domains above the particular domain. Essentially, the built-in broadcast protocol enables dynamic configuration and a certain degree of robustness. Also, when a prefix for a domain exists in a client's table, a direct client–server connection is established as soon as the client attempts to open a file in that domain (in contrast to path-name traversal schemes).

A machine with a local disk that wishes to keep private some local files can place an entry for the private domain in its prefix table and refuse to respond to broadcast queries about that domain. One of the uses of this provision can be for the directory */usr/tmp*, which holds temporary files generated by many UNIX programs. Every workstation needs access to */usr/tmp*. But workstations with local disks would probably prefer to use their own disk for the temporary space. Recall that the designers of Sprite expect reads from a server cache to be faster than those from a local disk, but do not predict this relationship for writes. They can set up their */usr/tmp* domains for private use, with a network file server providing a

public version of the domain for diskless clients. All broadcast queries for */usr/tmp* would be handled by the public server.

A primitive form of read-only replication can also be provided. It can be arranged that servers storing a replicated domain give different clients different prefix entries (standing for different replicas) for the same domain. The same technique can be used for sharing binary files by different hardware types of machines.

Since the prefix tables bypass part of the directory-lookup mechanism, the permission checking done during lookup is bypassed too. The effect is that all programs implicitly have search permission along all the paths denoting prefixes of domains. If access to a domain is to be restricted, it must be restricted at or below the root of the domain.

17.6.4.3 Caching and Consistency

An important aspect of the Sprite file-system design is the extent of the use of caching techniques. Capitalizing on the large main memories and advocating diskless workstations, file caches are stored in memory, instead of on local disks (as in Andrew). Caching is used by both client and server workstations. The caches are organized on a block basis, rather than on a file basis (as in Andrew). The size of the blocks is currently 4K. Each block in the cache is virtually addressed by the file designator and a block location within the file. Using virtual addresses instead of physical disk addresses enables clients to create new blocks in the cache and to locate any block without the file inode being brought from the server.

When a read kernel call is invoked to read a block of a file, the kernel first checks its cache and returns the information from the cache, if it is present. If the block is not in the cache, the kernel reads it from disk (if the file is locally stored), or requests it from the server; in either case, the block is added to the cache, replacing the least recently used block. If the block is requested from the server, the server checks its own cache before issuing a disk I/O request, and adds the block to its cache, if the block was not already there. Currently, Sprite does not use read-ahead to speed up sequential read (in contrast to NFS).

A delayed-write approach is used to handle file modification. When an application issues a write kernel call, the kernel simply writes the block into its cache and returns to the application. The block is not written through to the server's cache or the disk until it is ejected from the cache, or 30 seconds have elapsed since the block was last modified. Hence, a block written on a client machine will be written to the server's cache in at most 30 seconds, and will be written to the server's disk after an additional 30 seconds. This policy results in better performance in exchange for the possibility of recent changes being lost in a crash.

Sprite employs a version-number scheme to enforce consistency of shared files. The version number of a file is incremented whenever a file

is opened in write mode. When a client opens a file, it obtains from the server the file's current version number, which the client compares to the version number associated with the cached blocks for that file. If they are different, the client discards all cached blocks for the file and reloads its cache from the server when the blocks are needed. Because of the delayed-write policy, the server does not always have the current file data. Servers handle this situation by keeping track of the last writer for each file. When a client other than the last writer opens the file, the server forces the last writer to write back all its modified data blocks to the server's cache.

When a server detects (during an open operation) that a file is open on two or more workstations and that at least one of them is writing the file, it disables client caching for that file. All subsequent reads and writes go through the server, which serializes the accesses. Caching is disabled on a file basis, resulting in only clients with open files being affected. Obviously, a substantial degradation of performance occurs when caching is disabled. A noncachable file becomes cachable again when it has been closed by all clients. A file may be cached simultaneously by several active readers.

This approach depends on the server being notified whenever a file is opened or closed. This notification requirement prohibits performance optimizations such as name caching in which clients open files without contacting the file servers. Essentially, the servers are used as centralized control points for cache consistency. To fulfill this function, they must maintain state information about open files.

17.6.5 Locus

Locus is a project at the University of California at Los Angeles to build a full-scale distributed operating system. The system is upward-compatible with UNIX, but unlike those in NFS, UNIX United, and other UNIX-based distributed systems, the extensions are major and necessitate an entirely new kernel, rather than a modified one.

17.6.5.1 Overview

The Locus file system presents to clients and applications a single tree-structure naming hierarchy. This structure covers all objects (files, directories, executable files, and devices) of all the machines in the system. Locus names are fully transparent; it is not possible to discern from a name of an object the object's location in the network. To a first approximation, there is almost no way to distinguish the Locus name structure from a standard UNIX tree.

A Locus file may correspond to a set of copies distributed on different sites. An additional transparency dimension is introduced since it is the

system's responsibility to keep all copies up to date and to ensure that access requests are served by the most recent available version. Clients may have control over both the number and location of replicated files. Conversely, clients may prefer to be totally unaware of the replication scheme. In Locus, file replication serves mainly to increase availability for reading purposes in the event of failures and partitions. A primary-copy approach is adopted for modifications.

Locus adheres to the same file-access semantics with which standard UNIX presents clients. Locus provides these semantics in the distributed and replicated environment in which it operates. Alternate mechanisms of advisory and enforced locking of files and parts of files are also offered. Moreover, atomic updates of files are supported by **commit** and **abort** system calls.

Operation during failures and network partitions is emphasized in Locus' design. As long as a copy of a file is available, read requests can be served, and it is still guaranteed that the version read is the most recent available one. Automatic mechanisms update stale copies of files at the time of the merge of the latter's storage site to a partition.

Emphasis on high performance in the design of Locus led to the incorporation of networking functions (such as formatting, queuing, transmitting, and retransmitting messages) into the operating system. Specialized remote-operations protocols were devised for kernel-to-kernel communication, in contrast to the prevalent approach of using the RPC protocol, or some other existing protocol. The reduction of the number of network layers has achieved high performance for remote operations. On the other hand, this specialized protocol hampers the portability of Locus to different networks and file systems.

An efficient but limited process facility called *server processes* (lightweight processes) was created for serving remote requests. These processes have no nonprivileged address space. All their code and stacks are resident in the operating-system nucleus; they can call internal system routines directly, and can share some data. These processes are assigned to serve network requests that accumulate in a system queue. The system is configured with some number of these processes, but that number is automatically and dynamically altered during system operation.

17.6.5.2 The Name Structure

The logical name structure disguises both location and replication details from clients and applications. In effect, logical filegroups are joined together to form this unified structure. Physically, a logical filegroup is mapped to multiple *physical containers* (called also *packs*) that reside at various sites and that store replicas of the files of that filegroup. The pair <logical-filegroup-number, inode number>, which is referred to as a file's

designator, serves as a globally unique low-level name for a file. Observe that the designator itself hides both location and replication details.

Each site has a consistent and complete view of the logical name structure. A logical mount table is replicated globally and contains an entry for each logical filegroup. An entry records the file designator of the directory over which the filegroup is logically mounted, and indication of which site is currently responsible for access synchronization within the filegroup. We shall explain the function of this site later in this section. In addition, each site that stores a copy of the directory over which a subtree is mounted must keep that directory's inode in memory with an indication that it is mounted over. Keeping the inode in memory is done so that any access from any site to that directory will be caught, allowing the standard UNIX mount indirection to function (via the logical mount table, Section 11.1.2). A protocol, implemented within the **mount** and **unmount** Locus system calls, performs update of the logical mount tables on all sites, when necessary.

On the physical level, physical containers correspond to disk partitions and are assigned pack numbers that, together with a logical filegroup number, identify an individual pack. One of the packs is designated as the *primary copy*. A file must be stored at the site of the primary copy, and in addition can be stored at any subset of the other sites where there exists a pack corresponding to its filegroup. Thus, the primary copy stores the filegroup completely, whereas the rest of the packs might be partial.

Replication is especially useful for directories in the high levels of the name hierarchy. Such directories exhibit mostly read-only characteristics and are crucial for path-name translation of most files.

The various copies of a file are assigned the same inode number on all the filegroup's packs. Consequently, a pack has an empty inode slot for all files that it does not store. Data-page numbers may be different on different packs; hence, reference over the network to data pages use logical page numbers rather than physical ones. Each pack has a mapping of these logical numbers to its physical numbers. So that automatic replication management will be facilitated, each inode of a file copy contains a version number, determining which copy dominates other copies.

Each site has a container table, which maps logical filegroup numbers to disk locations for the filegroups that have packs locally on this site. When requests for accesses to files stored locally arrive at a site, the system consults this table to map the file designator to a local disk address.

Although globally unique file naming is important most of the time, there are certain files and directories that are hardware and site specific (that is, */bin*, which is hardware specific, and */dev*, which is site specific). Locus provides transparent means for translating references to these traditional file names to hardware- and site-specific files.

17.6.5.3 File Operations

Locus' approach to file operations is certainly a departure from the prevalent client–server model. Providing replicated files with synchronous access necessitates an additional function. Locus distinguishes three logical roles in file accesses, each one potentially performed by a different site:

- **Using site (US):** The US issues the requests to open and access a remote file.
- **Storage site (SS):** The SS is the site selected to serve the requests.
- **Current synchronization site (CSS):** The CSS enforces global synchronization policy for a filegroup, and selects an SS for each open request referring to a file in the filegroup. There is at most one CSS for each filegroup in any set of communicating sites (that is, a partition). The CSS maintains the version number and a list of physical containers for every file in the filegroup.

We now describe the open, read, write, close, commit, and abort operations, as they are carried out by the US, SS, and CSS entities. Related synchronization issues are described separately in the next subsection.

Opening a file commences as follows. The US determines the relevant CSS by looking up the filegroup in the logical mount table, and then forwards the open request to the CSS. The CSS polls potential SSs for that file to decide which one of them will act as the real SS. In its polling messages, the CSS includes the version number for the particular file, so that the potential SS can, by comparing this number to their own, decide whether or not their copy is up to date. The CSS selects an SS by considering the responses it received from the candidate sites, and sends the selected SS identity to the US. Both the CSS and the SS allocate in-core inode structures for the opened file. The CSS needs this information to make future synchronization decisions, and the SS maintains the inode to serve forthcoming accesses efficiently.

After a file is open, read requests are sent directly to the SS without the CSS intervention. A read request contains the designator of the file, the logical number of the needed page within that file, and a guess as to where the in-core inode is stored in the SS. Once the inode is found, the SS translates the logical page number to a physical number, and a standard low-level routine is called to allocate a buffer and to obtain the appropriate page from disk. The buffer is queued on the network queue for transmission as a response to the US, where it is stored in a kernel buffer. Once a page is fetched to the US, further **read** calls are serviced from the kernel buffer. As in the case of local disk reads, read-ahead is useful to speed up sequential reads, at both the US and the SS.

If a process loses its connection with a file that it is reading remotely, the system attempts to reopen a different copy of the same version of the file.

Translating a path name into a file designator proceeds by a seemingly conventional path-name traversal mechanism, since path names are regular UNIX path names, with no exceptions (unlike UNIX United). Every lookup of a component of the path name within a directory involves opening the directory and reading from it. Observe that there is no parallel to NFS's remote lookup operation, and that the actual directory searching is performed by the client, rather than by the server.

A directory opened for path-name searching is open not for normal read, but instead for an internal unsynchronized read. The distinction is that no global synchronization is needed, and no locking is done while the reading is performed; that is, updates to the directory can occur while the search is ongoing. When the directory is local, the CSS is not even informed of such access.

In Locus, a primary-copy policy is employed for file modification. The CSS has to select the primary-copy pack site as the SS for an open for a write. The act of modifying data takes on two forms. If the modification does not include the entire page, the old page is read from the SS using the read protocol. If the change involves the entire page, a buffer is set up at the US without any reads. In either case, after changes are made, possibly by delayed-write, the page is sent back to the SS. All modified pages must be flushed to the SS before a modified file can be closed.

If a file is closed by the last client process at a US, the SS and CSS must be informed so that they can deallocate in-core inode structures, and so that the CSS can alter state data that might affect its next synchronization decision.

Commit and abort system calls are provided, and closing a file commits it. If a file is open for modification by more than one process, the changes are not made permanent until one of the processes issues a commit system call or until all the processes close the file.

When a file is modified, shadow pages are allocated at the SS. The in-core copy of the disk inode is updated to point to these new shadow pages. The disk inode is kept intact, pointing to the original pages. The atomic commit operation consists of replacing the disk inode with the in-core inode. After that point, the file contains the new information. To abort a set of changes, we merely discard the in-core inode information and free up the disk space used to record the changes. The US function never deals with actual disk pages, but rather deals with logical pages. Thus, the entire shadow-page mechanism is implemented at the SS and is transparent to the US.

Locus deals with file modification by first committing the change to the primary copy. Later, messages are sent to all other SSs of the modified file,

as well as to the CSS. At minimum, these messages identify the modified file and contain the new version number (to prevent attempts to read the old versions). At this point, it is the responsibility of these additional SSs to bring their files up to date by propagating the entire file or just the changes. A queue of propagation requests is kept within the kernel at each site, and a kernel process services the queue efficiently by issuing appropriate read requests. This propagation procedure uses the standard commit mechanism. Thus, if contact with the file containing the newer version is lost, the local file is left with a coherent copy, albeit one still out of date.

Given this commit mechanism, we are always left with either the original file or a completely changed file, but never with a partially made change, even in the face of site failures.

17.6.5.4 Synchronized Accesses to Files

Locus tries to emulate conventional UNIX semantics on file accesses in a distributed environment. In standard UNIX, multiple processes are permitted to have the same file open concurrently. These processes issue **read** and **write** system calls, and the system guarantees that each successive operation sees the effects of the ones that precede it. We can implement this scheme fairly easily by having the processes share the same operating-system data structures and caches, and by using locks on data structures to serialize requests. Since remote tasking is supported in Locus, such situations can arise when the sharing processes do not co-reside on the same machine and hence complicate the implementation significantly.

There are two sharing modes to consider. First, in UNIX, several processes descending from the same ancestor process can share the same current position (offset) in a file. A single token scheme is devised to preserve this special mode of sharing. A site can proceed to execute system calls that need the offset only when the token is present.

Second, in UNIX, the same in-core inode for a file can be shared by several processes. In Locus, the situation is much more complicated, since the inode of the file can be cached at several sites. Also, data pages are cached at multiple sites. A multiple-data-tokens scheme is used to synchronize sharing of the file's inode and data. A single exclusive-writer, multiple-readers policy is enforced. Only a site with the write token for a file may modify the file, and any site with a read token can read the file. Both token schemes are coordinated by token managers operating at the corresponding storage sites.

The cached data pages are guaranteed to contain valid data only when the file's data token is present. When the write data token is taken from that site, the inode is copied back to the SS, as are all modified pages. Since arbitrary changes may have occurred to the file when the token was not

present, all cached buffers are invalidated when the token is released. When a data token is granted to a site, both the inode and data pages need to be fetched from the SS. There are some exceptions to this policy. Some attribute reading and writing calls (for example, **stat**), as well as directory reading and modifying (for example, **lookup**) calls are not subject to the synchronization constraints. These calls are sent directly to the SS, where the changes are made, committed, and propagated to all storage and using sites.

This mechanism guarantees consistency; each access sees the most recent data. A different issue regarding access synchronization is serializability of accesses. To this end, Locus offers facilities for locking entire files or parts of them. Locking can be advisory (checked only as a result of a locking attempt), or enforced (checked on all reads and writes). A process can choose either to fail if it cannot immediately get a lock, or to wait for the lock to be released.

17.6.5.5 Operation in a Faulty Environment

The basic approach in Locus is to maintain, within a single partition, strict synchronization among copies of a file, so that all clients of that file within that partition see the most recent version.

The primary-copy approach eliminates the possibility of conflicting updates, since the primary copy must be in the client's partition to allow an update. However, the problem of detecting updates and propagating them to all the copies remains, especially since updates are allowed in a partitioned network. During normal operation, the commit protocol ascertains proper detection and propagation of updates, as was described in the last subsection. However, a more elaborate scheme has to be employed by recovering sites that wish to bring their packs up to date. To this end, the system maintains a *commit count* for each filegroup, enumerating each commit of every file in the filegroup. Each pack has a *lower-water mark (LWM)* that is a commit-count value, up to which the system guarantees that all prior commits are reflected in the pack. Also, the primary copy pack keeps a complete list of all the recent commits in secondary storage. When a pack joins a partition, it contacts the primary copy site, and checks whether its LWM is within the recent commit-list bounds. If it is, the pack site schedules a kernel process, which brings the pack to a consistent state by performing the missing updates. If the primary pack is not available, writing is disallowed in this partition, but reading is possible after a new CSS is chosen. The new CSS communicates with the partition members so that it will be informed of the most recent available (in the partition) version of each file in the filegroup. Once the new CSS accomplishes this objective, other pack sites can reconcile

themselves with it. As a result, all communicating sites see the same view of the filegroup, and this view is as complete as possible, given a particular partition. Note that, since updates are allowed within the partition with the primary copy, and reads are allowed in the rest of the partitions, it is possible to read out-of-date replicas of a file. Thus, Locus sacrifices consistency for the ability to continue and both to update and to read files in a partitioned environment.

When a pack is too far out of date (that is, its LWM indicates a value prior to the earliest commit-count value in the primary-copy commit list), the system invokes an application-level process to bring the filegroup up to date. At this point, the system lacks sufficient knowledge of the most recent commits to redo the changes. Instead, the site must inspect the entire inode space to determine which files in its pack are out of date.

When a site is lost from an operational Locus network, a clean-up procedure is necessary. Essentially, once a site has decided that a particular site is unavailable, it must invoke failure handling for all the resources that the processes were using at that site. This substantial cleaning procedure is the penalty of the state information kept by all three sites participating in file access.

Since directory updates are not restricted to being applied to the primary copy, conflicts among updates in different partitions may arise. However, because of the simple nature of directory-entry modification, an automatic reconciliation procedure is devised. This procedure is based on comparing the inodes and string-name pairs of replicas of the same directory. The most extreme action taken is when the same name string corresponds to two different inodes. The file's name is altered slightly, and the file's owner is notified by electronic mail.

17.7 ■ Summary

A DFS is a file-service system whose clients, servers, and storage devices are dispersed among the various sites of a distributed system. Accordingly, service activity has to be carried out across the network; instead of a single centralized data repository, there are multiple and independent storage devices.

Ideally, a DFS should look to its clients like a conventional, centralized file system. The multiplicity and dispersion of its servers and storage devices should be made transparent. That is, the client interface of a DFS should not distinguish between local and remote files. It is up to the DFS to locate the files and to arrange for the transport of the data. A transparent DFS facilitates client mobility by bringing over the client's environment to the site where a client logs in.

There are several approaches to naming schemes in a DFS. In the simplest approach, files are named by some combination of their host name and local name, which guarantees a unique systemwide name. Another approach, popularized by NFS, provides means to attach remote directories to local directories, thus giving the appearance of a coherent directory tree.

Requests to access a remote file are usually handled by two complementary methods. With remote service, requests for accesses are delivered to the server. The server machine performs the accesses, and their results are forwarded back to the client. With *caching*, if the data needed to satisfy the access request are not already cached, then a copy of those data is brought from the server to the client. Accesses are performed on the cached copy. The idea is to retain recently accessed disk blocks in the cache, so that repeated accesses to the same information can be handled locally, without additional network traffic. A replacement policy is used to keep the cache size bounded. The problem of keeping the cached copies consistent with the master file is the *cache-consistency problem*.

There are two approaches to server-side information. Either the server tracks each file being accessed by each client, or it simply provides blocks as they are requested by the client without knowledge of their use. These approaches are the stateful versus stateless service paradigms.

Replication of files on different machines is a useful redundancy for improving availability. Multimachine replication can benefit performance too, since selecting a nearby replica to serve an access request results in shorter service time.

■ Exercises

17.1 What are the benefits of a DFS when compared to a file system in a centralized system?

17.2 Which of the example DFSs would handle a large, multiclient database application most efficiently? Explain your answer.

17.3 Under which circumstances would a client prefer a location-transparent DFS? Under which would she prefer a location-independent DFS? Discuss the reasons for these preferences.

17.4 What aspects of a distributed system would you select for a system running on a totally reliable network?

17.5 Compare and contrast the techniques of caching disk blocks locally, on a client system, and remotely, on a server.

17.6 What are the benefits of mapping objects into virtual memory, as Apollo Domain does? What are the detriments?

Bibliographic Notes

A distributed file service based on optimistic concurrency control was described by Mullender and Tanenbaum [1985]. Discussions concerning consistency and recovery control for replicated files were offered by Davcev and Burkhard [1985]. Management of replicated files in a UNIX environment is covered by Brereton [1986] and Purdin et al. [1987]. Wah [1984] discussed the issue of file placement on distributed computer systems.

UNIX United was described by Brownbridge et al. [1982]. The Locus system was discussed by Popek and Walker [1985]. The Sprite system was described by Ousterhout et al. [1988], and Nelson et al. [1988]. Sun's Network File System (NFS) was presented in Sandberg et al. [1985], Sandberg [1987], and Sun Microsystems [1990]. The Andrew system was discussed by Morris et al. [1986], Howard et al. [1988], and Satyanarayanan [1990]. The Apollo Domain system was discussed by Leach et al. [1982].

A detailed survey of mainly centralized file servers was given in Svobodova [1984]. The emphasis there is on support of atomic transactions, and not on location transparency and naming.

CHAPTER 18

DISTRIBUTED COORDINATION

In Chapter 6, we described various mechanisms that allow processes to synchronize their actions. We also discussed a number of schemes to ensure the atomicity property of a transaction that executes either in isolation or concurrently with other transactions. In Chapter 7 we described various methods that an operating system can use to deal with the deadlock problem. In this chapter, we examine how the centralized synchronization mechanisms can be extended to a distributed environment. We also discuss various methods for handling deadlocks in a distributed system.

18.1 ■ Event Ordering

In a centralized system, it is always possible to determine the order in which two events have occurred, since there is a single common memory and clock. In many applications, it is of utmost importance to be able to determine order. For example, in a resource-allocation scheme, we specify that a resource can be used only *after* the resource has been granted. In a distributed system, however, there is no common memory and no common clock. Therefore, it is sometimes impossible to say which of two events occurred first. The *happened-before* relation is only a partial ordering of the events in distributed systems. Since the ability to define a total ordering is crucial in many applications, we present a distributed algorithm for extending the *happened-before* relation to a consistent total ordering of all the events in the system.

18.1.1 The Happened-Before Relation

Since we are considering only sequential processes, all events executed in a single process are totally ordered. Also, by the law of causality, a message can be received only after it has been sent. Therefore, we can define the *happened-before* relation (denoted by $\rightarrow$) on a set of events as follows (assuming that sending and receiving a message constitutes an event):

1. If A and B are events in the same process, and A was executed before B, then $A \rightarrow B$.
2. If A is the event of sending a message by one process and B is the event of receiving that message by another process, then $A \rightarrow B$.
3. If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$.

Since an event cannot happen before itself, the $\rightarrow$ relation is an irreflexive partial ordering.

If two events, A and B, are not related by the $\rightarrow$ relation (that is, A did not happen before B, and B did not happen before A), then we say that these two events were executed *concurrently*. In this case, neither event can causally affect the other. If, however, $A \rightarrow B$, then it is possible for event A to affect event B causally.

The definitions of concurrency and of *happened-before* can best be illustrated by a space–time diagram, such as that in Figure 18.1. The horizontal direction represents space (that is, different processes), and the vertical direction represents time. The labeled vertical lines denote processes (or processors). The labeled dots denote events. A wavy line denotes a message sent from one process to another. From this diagram, it is clear that events A and B are concurrent if and only if no path exists either from A to B or from B to A.

For example, consider Figure 18.1. Some of the events related by the *happened-before* relation are

$$p_1 \rightarrow q_2,$$
$$r_0 \rightarrow q_4,$$
$$q_3 \rightarrow r_1,$$
$$p_1 \rightarrow q_4 \quad (\text{since } p_1 \rightarrow q_2 \text{ and } q_2 \rightarrow q_4).$$

Some of the concurrent events in the system are

$$q_0 \text{ and } p_2,$$
$$r_0 \text{ and } q_3,$$
$$r_0 \text{ and } p_3,$$
$$q_3 \text{ and } p_3.$$

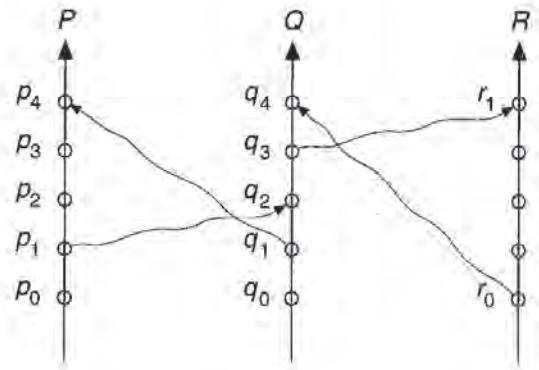


Figure 18.1 Relative time for three concurrent processes.

We cannot know which of two concurrent events, such as q_0 and p_2, happened first. However, since neither event can affect the other (there is no way for one of them to know whether the other has occurred yet), it is not important which of them happened first. It is important only that any processes that care about the order of two concurrent events agree on some order.

18.1.2 Implementation

To determine that an event A happened before an event B, we need either a common clock or a set of perfectly synchronized clocks. Since, in a distributed system neither of these is available, we must define the *happened-before* relation without the use of physical clocks.

We associate with each system event a *timestamp*. We can then define the *global ordering* requirement: For every pair of events A and B, if A → B, then the timestamp of A is less than the timestamp of B. Below we will see that the converse does not need to be true.

How do we enforce the global ordering requirement in a distributed environment? We define within *each* process P_i a *logical clock*, LC_i. The logical clock can be implemented as a simple counter that is incremented between any two successive events executed within a process. Since the logical clock has a monotonically increasing value, it assigns a unique number to every event, and if an event A occurs before event B in process P_i, then $LC_i(A) < LC_i(B)$. The timestamp for an event is the value of the logical clock for that event. Clearly, this scheme ensures that, for any two events in the same process, the global ordering requirement is met.

Unfortunately, this scheme does not ensure that the global ordering requirement is met across processes. To illustrate the problem, we consider two processes P_1 and P_2 that communicate with each other. Suppose that P_1 sends a message to P_2 (event A) with $LC_1(A) = 200$, and P_2 receives the message (event B) with $LC_2(B) = 195$ (because the processor for P_2 is

slower than the processor for P_1 and so its logical clock ticks slower). This situation violates our requirement, since A → B, but the timestamp of A is greater than the timestamp of B.

To resolve this difficulty, we require a process to advance its logical clock when it receives a message whose timestamp is greater than the current value of its logical clock. In particular, if process P_i receives a message (event B) with timestamp t and $LC_i(B) \leq t$, then it should advance its clock such that $LC_i(B) = t+1$. Thus, in our example, when P_2 receives the message from P_1, it will advance its logical clock such that $LC_2(B) = 201$.

Finally, to realize a total ordering, we need only to observe that, with our timestamp ordering scheme, if the timestamps of two events A and B are the same, then the events are concurrent. In this case, we may use process identity numbers to break ties and to create a total ordering. The use of timestamps is discussed later in this chapter.

18.2 ■ Mutual Exclusion

In this section, we present a number of different algorithms for implementing mutual exclusion in a distributed environment. We assume that the system consists of n processes, each of which resides at a different processor. To simplify our discussion, we assume that processes are numbered uniquely from 1 to n, and that there is a one-to-one mapping between processes and processors (that is, each process has its own processor).

18.2.1 Centralized Approach

In a centralized approach to providing mutual exclusion, one of the processes in the system is chosen to coordinate the entry to the critical section. Each process that wants to invoke mutual exclusion sends a *request* message to the coordinator. When the process receives a *reply* message from the coordinator, it can proceed to enter its critical section. After exiting its critical section, the process sends a *release* message to the coordinator and proceeds with its execution.

On receiving a *request* message, the coordinator checks to see whether some other process is in its critical section. If no process is in its critical section, the coordinator immediately sends back a *reply* message. Otherwise, the request is queued. When the coordinator receives a *release* message, it removes one of the request messages from the queue (in accordance with some scheduling algorithm) and sends a *reply* message to the requesting process.

It should be clear that this algorithm ensures mutual exclusion. In addition, if the scheduling policy within the coordinator is fair (such as

first-come, first-served scheduling), no starvation can occur. This scheme requires three messages per critical-section entry: a *request*, a *reply*, and a *release*.

If the coordinator process fails, then a new process must take its place. In Section 18.6, we describe various algorithms for electing a unique new coordinator. Once a new coordinator has been elected, it must poll all the processes in the system, to reconstruct its *request* queue. Once the queue has been constructed, the computation may resume.

18.2.2 Fully Distributed Approach

If we want to distribute the decision making across the entire system, then the solution is far more complicated. We present an algorithm that is based on the event-ordering scheme described in Section 18.1.

When a process P_i wants to enter its critical section, it generates a new timestamp, *TS*, and sends the message *request*(P_i, *TS*) to all other processes in the system (including itself). On receiving a *request* message, a process may reply immediately (that is, send a *reply* message back to P_i), or it may defer sending a reply back (because it is already in its critical section, for example). A process that has received a *reply* message from all other processes in the system can enter its critical section, queueing incoming requests and deferring them. After exiting its critical section, the process sends *reply* messages to all its deferred requests.

The decision whether process P_i replies immediately to a *request*(P_j, *TS*) message or defers its reply is based on three factors:

1. If process P_i is in its critical section, then it defers its reply to P_j.
2. If process P_i does *not* want to enter its critical section, then it sends a reply immediately to P_j.
3. If process P_i wants to enter its critical section but has not yet entered it, then it compares its own *request* timestamp with the timestamp *TS* of the incoming request made by process P_j. If its own *request* timestamp is greater than *TS*, then it sends a reply immediately to P_j (P_j asked first). Otherwise, the reply is deferred.

This algorithm exhibits the following desirable behavior:

- Mutual exclusion is obtained.
- Freedom from deadlock is ensured.
- Freedom from starvation is ensured, since entry to the critical section is scheduled according to the timestamp ordering. The timestamp ordering ensures that processes are served in a first-come, first-served order.

- The number of messages per critical-section entry is $2 \times (n - 1)$. This number is the minimum number of required messages per critical-section entry when processes act independently and concurrently.

To illustrate how the algorithm functions, we consider a system consisting of processes P_1, P_2, and P_3. Suppose that processes P_1 and P_3 want to enter their critical sections. Process P_1 then sends a message *request* (P_1, timestamp = 10) to processes P_2 and P_3, while process P_3 sends a message *request* (P_3, timestamp = 4) to processes P_1 and P_2. The timestamps 4 and 10 were obtained from the logical clocks described in Section 18.1. When process P_2 receives these *request* messages, it replies immediately. When process P_1 receives the *request* from process P_3 it replies immediately, since the timestamp (10) on its own *request* message is greater than the timestamp (4) for process P_3. When process P_3 receives the *request* from process P_1, it defers its reply, since the timestamp (4) on its *request* message is less than the timestamp (10) for the message of process P_1. On receiving replies from both process P_1 and process P_2, process P_3 can enter its critical section. After exiting its critical section, process P_3 sends a reply to process P_1, which can then enter its critical section.

Note that this scheme requires the participation of all the processes in the system. This approach has three undesirable consequences:

1. The processes need to know the identity of all other processes in the system. When a new process joins the group of processes participating in the mutual-exclusion algorithm, the following actions need to be taken:

   a. The process must receive the names of all the other processes in the group.

   b. The name of the new process must be distributed to all the other processes in the group.

   This task is not as trivial as it may seem, since some *request* and *reply* messages may be circulating in the system when the new process joins the group. The interested reader is referred to the Bibliographic Notes for more details.

2. If one of the processes fails, then the entire scheme collapses. We can resolve this difficulty by continuously monitoring the state of all the processes in the system. If one process fails, then all other processes are notified, so that they will no longer send *request* messages to the failed process. When a process recovers, it must initiate the procedure that allows it to rejoin the group.

3. Processes that have not entered their critical section must pause frequently to assure other processes that they intend to enter the critical section. This protocol is therefore suited for small, stable sets of cooperating processes.

18.2.3 Token-Passing Approach

Another method of providing mutual exclusion is to circulate a token among the processes in the system. A *token* is a special type of message that is passed around the system. Possession of the token entitles the holder to enter the critical section. Since there is only a single token in the system, only one process can be in its critical section at a time.

We assume that the processes in the system are *logically* organized in a ring structure. The physical communication network does not need to be a ring. As long as the processes are connected to one another, it is possible to implement a logical ring. To implement mutual exclusion, we pass the token around the ring. When a process receives the token, it may enter its critical section, keeping the token. After the process exits its critical section, the token is passed around again. If the process receiving the token does not want to enter its critical section, it passes the token to its neighbor. This scheme is similar to algorithm 1 in Chapter 6, but a token is substituted for a shared variable.

Since there is only a single token, only one process at a time can be in its critical section. In addition, if the ring is unidirectional, freedom from starvation is ensured. The number of messages required to implement mutual exclusion may vary from one message per entry, in the case of high contention (that is, every process wants to enter its critical section), to an infinite number of messages, in the case of low contention (that is, no process wants to enter its critical section).

Two types of failure must be considered. First, if the token is lost, an election must be called to generate a new token. Second, if a process fails, a new logical ring must be established. There are several different algorithms for election and for reconstructing a logical ring. In Section 18.6, we present an election algorithm. The development of an algorithm for reconstructing the ring is left to you in Exercise 18.6.

18.3 ■ Atomicity

In Chapter 6, we introduced the concept of an atomic transaction, which is a program unit that must be executed *atomically*. That is, either all the operations associated with it are executed to completion, or none are performed. When we are dealing with a distributed system, it becomes much more complicated to ensure the atomicity property of a transaction,

as compared to in a centralized system. This difficulty occurs because several sites may be participating in the execution of a single transaction. The failure of one of these sites, or the failure of a communication link connecting these sites, may result in erroneous computations.

It is the function of the *transaction coordinator* of a distributed system to ensure that the execution of the various transactions in the distributed system preserves atomicity. Each site has its own local transaction coordinator that is responsible for coordinating the execution of all the transactions initiated at that site. For each such transaction, the coordinator is responsible for the following:

- Starting the execution of the transaction
- Breaking the transaction into a number of subtransactions, and distributing these subtransactions to the appropriate sites for execution
- Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites

We assume that each local site maintains a log for recovery purposes.

18.3.1 The Two-Phase Commit Protocol

For atomicity to be ensured, all the sites in which a transaction T executed must agree on the final outcome of the execution. T must either commit at all sites or it must abort at all sites. To ensure this property, the transaction coordinator of T must execute a *commit protocol*. Among the simplest and most widely used commit protocols is the *two-phase commit (2PC)* protocol, which we discuss here.

Let T be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i. When T completes its execution — that is, when all the sites at which T has executed inform C_i that T has completed — then C_i starts the 2PC protocol.

- **Phase 1:** C_i adds the record <prepare T> to the log and forces the record onto stable storage. It then sends a prepare T message to all sites at which T executed. On receiving such a message, the transaction manager at that site determines whether it is willing to commit its portion of T. If the answer is no, it adds a record <no T> to the log, and then it responds by sending an abort T message to C_i. If the answer is yes, it adds a record <ready T> to the log, and forces all the log records corresponding to T onto stable storage. The transaction manager then replies with a ready T message to C_i.
- **Phase 2:** When C_i receives responses to the prepare T message from all the sites, or when a prespecified interval of time has elapsed since the prepare T message was sent out, C_i can determine whether the

transaction T can be committed or aborted. Transaction T can be committed if C_i received a ready T message from all the participating sites. Otherwise, transaction T must be aborted. Depending on the verdict, either a record <commit T> or a record <abort T> is added to the log and is forced onto stable storage. At this point, the fate of the transaction has been sealed. Following this, the coordinator sends either a commit T or an abort T message to all participating sites. When a site receives that message, it records the message in the log.

A site at which T executed can unconditionally abort T at any time prior to its sending the message ready T to the coordinator. The ready T message is, in effect, a promise by a site to follow the coordinator's order to commit T or to abort T. The only situation in which a site can make such a promise is if the needed information is stored in stable storage. Otherwise, if the site crashes after sending ready T, it may be unable to make good on its promise.

Since unanimity is required to commit a transaction, the fate of T is sealed as soon as at least one site responds abort T. Since the coordinator site S_i is one of the sites at which T executed, the coordinator can decide unilaterally to abort T. The final verdict regarding T is determined at the time the coordinator writes that verdict (commit or abort) to the log and forces it to stable storage. In some implementations of the 2PC protocol, a site sends an acknowledge T message to the coordinator at the end of the second phase of the protocol. When the coordinator receives the acknowledge T message from all the sites, it adds the record <complete T> to the log.

18.3.2 Failure Handling in 2PC

We now examine in detail how 2PC responds to various types of failures. As we shall see, one major disadvantage of the 2PC protocol is that coordinator failure may result in blocking, where a decision either to commit or to abort T may have to be postponed until C_i recovers.

18.3.2.1 Failure of a Participating Site

When a participating site S_k recovers from a failure, it must examine its log to determine the fate of those transactions that were in the midst of execution when the failure occurred. Let T be one such transaction. We consider each of the possible cases:

- The log contains a <commit T> record. In this case, the site executes **redo**(T).
- The log contains an <abort T> record. In this case, the site executes **undo**(T).

- The log contains a <ready T> record. In this case, the site must consult C_i to determine the fate of T. If C_i is up, it notifies S_k regarding whether T committed or aborted. In the former case, it executes **redo**(T); in the latter case, it executes **undo**(T). If C_i is down, S_k must try to find the fate of T from other sites. It does so by sending a **query-status** T message to all the sites in the system. On receiving such a message, a site must consult its log to determine whether T has executed there, and if so, whether T committed or aborted. It then notifies S_k about this outcome. If no site has the appropriate information (that is, whether T committed or aborted), then S_k can neither abort nor commit T. The decision concerning T is postponed until S_k can obtain the needed information. Thus, S_k must periodically resend the **query-status** message to the other sites. It does so until a site recovers that contains the needed information. Note that the site at which C_i resides always has the needed information.
- The log contains no control records (abort, commit, ready) concerning T. The absence of control records implies that S_k failed before responding to the prepare T message from C_i. Since the failure of S_k precludes the sending of such a response, by our algorithm C_i must abort T. Hence, S_k must execute **undo**(T).

18.3.2.2 Failure of the Coordinator

If the coordinator fails in the midst of the execution of the commit protocol for transaction T, then the participating sites must decide on the fate of T. We shall see that, in certain cases, the participating sites cannot decide whether to commit or abort T, and therefore it is necessary for these sites to wait for the recovery of the failed coordinator.

- If an active site contains a <commit T> record in its log, then T must be committed.
- If an active site contains an <abort T> record in its log, then T must be aborted.
- If some active site does *not* contain a <ready T> record in its log, then the failed coordinator C_i cannot have decided to commit T. We can draw this conclusion because a site that does not have a <ready T> record in its log cannot have sent a ready T message to C_i. However, the coordinator may have decided to abort T, but not to commit T. Rather than wait for C_i to recover, it is preferable to abort T.
- If none of the preceding cases holds, then all active sites must have a <ready T> record in their logs, but no additional control records (such

as <abort T> or <commit T>). Since the coordinator has failed, it is impossible to determine whether a decision has been made, or what that decision is, until the coordinator recovers. Thus, the active sites must wait for C_i to recover. Since the fate of T remains in doubt, T may continue to hold system resources. For example, if locking is used, T may hold locks on data at active sites. Such a situation is undesirable because it may take hours or days before C_i is again active. During this time other transactions may be forced to wait for T. As a result, data are unavailable not only on the failed site (C_i) but on active sites as well. The number of unavailable data increases as the downtime of C_i grows. This situation is called the *blocking* problem, because T is blocked pending the recovery of site C_i.

18.3.2.3 Failure of the Network

When a link fails, all the messages that are in the process of being routed through the link do not arrive at their destination intact. From the viewpoint of the sites connected throughout that link, it appears that the other sites have failed. Thus, our previous schemes apply here as well.

When a number of links fail, the network may partition. In this case, two possibilities exist. The coordinator and all its participants may remain in one partition; in this case, the failure has no effect on the commit protocol. Alternatively, the coordinator and its participants may belong to several partitions; in this case, messages between the participant and the coordinator are lost, reducing the case to a link failure, as discussed.

18.4 ■ Concurrency Control

In this section, we show how certain of the concurrency-control schemes discussed in Chapter 6 can be modified so that they can be used in a distributed environment.

It is the function of the *transaction manager* of a distributed database system to manage the execution of those transactions (or subtransactions) that access data stored in a local site. Note that each such transaction may be either a local transaction (that is, a transaction that only executes at that site) or part of a global transaction (that is, a transaction that executes at several sites). Each transaction manager is responsible for maintaining a log for recovery purposes, and for participating in an appropriate concurrency-control scheme to coordinate the concurrent execution of the transactions executing at that site. As we shall see, the concurrency schemes described in Chapter 6 need to be modified to accommodate the distribution of transactions.

18.4.1 Locking Protocols

The two-phase locking protocols described in Chapter 6 can be used in a distributed environment. The only change that needs to be incorporated is in the way the lock manager is implemented. In this section, we present several possible schemes, the first of which deals with the case where no data replication is allowed. The other schemes are applicable to the more general case where data can be replicated in several sites. As in Chapter 6, we shall assume the existence of the *shared* and *exclusive* lock modes.

18.4.1.1 Nonreplicated Scheme

If no data are replicated in the system, then the locking schemes described in Section 6.9 can be applied as follows. Each site maintains a local lock manager whose function is to administer the lock and unlock requests for those data items that are stored in that site. When a transaction wishes to lock data item Q at site S_i, it simply sends a message to the lock manager at site S_i requesting a lock (in a particular lock mode). If data item Q is locked in an incompatible mode, then the request is delayed until that request can be granted. Once it has been determined that the lock request can be granted, the lock manager sends a message back to the initiator indicating that the lock request has been granted.

The scheme has the advantage of simple implementation. It requires two message transfers for handling lock requests, and one message transfer for handling unlock requests. However, deadlock handling is more complex. Since the lock and unlock requests are no longer made at a single site, the various deadlock-handling algorithms discussed in Chapter 7 must be modified, as will be discussed in Section 18.5.

18.4.1.2 Single-Coordinator Approach

Under the single-coordinator approach, the system maintains a *single* lock manager that resides in a *single* chosen site, say S_i. All lock and unlock requests are made at site S_i. When a transaction needs to lock a data item, it sends a lock request to S_i. The lock manager determines whether the lock can be granted immediately. If so, it sends a message to that effect to the site at which the lock request was initiated. Otherwise, the request is delayed until it can be granted, at which time a message is sent to the site at which the lock request was initiated. The transaction can read the data item from *any* one of the sites at which a replica of the data item resides. In the case of a **write**, all the sites where a replica of the data item resides must be involved in the writing.

The scheme has the following advantages:.

- **Simple implementation:** This scheme requires two messages for handling lock requests, and one message for handling unlock requests.

- **Simple deadlock handling:** Since all lock and unlock requests are made at one site, the deadlock-handling algorithms discussed in Chapter 7 can be applied directly to this environment.

The disadvantages of the scheme include the following:

- **Bottleneck:** The site S_i becomes a bottleneck, since all requests must be processed there.
- **Vulnerability:** If the site S_i fails, the concurrency controller is lost. Either processing must stop or a recovery scheme must be used.

A compromise between these advantages and disadvantages can be achieved through a *multiple-coordinator approach,* in which the lock-manager function is distributed over several sites.

Each lock manager administers the lock and unlock requests for a subset of the data items. Each lock manager resides in a different site. This distribution reduces the degree to which the coordinator is a bottleneck, but it complicates deadlock handling, since the lock and unlock requests are not made at one single site.

18.4.1.3 Majority Protocol

The majority protocol is a modification of the nonreplicated data scheme that we presented earlier. The system maintains a lock manager at each site. Each manager manages the locks for all the data or replicas of data stored at that site. When a transaction wishes to lock a data item Q, which is replicated in n different sites, it must send a lock request to more than one-half of the n sites in which Q is stored. Each lock manager determines whether the lock can be granted immediately (as far as it is concerned). As before, the response is delayed until the request can be granted. The transaction does not operate on Q until it has successfully obtained a lock on a majority of the replicas of Q.

This scheme deals with replicated data in a decentralized manner, thus avoiding the drawbacks of central control. However, it suffers from its own disadvantages:

- **Implementation:** The majority protocol is more complicated to implement than the previous schemes. It requires $2(n/2 + 1)$ messages for handling lock requests, and $(n/2 + 1)$ messages for handling unlock requests.
- **Deadlock handling:** Since the lock and unlock requests are not made at one site, the deadlock-handling algorithms must be modified (see Section 18.5). In addition, it is possible for a deadlock to occur even if only one data item is being locked. To illustrate, consider a system

with four sites and full replication. Suppose that transactions T_1 and T_2 wish to lock data item Q in exclusive mode. Transaction T_1 may succeed in locking Q at sites S_1 and S_3, while transaction T_2 may succeed in locking Q at sites S_2 and S_4. Each then must wait to acquire the third lock, and hence a deadlock has occurred.

18.4.1.4 Biased Protocol

The biased protocol is based on a model similar to that of the majority protocol. The difference is that requests for shared locks are given more favorable treatment than are requests for exclusive locks. The system maintains a lock manager at each site. Each manager manages the locks for all the data items stored at that site. *Shared* and *exclusive* locks are handled differently.

- **Shared locks:** When a transaction needs to lock data item Q, it simply requests a lock on Q from the lock manager at one site containing a replica of Q.
- **Exclusive locks:** When a transaction needs to lock data item Q, it requests a lock on Q from the lock manager at all sites containing a replica of Q.

As before, the response to the request is delayed until the request can be granted.

The scheme has the advantage of imposing less overhead on **read** operations than does the majority protocol. This advantage is especially significant in common cases in which the frequency of **reads** is much greater than is the frequency of **writes.** However, the additional overhead on **writes** is a disadvantage. Furthermore, the biased protocol shares the majority protocol's disadvantage of complexity in handling deadlock.

18.4.1.5 Primary Copy

In the case of data replication, we may choose one of the replicas as the primary copy. Thus, for each data item Q, the primary copy of Q must reside in precisely one site, which we call the *primary site of Q*.

When a transaction needs to lock a data item Q, it requests a lock at the primary site of Q. As before, the response to the request is delayed until the request can be granted.

Thus, the primary copy enables concurrency control for replicated data to be handled in a manner similar to that for unreplicated data. This method of handling allows for a simple implementation. However, if the primary site of Q fails, Q is inaccessible even though other sites containing a replica may be accessible.

18.4.2 Timestamping

The principal idea behind the timestamping scheme discussed in Section 6.9 is that each transaction is given a *unique* timestamp that is used in deciding the serialization order. Our first task, then, in generalizing the centralized scheme to a distributed scheme is to develop a scheme for generating unique timestamps. Once this scheme has been developed, our previous protocols can be applied directly to the nonreplicated environment.

18.4.2.1 Generation of Unique Timestamps

There are two primary methods for generating unique timestamps, one centralized and one distributed. In the centralized scheme, a single site is chosen for distributing the timestamps. The site can use a logical counter or its own local clock for this purpose.

In the distributed scheme, each site generates a unique local timestamp using either a logical counter or the local clock. The global unique timestamp is obtained by concatenation of the unique local timestamp with the site identifier, which must be unique (Figure 18.2). The order of concatenation is important! We use the site identifier in the least significant position to ensure that the global timestamps generated in one site are not always greater than those generated in another site. Compare this technique for generating unique timestamps with the one we presented in Section 18.1.2 for generating unique names.

We may still have a problem if one site generates local timestamps at a faster rate than do other sites. In such a case, the fast site's logical counter will be larger than that of other sites. Therefore, all timestamps generated by the fast site will be larger than those generated by other sites. What is needed is a mechanism to ensure that local timestamps are generated fairly across the system. To accomplish the fair generation of timestamps, we define within each site S_i a *logical clock* (LC_i), which generates the unique local timestamp. The logical clock can be implemented as a counter that is

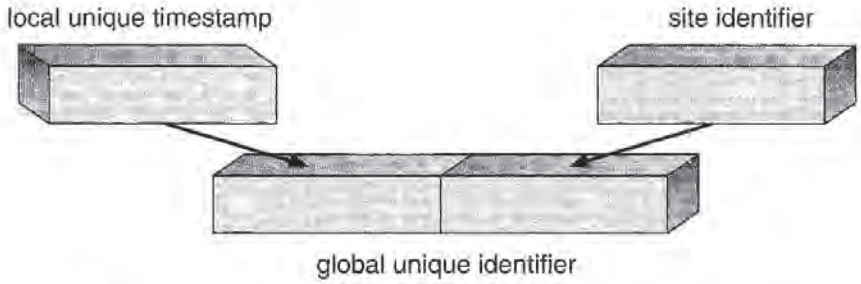


Figure 18.2 Generation of unique timestamps.

incremented after a new local timestamp is generated. To ensure that the various logical clocks are synchronized, we require that a site S_i advance its logical clock whenever a transaction T_i with timestamp $<x,y>$ visits that site and x is greater than the current value of LC_i. In this case, site S_i advances its logical clock to the value $x + 1$.

If the system clock is used to generate timestamps, then timestamps are assigned fairly provided that no site has a system clock that runs fast or slow. Since clocks may not be perfectly accurate, a technique similar to that used for logical clocks must be used to ensure that no clock gets far ahead or far behind another clock.

18.4.2.2 Timestamp-Ordering Scheme

The basic timestamp scheme introduced in Section 6.9 can be extended in a straightforward manner to a distributed system. As in the centralized case, cascading rollbacks may result if no mechanism is used to prevent a transaction from reading a data item value that is not yet committed. To eliminate cascading rollbacks, we can combine the basic timestamp scheme of Section 6.9 with the 2PC protocol of Section 18.3 to obtain a protocol that ensures serializability with no cascading rollbacks. We leave the development of such an algorithm to you.

The basic timestamp scheme just described suffers from the undesirable property that conflicts between transactions are resolved through rollbacks, rather than through waits. To alleviate this problem, we can buffer the various **read** and **write** operations (that is, *delay* them) until a time when we are assured that these operations can take place without causing aborts. A **read**(x) operation by T_i must be delayed if there exists a transaction T_j that will perform a **write**(x) operation but has not yet done so, and $TS(T_j) < TS(T_i)$. Similarly, a **write**(x) operation by T_i must be delayed if there exists a transaction T_j that will perform either **read**(x) or **write**(x) operation and $TS(T_j) < TS(T_i)$. There are various methods for ensuring this property. One such method, called the *conservative timestamp-ordering scheme*, requires each site to maintain a **read** and **write** queue consisting of all the **read** and **write** requests, respectively, that are to be executed at the site and that must be delayed to preserve the above property. We shall not present the scheme here. Rather, we leave the development of the algorithm to you.

18.5 ■ Deadlock Handling

The deadlock-prevention, deadlock-avoidance, and deadlock-detection algorithms presented in Chapter 7 can be extended so that they can also be used in a distributed system. In the following, we describe several of these distributed algorithms.

18.5.1 Deadlock Prevention

The deadlock-prevention and deadlock-avoidance algorithms presented in Chapter 7 can also be used in a distributed system, provided that appropriate modifications are made. For example, we can use the resource-ordering deadlock-prevention technique by simply defining a *global* ordering among the system resources. That is, all resources in the entire system are assigned unique numbers, and a process may request a resource (at any processor) with unique number i only if it is not holding a resource with a unique number greater than i. Similarly, we can use the banker's algorithm in a distributed system by designating one of the processes in the system (the *banker*) as the process that maintains the information necessary to carry out the banker's algorithm. Every resource request must be channeled through the banker.

These two schemes can be used in dealing with the deadlock problem in a distributed environment. The first scheme is simple to implement and requires little overhead. The second scheme can also be implemented easily, but it may require too much overhead. The banker may become a bottleneck, since the number of messages to and from the banker may be large. Thus, the banker's scheme does not seem to be of practical use in a distributed system.

In this section, we present a new deadlock-prevention scheme that is based on a timestamp-ordering approach with resource preemption. For simplicity, we consider only the case of a single instance of each resource type.

To control the preemption, we assign a unique priority number to each process. These numbers are used to decide whether a process P_i should wait for a process P_j. For example, we can let P_i wait for P_j if P_i has a priority higher than that of P_j; otherwise P_i is rolled back. This scheme prevents deadlocks because, for every edge $P_i \rightarrow P_j$ in the wait-for graph, P_i has a higher priority than P_j. Thus, a cycle cannot exist.

One difficulty with this scheme is the possibility of starvation. Some processes with extremely low priority may always be rolled back. This difficulty can be avoided through the use of timestamps. Each process in the system is assigned a unique timestamp when it is created. Two complementary deadlock-prevention schemes using timestamps have been proposed:

- **The wait–die scheme:** This approach is based on a nonpreemptive technique. When process P_i requests a resource currently held by P_j, P_i is allowed to wait only if it has a smaller timestamp than does P_j (that is, P_i is older than P_j). Otherwise, P_i is rolled back (dies). For example, suppose that processes P_1, P_2, and P_3 have timestamps 5, 10, and 15, respectively. If P_1 requests a resource held by P_2, P_1 will wait. If P_3 requests a resource held by P_2, P_3 will be rolled back.

- **The wound–wait scheme:** This approach is based on a preemptive technique and is a counterpart to the wait–die system. When process P_i requests a resource currently held by P_j, P_i is allowed to wait only if it has a larger timestamp than does P_j (that is, P_i is younger than P_j). Otherwise, P_j is rolled back (P_j is *wound* by P_i). Returning to our previous example, with processes P_1, P_2, and P_3, if P_1 requests a resource held by P_2, then the resource will be preempted from P_2 and P_2 will be rolled back. If P_3 requests a resource held by P_2, then P_3 will wait.

Both schemes can avoid starvation, provided that, when a process is rolled back, it is *not* assigned a new timestamp. Since timestamps always increase, a process that is rolled back will eventually have the smallest timestamp. Thus, it will not be rolled back again. There are, however, significant differences in the way the two schemes operate.

- In the wait–die scheme, an older process must wait for a younger one to release its resource. Thus, the older the process gets, the more it tends to wait. By contrast, in the wound–wait scheme, an older process never waits for a younger process.
- In the wait–die scheme, if a process P_i dies and is rolled back because it requested a resource held by process P_j, then P_i may reissue the same sequence of requests when it is restarted. If the resource is still held by P_j, then P_i will die again. Thus, P_i may die several times before acquiring the needed resource. Contrast this series of events with what happens in the wound–wait scheme. Process P_i is wounded and rolled back because P_j requested a resource it holds. When P_i is restarted and requests the resource now being held by P_j, P_i waits. Thus, there are fewer rollbacks in the wound–wait scheme.

The major problem with these two schemes is that unnecessary rollbacks may occur.

18.5.2 Deadlock Detection

The deadlock-prevention algorithm may preempt resources even if no deadlock has occurred. To prevent unnecessary preemptions, we can use a deadlock-detection algorithm. We construct a wait-for graph describing the resource-allocation state. Since we are assuming only a single resource of each type, a cycle in the wait-for graph represents a deadlock.

The main problem in a distributed system is deciding how to maintain the wait-for graph. We elaborate this problem by describing several common techniques to deal with this issue. These schemes require that each site keep a *local* wait-for graph. The nodes of the graph correspond

to all the processes (local as well as nonlocal) that are currently either holding or requesting any of the resources local to that site. For example, in Figure 18.3 we have a system consisting of two sites, each maintaining its local wait-for graph. Note that processes P_2 and P_3 appear in both graphs, indicating that the processes have requested resources at both sites.

These local wait-for graphs are constructed in the usual manner for local processes and resources. When a process P_i in site A needs a resource held by process P_j in site B, a request message is sent by P_i to site B. The edge $P_i \rightarrow P_j$ is then inserted in the local wait-for graph of site B.

Clearly, if any local wait-for graph has a cycle, deadlock has occurred. On the other hand, the fact that there are no cycles in any of the local wait-for graphs does not mean that there are no deadlocks. To illustrate this problem, we consider the system depicted in Figure 18.3. Each wait-for graph is acyclic; nevertheless, a deadlock exists in the system. To prove that a deadlock has not occurred, we must show that the *union* of all local graphs is acyclic. The graph (shown in Figure 18.4) that we obtain by taking the union of the two wait-for graphs of Figure 18.3 does indeed contain a cycle, implying that the system is in a deadlock state.

There are a number of different methods for organizing the wait-for graph in a distributed system. We shall describe several common schemes.

18.5.2.1 Centralized Approach

In the centralized approach, a global wait-for graph is constructed as the union of all the local wait-for graphs. It is maintained in a *single* process: the *deadlock-detection coordinator*. Since there is communication delay in the system, we must distinguish between two types of wait-for graphs. The *real* graph describes the real but unknown state of the system at any instance in time, as would be seen by an omniscient observer. The *constructed* graph is an approximation generated by the coordinator during the execution of its algorithm. Obviously, the constructed graph must be generated such that, whenever the detection algorithm is invoked, the

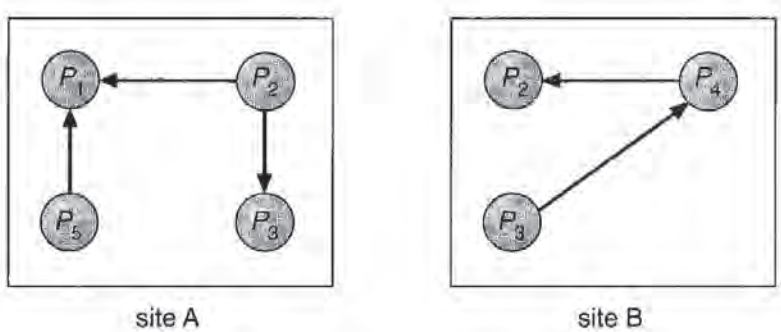


Figure 18.3 Two local wait-for graphs.

reported results are correct in a sense that, if a deadlock exists, then it is reported properly, and if a deadlock is reported, then the system is indeed in a deadlock state. As we shall show, it is not easy to construct such correct algorithms.

There are three different options (points in time) when the wait-for graph may be constructed:

1. Whenever a new edge is inserted or removed in one of the local wait-for graphs
2. Periodically, when a number of changes have occurred in a wait-for graph
3. Whenever the coordinator needs to invoke the cycle-detection algorithm

Let us consider option 1. Whenever an edge is either inserted or removed in a local graph, the local site must also send a message to the coordinator to notify it of this modification. On receiving such a message, the coordinator updates its global graph. Alternatively, a site can send a number of such changes in a single message periodically. Returning to our previous example, the coordinator process will maintain the global wait-for graph as depicted in Figure 18.4. When site B inserts the edge $P_3 \rightarrow P_4$ in its local wait-for graph, it also sends a message to the coordinator. Similarly, when site A deletes the edge $P_5 \rightarrow P_1$, because P_1 has released a resource that was requested by P_5, an appropriate message is sent to the coordinator.

When the deadlock-detection algorithm is invoked, the coordinator searches its global graph. If a cycle is found, a victim is selected to be rolled back. The coordinator must notify all the sites that a particular process has been selected as victim. The sites, in turn, roll back the victim process.

Note that, in this scheme (option 1), unnecessary rollbacks may occur, as a result of two situations:

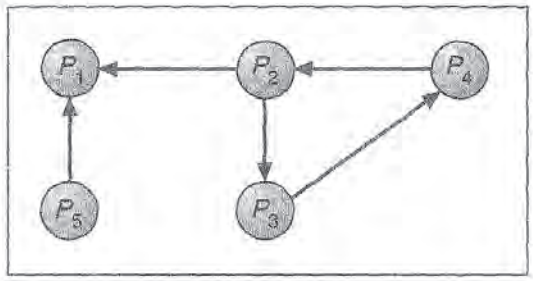


Figure 18.4 Global wait-for graph for Figure 18.3.

- *False cycles* may exist in the global wait-for graph. To illustrate this point, we consider a snapshot of the system as depicted in Figure 18.5. Suppose that P_2 releases the resource it is holding in site A, resulting in the deletion of the edge $P_1 \rightarrow P_2$ in A. Process P_2 then requests a resource held by P_3 at site B, resulting in the addition of the edge $P_2 \rightarrow P_3$ in B. If the *insert* $P_2 \rightarrow P_3$ message from B arrives before the *delete* $P_1 \rightarrow P_2$ message from A, the coordinator may discover the false cycle $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_1$ after the *insert* (but before the *delete*). Deadlock recovery may be initiated, although no deadlock has occurred.
- Unnecessary rollbacks may also result when a *deadlock* has indeed occurred and a victim has been picked, but at the same time one of the processes was aborted for reasons unrelated to the deadlock (such as the process exceeding its allocated time). For example, suppose that site A in Figure 18.3 decides to abort P_2. At the same time, the coordinator has discovered a cycle and picked P_3 as a victim. Both P_2 and P_3 are now rolled back, although only P_2 needed to be rolled back.

Note that the same problems are inherited in solutions employing the other two options (that is, options 2 and 3).

Let us now present a centralized deadlock-detection algorithm, using option 3, which detects all deadlocks that actually occur, and does not detect false deadlocks. To avoid the report of false deadlocks, we require that requests from different sites be appended with unique identifiers (timestamps). When process P_i, at site A, requests a resource from P_j, at site B, a request message with timestamp TS is sent. The edge $P_i \rightarrow P_j$ with the label TS is inserted in the local wait-for of A. This edge is inserted in the local wait-for graph of B only if B has received the request message and cannot immediately grant the requested resource. A request from P_i to P_j in the same site is handled in the usual manner; no timestamps are associated with the edge $P_i \rightarrow P_j$. The detection algorithm is then as follows:

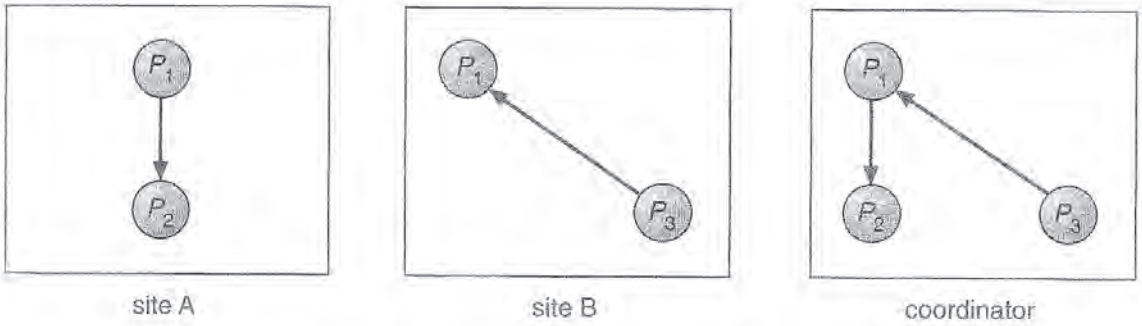


Figure 18.5 Local and global wait-for graphs.

1. The controller sends an initiating message to each site in the system.
2. On receiving this message, a site sends its local wait-for graph to the coordinator. Note that each of these wait-for graphs contains all the local information the site has about the state of the real graph. The graph reflects an instantaneous state of the site, but it is not synchronized with respect to any other site.
3. When the controller has received a reply from each site, it constructs a graph as follows:

   a. The constructed graph contains a vertex for every process in the system.

   b. The graph has an edge $P_i \rightarrow P_j$ if and only if (1) there is an edge $P_i \rightarrow P_j$ in one of the wait-for graphs, or (2) an edge $P_i \rightarrow P_j$ with some label *TS* appears in more than one wait-for graph.

We assert that, if there is a cycle in the constructed graph, then the system is in a deadlock state. If there is no cycle in the constructed graph, then the system was not in a deadlock state when the detection algorithm was invoked as result of the initiating messages sent by the coordinator (in step 1).

18.5.2.2 Fully Distributed Approach

In the *fully distributed* deadlock-detection algorithm, all controllers share equally the responsibility for detecting deadlock. In this scheme, every site constructs a wait-for graph that represents a part of the total graph, depending on the dynamic behavior of the system. The idea is that, if a deadlock exists, a cycle will appear in (at least) one of the partial graphs. We present one such algorithm, which involves construction of partial graphs in every site.

Each site maintains its own local wait-for graph. A local wait-for graph in this scheme differs from the one described earlier in that we add one additional node P_{ex} to the graph. An arc $P_i \rightarrow P_{ex}$ exists in the graph if P_i is waiting for a data item in another site being held by *any* process. Similarly, an arc $P_{ex} \rightarrow P_j$ exists in the graph if there exists a process at another site that is waiting to acquire a resource currently being held by P_j in this local site.

To illustrate this situation, we consider the two local wait-for graphs of Figure 18.3. The addition of the node P_{ex} in both graphs results in the local wait-for graphs shown in Figure 18.6.

If a local wait-for graph contains a cycle that does not involve node P_{ex}, then the system is in a deadlock state. If, however, there exists a cycle involving P_{ex}, then this implies that there is a *possibility* of a deadlock. To

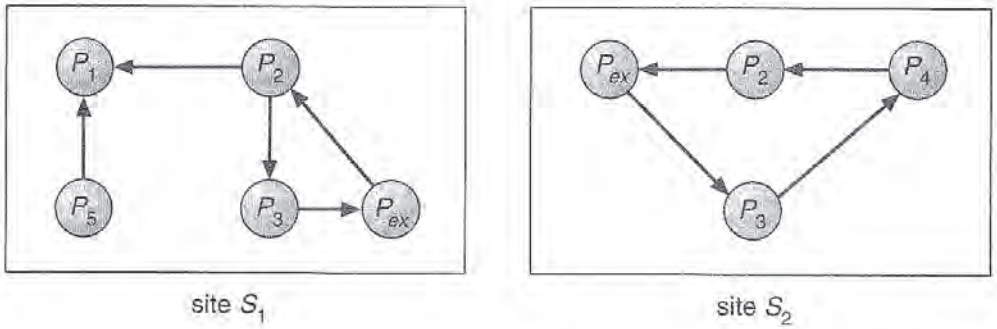


Figure 18.6 Augmented local wait-for graphs of Figure 18.3.

ascertain whether a deadlock does exist, we must invoke a distributed deadlock-detection algorithm.

Suppose that, at site S_i, the local wait-for graph contains a cycle involving node P_{ex}. This cycle must be of the form

$$P_{ex} \rightarrow P_{k_1} \rightarrow P_{k_2} \rightarrow \cdots \rightarrow P_{k_n} \rightarrow P_{ex},$$

which indicates that transaction P_{k_n} in site S_i is waiting to acquire a data item located in some other site — say, S_j. On discovering this cycle, site S_i sends to site S_j a deadlock-detection message containing information about that cycle.

When site S_j receives this deadlock-detection message, it updates its local wait-for graph with the new information. Then, it searches the newly constructed wait-for graph for a cycle not involving P_{ex}. If one exists, a deadlock is found and an appropriate recovery scheme is invoked. If a cycle involving P_{ex} is discovered, then S_j transmits a deadlock-detection message to the appropriate site — say, S_k. Site S_k, in return, repeats the procedure. Thus, after a finite number of rounds, either a deadlock is discovered, or the deadlock-detection computation halts.

To illustrate this procedure, we consider the local wait-for graphs of Figure 18.6. Suppose that site S_1 discovers the cycle

$$P_{ex} \rightarrow P_2 \rightarrow P_3 \rightarrow P_{ex}.$$

Since P_3 is waiting to acquire a data item in site S_2, a deadlock-detection message describing that cycle is transmitted from site S_1 to site S_2. When site S_2 receives this message, it updates its local wait-for graph, obtaining the wait-for graph of Figure 18.7. This graph contains the cycle

$$P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_2,$$

which does not include node P_{ex}. Therefore, the system is in a deadlock state and an appropriate recovery scheme must be invoked.

Note that the outcome would be the same if site S_2 discovered the cycle first in its local wait-for graph and sent the deadlock-detection message to site S_1. In the worst case, both sites will discover the cycle at about the same time, and two deadlock-detection messages will be sent: one by S_1 to S_2 and another by S_2 to S_1. This situation results in unnecessary message transfer and overhead in updating the two local wait-for graphs and searching for cycles in both graphs.

To reduce message traffic, we assign to each transaction P_i a unique identifier, which we denote by $ID(P_i)$. When site S_k discovers that its local wait-for graph contains a cycle involving node P_{ex} of the form

$$P_{ex} \rightarrow P_{K_1} \rightarrow P_{K_2} \rightarrow \cdots \rightarrow P_{K_n} \rightarrow P_{ex},$$

it sends a deadlock-detection message to another site only if

$$ID(P_{K_n}) < ID(P_{K_1}).$$

Otherwise, site S_k continues its normal execution, leaving the burden of initiating the deadlock-detection algorithm to some other site.

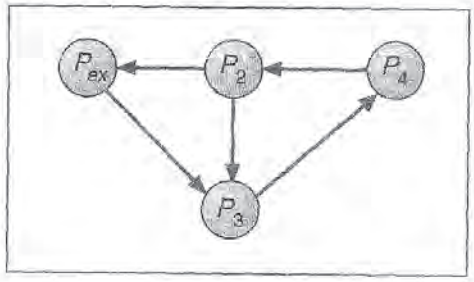
To illustrate this scheme, we consider again the wait-for graphs maintained at sites S_1 and S_2 of Figure 18.6. Suppose that

$$ID(P_1) < ID(P_2) < ID(P_3) < ID(P_4).$$

Let both sites discover these local cycles at about the same time. The cycle in site S_1 is of the form

$$P_{ex} \rightarrow P_2 \rightarrow P_3 \rightarrow P_{ex}.$$

Since $ID(P_3) > ID(P_2)$, site S_1 does not send a deadlock-detection message to site S_2.



site S_2

Figure 18.7 Augmented local wait-for graph in site S_2 of Figure 18.6.

The cycle in site S_2 is of the form

$$P_{ex} \rightarrow P_3 \rightarrow P_4 \rightarrow P_2 \rightarrow P_{ex}.$$

Since $ID(P_2) < ID(P_3)$, site S_2 does send a deadlock-detection message to site S_1, which, on receiving the message, updates its local wait-for graph. Site S_1 then searches for a cycle in the graph and discovers that the system is in a deadlock state.

18.6 ■ Election Algorithms

As we pointed out in Section 18.3, many distributed algorithms employ a coordinator process that performs functions needed by the other processes in the system. These functions include enforcing mutual exclusion, maintaining a global wait-for graph for deadlock detection, replacing a lost token, or controlling an input or output device in the system. If the coordinator process fails due to the failure of the site at which it resides, the system can continue execution only by restarting a new copy of the coordinator on some other site. The algorithms that determine where a new copy of the coordinator should be restarted are called *election* algorithms.

Election algorithms assume that a unique priority number is associated with each active process in the system. For ease of notation, we assume that the priority number of process P_i is i. To simplify our discussion, we assume a one-to-one correspondence between processes and sites, and thus refer to both as processes. The coordinator is always the process with the largest priority number. Hence, when a coordinator fails, the algorithm must elect that active process with the largest priority number. This number must be sent to each active process in the system. In addition, the algorithm must provide a mechanism for a recovered process to identify the current coordinator.

In this section, we present two interesting examples of election algorithms for two different configurations of distributed systems. The first algorithm is applicable to systems where every process can send a message to every other process in the system. The second algorithm is applicable to systems organized as a ring (logically or physically). Both algorithms require n^2 messages for an election, where n is the number of processes in the system. We assume that a process that has failed knows on recovery that it indeed has failed and thus takes appropriate actions to rejoin the set of active processes.

18.6.1 The Bully Algorithm

Suppose that process P_i sends a request that is not answered by the coordinator within a time interval T. In this situation, it is assumed that

the coordinator has failed, and P_i tries to elect itself as the new coordinator. This task is completed through the following algorithm.

Process P_i sends an election message to every process with a higher priority number. Process P_i then waits for a time interval T for an answer from any one of these processes.

If no response is received within time T, P_i assumes that all processes with numbers greater than i have failed, and elects itself the new coordinator. Process P_i restarts a new copy of the coordinator and sends a message to inform all active processes with priority numbers less than i that P_i is the new coordinator.

However, if an answer is received, P_i begins a time interval T', waiting to receive a message informing it that a process with a higher priority number has been elected. (Some other process is electing itself coordinator, and should report the results within time T'.) If no message is sent within T', then the process with a higher number is assumed to have failed, and process P_i should restart the algorithm.

If P_i is not the coordinator, then, at any time during execution, P_i may receive one of the following two messages from process P_j:

1. P_j is the new coordinator ($j > i$). Process P_i, in turn, records this information.
2. P_j started an election ($j < i$). Process P_i sends a response to P_j and begins its own election algorithm, provided that P_i has not already initiated such an election.

The process that completes its algorithm has the highest number and is elected as the coordinator. It has sent its number to all active processes with smaller numbers. After a failed process recovers, it immediately begins execution of the same algorithm. If there are no active processes with higher numbers, the recovered process forces all processes with lower numbers to let it become the coordinator process, even if there is a currently active coordinator with a lower number. For this reason, the algorithm is termed the *bully* algorithm.

Let us demonstrate the operation of the algorithm with a simple example of a system consisting of processes P_1 through P_4. The operations are as follows:

1. All processes are active; P_4 is the coordinator process.
2. P_1 and P_4 fail. P_2 determines P_4 has failed by sending a request that is not answered within time T. P_2 then begins its election algorithm by sending a request to P_3.
3. P_3 receives the request, responds to P_2, and begins its own algorithm by sending an election request to P_4.

4. P_2 receives P_3's response, and begins waiting for an interval T'.
5. P_4 does not respond within an interval T, so P_3 elects itself the new coordinator, and sends the number 3 to P_2 and P_1 (which P_1 does not receive, since it has failed).
6. Later, when P_1 recovers, it sends an election request to P_2, P_3, and P_4.
7. P_2 and P_3 respond to P_1 and begin their own election algorithms. P_3 will again be elected, using the same events as before.
8. Finally, P_4 recovers and notifies P_1, P_2, and P_3 that it is the current coordinator. (P_4 sends no election requests, since it is the process with the highest number in the system.)

18.6.2 Ring Algorithm

The *ring* algorithm assumes that the links are unidirectional, and that processes send their messages to their right neighbors. The main data structure used by the algorithm is the *active list*, a list that contains the priority numbers of all active processes in the system when the algorithm ends; each process maintains its own active list. The algorithm works as follows:

1. If process P_i detects a coordinator failure, it creates a new active list that is initially empty. It then sends a message *elect*(i) to its right neighbor, and adds the number i to its active list.
2. If P_i receives a message *elect*(j) from the process on the left, it must respond in one of three ways:
   a. If this is the first *elect* message it has seen or sent, P_i creates a new active list with the numbers i and j. It then sends the message *elect*(i), followed by the message *elect*(j).
   b. If $i \neq j$ (that is, the message received does not contain P_i's number), then P_i adds j to its active list and forwards the message to its right neighbor.
   c. If $i = j$ (that is, P_i receives the message *elect*(i)), then the active list for P_i now contains the numbers of all the active processes in the system. Process P_i can now determine the largest number in the active list to identify the new coordinator process.

This algorithm does not specify how a recovering process determines the number of the current coordinator process. One solution would be to require a recovering process to send an inquiry message. This message is forwarded around the ring to the current coordinator, which in turn sends a reply containing its number.

18.7 ■ Reaching Agreement

For a system to be reliable, we need a mechanism that allows a set of processes to agree on a common "value." There are several reasons why such an agreement may not take place. First, the communication medium may be faulty, resulting in lost or garbled messages. Second, the processes themselves may be faulty, resulting in unpredictable process behavior. The best we can hope for, in this case, is that processes fail in a clean way, stopping their execution without deviating from their normal execution pattern. In the worst case, processes may send garbled or incorrect messages to other processes, or even collaborate with other failed processes in an attempt to destroy the integrity of the system.

This problem has been expressed as the *Byzantine generals problem*. Several divisions of the Byzantine army, each commanded by its own general, surround an enemy camp. The Byzantine generals must reach a common agreement on whether or not to attack the enemy at dawn. It is crucial that all generals agree, since an attack by only some of the divisions would result in defeat. The various divisions are geographically dispersed and the generals can communicate with one another only via messengers who run from camp to camp. There are at least two major reasons why the generals may not be able to reach an agreement:

- Messengers may get caught by the enemy and thus may be unable to deliver their messages. This situation corresponds to unreliable communication in a computer system, and is discussed further in Section 18.7.1.
- Generals may be *traitors*, trying to prevent the *loyal* generals from reaching an agreement. This situation corresponds to faulty processes in a computer system, and is discussed further in Section 18.7.2.

18.7.1 Unreliable Communications

Let us assume that, if processes fail, they do so in a clean way, and that the communication medium is unreliable. Suppose that process P_i at site A, which has sent a message to process P_j at site B, needs to know whether P_j has received the message so that it can decide how to proceed with its computation. For example, P_i may decide to compute a function S if P_j has received its message, or to compute a function F if P_j has not received the message (because of some hardware failure).

To detect failures, we can use a *time-out* scheme similar to the one described in Section 16.4.1. When P_i sends out a message, it also specifies a time interval during which it is willing to wait for an acknowledgment message from P_j. When P_j receives the message, it immediately sends an acknowledgment to P_i. If P_i receives the acknowledgment message within

the specified time interval, it can safely conclude that P_j has received its message. If, however, a time-out occurs, then P_i needs to retransmit its message and to wait for an acknowledgment. This procedure continues until P_i either gets the acknowledgment message back, or is notified by the system that site B is down. In the first case, it will compute S; in the latter case, it will compute F. Note that, if these are the only two viable alternatives, P_i must wait until it has been notified that one of the situations has occurred.

Suppose now that P_j also needs to know that P_i has received its acknowledgment message, to decide on how to proceed with its computation. For example, P_j may want to compute S only if it is assured that P_i got its acknowledgment. In other words, P_i and P_j will compute S if and only if both have agreed on it. It turns out that, in the presence of failure, it is not possible to accomplish this task. More precisely, it is not possible in a distributed environment for processes P_i and P_j to agree completely on their respective states.

Let us prove this claim. Suppose that there exists a minimal sequence of message transfers such that, after the messages have been delivered, both processes agree to compute S. Let m' be the last message sent by P_i to P_j. Since P_i does not know whether its message will arrive at P_j (since the message may be lost due to a failure), P_i will execute S regardless of the outcome of the message delivery. Thus, m' could be removed from the sequence without affecting the decision procedure. Hence, the original sequence was not minimal, contradicting our assumption and showing that there is no sequence. The processes can never be sure that both will compute S.

18.7.2 Faulty Processes

Let us assume that the communication medium is reliable but that processes can fail in unpredictable ways. Consider a system of n processes, of which no more than m are faulty. Suppose that each process P_i has some private value of V_i. We wish to devise an algorithm that allows each nonfaulty process P_i to construct a vector $X_i = A_{i,1}, A_{i,2}, \ldots, A_{i,n})$ such that

1. If P_j is a nonfaulty process, then $A_{i,j} = V_j$.
2. If P_i and P_j are both nonfaulty processes, then $X_i = X_j$.

There are many solutions to this problem. These solutions share the following properties:

1. A correct algorithm can be devised only if $n \geq 3 \times m + 1$.
2. The worst-case delay for reaching agreement is proportionate to $m + 1$ message-passing delays.

3. The number of messages required for reaching agreement is large. No single process is trustworthy, so all processes must collect all information and make their own decisions.

Rather than presenting a general solution, which would be complicated, we present an algorithm for the simple case where $m = 1$ and $n = 4$. The algorithm requires two rounds of information exchange:

1. Each process sends its private value to the other three processes.
2. Each process sends the information it has obtained in the first round to all other processes.

A faulty process obviously may refuse to send messages. In this case, a nonfaulty process can choose an arbitrary value and pretend that that value was sent by that process.

Once these two rounds are completed, a nonfaulty process P_i can construct its vector $X_i = (A_{i,1}, A_{i,2}, A_{i,3}, A_{i,4})$ as follows:

1. $A_{i,i} = V_i$.
2. For $j \neq i$, if at least two of the three values reported for process P_j (in the two rounds of exchange) agree, then the majority value is used to set the value of $A_{i,j}$. Otherwise, a default value, say *nil*, is used to set the value of $A_{i,j}$.

18.8 ■ Summary

In a distributed system with no common memory and no common clock, it is sometimes impossible to determine the exact order in which two events occur. The happened-before relation is only a partial ordering of the events in distributed systems. Timestamps can be used to provide a consistent event ordering in a distributed system.

Mutual exclusion in a distributed environment can be implemented in a variety of ways. In a centralized approach, one of the processes in the system is chosen to coordinate the entry to the critical section. In the fully distributed approach, the decision making is distributed across the entire system. A distributed algorithm, which is applicable to ring-structured network, is the token-passing approach.

For atomicity to be ensured, all the sites in which a transaction T executed must agree on the final outcome of the execution. T either commits at all sites or aborts at all sites. To ensure this property, the transaction coordinator of T must execute a *commit* protocol. The most widely used commit protocol is the 2PC protocol.

The various concurrency-control schemes that can be used in a centralized system can be modified for use in a distributed environment. In the case of locking protocols, the only change that needs to be incorporated is in the way that the lock manager is implemented. In the case of timestamping and validation schemes, the only needed change is the development of a mechanism for generating unique *global* timestamps. The mechanism can either concatenate a local timestamp with the site identification or advance local clocks whenever a message arrives that has a larger timestamp.

The primary method for dealing with deadlocks in a distributed environment is deadlock detection. The main problem is deciding how to maintain the wait-for graph. Different methods for organizing the wait-for graph include a centralized approach and a fully distributed approach.

Some of the distributed algorithms require the use of a coordinator. If the coordinator fails owing to the failure of the site at which it resides, the system can continue execution only by restarting a new copy of the coordinator on some other site. It does so by maintaining a backup coordinator that is ready to assume responsibility if the coordinator fails. Another approach is to choose the new coordinator after the coordinator has failed. The algorithms that determine where a new copy of the coordinator should be restarted are called *election* algorithms. Two algorithms, the bully algorithm and a ring algorithm, can be used to elect a new coordinator in case of failures.

■ Exercises

18.1 Discuss the advantages and disadvantages of the two methods we presented for generating globally unique timestamps.

18.2 Your company is building a computer network, and you are asked to write an algorithm for achieving distributed mutual exclusion. Which scheme will you use? Explain your choice.

18.3 Why is deadlock detection much more expensive in a distributed environment than it is in a centralized environment?

18.4 Your company is building a computer network, and you are asked to develop a scheme for dealing with the deadlock problem.

a. Would you use a deadlock-detection scheme, or a deadlock-prevention scheme?

b. If you were to use a deadlock-prevention scheme, which one would you use? Explain your choice.

c. If you were to use a deadlock-detection scheme, which one would you use? Explain your choice.

18.5 Consider the following *hierarchical* deadlock-detection algorithm, in which the global wait-for graph is distributed over a number of different *controllers*, which are organized in a tree. Each nonleaf controller maintains a wait-for graph that contains relevant information from the graphs of the controllers in the subtree below it. In particular, let S_A, S_B, and S_C be controllers such that S_C is the lowest common ancestor of S_A and S_B (S_C must be unique, since we are dealing with a tree). Suppose that node T_i appears in the local wait-for graph of controllers S_A and S_B. Then, T_i must also appear in the local wait-for graph of

- Controller S_C
- Every controller in the path from S_C to S_A
- Every controller in the path from S_C to S_B

In addition, if T_i and T_j appear in the wait-for graph of controller S_D and there exists a path from T_i to T_j in the wait-for graph of one of the children of D, then an edge $T_i \rightarrow T_j$ must be in the wait-for graph of S_D.

Show that, if a cycle exists in any of the wait-for graphs, then the system is deadlocked.

18.6 Derive an election algorithm for bidirectional rings that is more efficient than the one presented in this chapter. How many messages are needed for n processes?

18.7 Consider a failure that occurs during two-phase commit for a transaction. For each possible failure, explain how two-phase commit ensures transaction atomicity despite the failure.

Bibliographic Notes

The distributed algorithm for extending the happened-before relation to a consistent total ordering of all the events in the system (Section 18.1) was developed by Lamport [1978a].

The first general algorithm for implementing mutual exclusion in a distributed environment was developed by Lamport [1978a] also. Lamport's scheme requires $3 \times (n - 1)$ messages per critical-section entry. Subsequently, Ricart and Agrawala [1981] proposed a distributed algorithm that requires only $2 \times (n - 1)$ messages. Their algorithm was presented in Section 18.2.2. A square root algorithm for distributed mutual exclusion was presented by Maekawa [1985]. The token-passing algorithm for ring-structured systems presented in Section 18.2.3 was developed by Le Lann [1977]. Discussions concerning mutual exclusion in computer networks

were presented by Carvalho and Roucairol [1983]. An efficient and fault-tolerant solution of distributed mutual exclusion was presented by Agrawal and El Abbadi [1991]. A simple taxonomy for distributed mutual exclusion algorithms was presented by Raynal [1991].

The issue of distributed synchronization was discussed by Reed and Kanodia [1979] (shared-memory environment), Lamport [1978a, 1978b], and Schneider [1982] (totally disjoint processes). A distributed solution to the dining-philosophers problem was presented by Chang [1980].

The 2PC protocol was developed by Lampson and Sturgis [1976] and Gray [1978]. Mohan and Lindsay [1983] discussed two modified versions of 2PC, called presume commit and presume abort, that reduce the overhead of 2PC by defining default assumption regarding the fate of transactions.

Papers dealing with the problems of implementing the transaction concept in a distributed database were presented by Gray [1981], Traiger et al. [1982], and Spector and Schwarz [1983]. Comprehensive discussions covering distributed concurrency control were offered by Bernstein et al. [1987].

Rosenkrantz et al. [1978] reported the timestamp distributed deadlock-prevention algorithm. The fully distributed deadlock-detection scheme presented in Section 18.5.2 was developed by Obermarck [1982]. The hierarchical deadlock-detection scheme of Exercise 18.3 appeared in Menasce and Muntz [1979]. A survey of deadlock detection in distributed systems is offered by Knapp [1987] and Singhal [1989].

The Byzantine generals problem was discussed by Lamport et al. [1982] and Pease et al. [1980]. The bully algorithm was presented by Garcia-Molina [1982]. The election algorithm for a ring-structured system was written by Le Lann [1977].