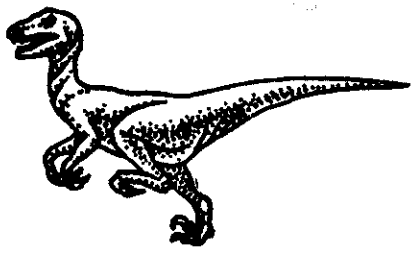
Distributed File Systems

CHAPTER



In the previous chapter, we discussed network construction and the low-level protocols needed for messages to be transferred between systems. Now we examine 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 11). The purpose of a DFS is to support the same kind of sharing when the files are physically dispersed among the sites of a distributed system.

In this chapter, we describe how a DFS can be designed and implemented. First, we discuss common concepts on which DFSs are based. Then, we illustrate our concepts by examining one influential DFS—the Andrew file system (AFS).

15.1 Background

As we noted in the preceding chapter, a distributed system is a collection of loosely coupled computers interconnected by a communication network. These computers can share physically dispersed files by using a distributed file system (DFS). In this chapter, we use the term *DFS* to mean distributed file systems in general, not the commercial Transarc DFS product. The latter is referenced as *Transarc DFS*. Also, NFS refers to NFS version 3, unless otherwise noted.

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 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 form its **client interface**. Sometimes a lower-level interface is defined for the actual cross-machine interaction; it is 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 clients' 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. Instead of a single centralized data repository, the system frequently has multiple and independent storage devices. As you will see in this text, the concrete configuration and implementation of a DFS may vary from system to system. In some configurations, servers run on dedicated machines; in others, 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 appear to its clients to be a conventional, centralized file system. The multiplicity and dispersion of its servers and storage devices should be made invisible. 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 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 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 to deliver the request to a server, as well as the time to get the response across the network back to the client. For each direction, in addition to the 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, these constituent storage spaces correspond to sets of files. A **component unit** is 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.

15.2 [illegible]

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 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 expanded 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 locations are hidden.

15.2.1 Naming Structures

We need to differentiate two related notions regarding name mappings in a DFS:

1. **Location transparency**. The name of a file does not reveal any hint of the file's physical storage location.
2. **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 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 AFS and a few experimental file systems support location independence and file mobility. AFS supports file mobility mainly for administrative purposes. A protocol provides migration of AFS component units to satisfy high-level user requests, without changing either the user-level names or the low-level names of the corresponding files.

A few aspects can further differentiate location independence and static location transparency:

- Divorce of data from location, as exhibited by location independence, provides a 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 configuration 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, clients can access files residing on remote server systems. 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 the client machines require no disks) and greater convenience (when an operating-system upgrade occurs, only the server needs to be modified). The disadvantages are the added complexity of the boot protocols and the performance loss resulting from the use of a network rather than a local disk.

The current trend is for clients to use both local disks and remote file servers. Operating systems and networking software are stored locally; file systems containing user data—and possibly applications—are stored on remote file systems. Some client systems may store commonly used applications, such as word processors and web browsers, on the local file system as well. Other, less commonly used applications may be **pushed** from the remote file server to the client on demand. The main reason for providing clients with local file systems rather than pure diskless systems is that disk drives are rapidly increasing in capacity and decreasing in cost, with new generations appearing every year or so. The same cannot be said for networks, which evolve every few years. Overall, systems are growing more quickly than are networks, so extra work is needed to limit network access to improve system throughput.

15.2.2 Naming Schemes

There are three main approaches to naming schemes in a DFS. In the simplest approach, a file is identified by some combination of its host name and local name, which guarantees a unique system-wide 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 DFS is structured as a collection of isolated component units, each of which is an entire conventional file system. In this first approach, component units remain 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 supported by many UNIX vendors. NFS provides a means to attach remote directories to local directories, thus giving the appearance of a coherent directory tree. Early NFS versions allowed only previously mounted remote directories to be accessed transparently. With the advent of the **automount** feature, mounts are done on demand, based on a table of mount points and file-structure names. Components are integrated to support transparent sharing, although this integration is limited and is not uniform, because each machine may attach different remote directories to its tree. The resulting structure is versatile.

We can achieve total integration of the component file systems by using the third approach. A single global name structure spans all the files in the system. Ideally, the composed file-system structure is isomorphic to the structure of a conventional file system. In practice, however, the many special files (for example, UNIX device files and machine-specific binary directories) make this goal difficult to attain.

To evaluate naming structures, we look at their **administrative complexity**. The most complex and most difficult-to-maintain structure is the NFS structure. Because any remote directory can be attached anywhere onto the local directory tree, the resulting hierarchy can be highly unstructured. If a server becomes unavailable, some arbitrary set of directories on different machines becomes unavailable. In addition, a separate accreditation mechanism controls which machine is allowed to attach which directory to its tree. Thus, a user might be able to access a remote directory tree on one client but be denied access on another client.

15.2.3 Implementation Techniques

Implementation of transparent naming requires a provision for the mapping of a file name to the associated location. To keep this mapping manageable, we must aggregate sets of files into component units and provide 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 replication, local caching, or both. As we noted, location independence means that the mapping changes over time; hence, replicating the mapping makes 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. The inevitable price is the need for a second level of mapping, which maps component units to locations and needs a simple yet consistent update mechanism. 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 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 AFS), or by using a timestamp as one part 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.

Aggregating files into component units and using lower-level location-independent file identifiers are techniques exemplified in AFS.

15.3 [illegible]

Consider a user who requests access to a remote file. The server storing the file has been located by the naming scheme, and now the actual data transfer must take place.

One way to achieve this transfer is through a **remote-service mechanism**, whereby 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 Chapter 3. A direct analogy exists between disk-access methods in conventional file systems and the remote-service method in a DFS: Using 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 discussion, we describe the implementation of caching in a DFS and contrast it with the basic remote-service paradigm.

15.3.1 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) keeps the cache size bounded. No direct correspondence exists between accesses and traffic to the server. Files are still identified with one master copy residing at the server machine, but copies (or 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 we discuss in Section 15.3.4. DFS caching could just as easily be called **network virtual memory**; it acts similarly to demand-paged virtual memory, except that the backing store usually is not a local disk but rather a remote server. NFS allows the swap space to be mounted remotely, so it actually can implement virtual memory over a network, notwithstanding the resulting performance penalty.

The granularity of the cached data in a DFS 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 11.6.2). AFS caches files in large chunks (64 KB). The other systems discussed in this chapter support caching of individual blocks driven by client demand. Increasing the caching unit increases the hit ratio, but it also increases the miss penalty, because each miss requires more data to be transferred. It increases the potential for consistency problems as well. Selecting the unit of caching involves considering parameters such as the network transfer unit and the RPC protocol service unit (if an RPC protocol is used). The network transfer unit (for Ethernet, a packet) is about 1.5 KB, so larger units of cached data need to be disassembled for delivery and reassembled on reception.

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

15.3.2 Cache Location

Where should the cached data be stored—on disk or in main memory? Disk caches have one clear advantage over main-memory caches: They are reliable. 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. Main-memory caches have several advantages of their own, however:

- 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.
- Technology is moving toward larger and less expensive memory. The achieved performance speedup is predicted to outweigh the advantages of disk caches.
- The server caches (used to speed up disk I/O) will be in main memory regardless of where user caches are located; if we use main-memory caches on the user machine, too, we can build a single caching mechanism for use by both servers and users.

Many remote-access implementations can be thought of as hybrids of caching and remote service. In NFS, for instance, the implementation is based on remote service but is augmented with client- and server-side memory caching for performance. Similarly, Sprite's implementation is based on caching; but under certain circumstances, a remote-service method is adopted. Thus, to evaluate the two methods, we must evaluate to what degree either method is emphasized.

The NFS protocol and most implementations do not provide disk caching. Recent Solaris implementations of NFS (Solaris 2.6 and beyond) include a client-side disk caching option, the **cachefs** file system. Once the NFS client reads blocks of a file from the server, it caches them in memory as well as on disk. If the memory copy is flushed, or even if the system reboots, the disk cache is referenced. If a needed block is neither in memory nor in the cachefs disk cache, an RPC is sent to the server to retrieve the block, and the block is written into the disk cache as well as stored in the memory cache for client use.

15.3.3 Cache-Update Policy

The policy used to write modified data blocks back to the server's master copy has a 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 in any cache. The advantage of a **write-through policy** is 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, so it causes poor write performance. Caching with write-through is equivalent to using remote service for write accesses and exploiting caching only for read accesses.

An alternative is the **delayed-write policy**, also known as **write-back caching**, where we 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 made to the cache, write accesses complete much more quickly. Second, data may be overwritten before they are written back, in which case only the last update needs to be written at all. Unfortunately, delayed-write schemes introduce reliability problems, since unwritten data are lost whenever a user machine crashes.

Variations of the delayed-write policy 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 most recent scan, just as UNIX scans its local cache. Sprite uses this policy with a 30-second interval. NFS uses the policy for file data, but once a write is issued to the server during a cache flush, the write must reach the server's disk before it is considered complete. NFS treats metadata (directory data and file-attribute data) differently. Any metadata changes are issued synchronously to the server. Thus, file-structure loss and directory-structure corruption are avoided when a client or the server crashes.

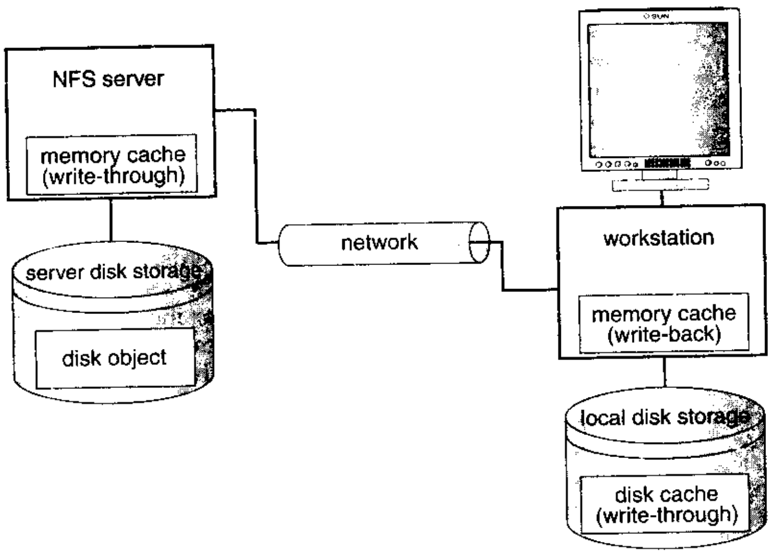

NFS server
memory cache (write-through)
server disk storage
disk object
network
workstation
memory cache (write-back)
local disk storage
disk cache (write-through)

Figure 15.1 Cachefs and its use of caching.

For NFS with cachefs, writes are also written to the local disk cache area when they are written to the server, to keep all copies consistent. Thus, NFS with cachefs improves performance over standard NFS on a read request with a cachefs cache hit but decreases performance for read or write requests with a cache miss. As with all caches, it is vital to have a high cache hit rate to gain performance. Cachefs and its use of write-through and write-back caching is shown in Figure 15.1.

Yet another variation on delayed write is to write data back to the server when the file is closed. This **write-on-close policy** is used in AFS. 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. For files that are open for long periods and are modified frequently, however, the performance advantages of this policy over delayed write with more frequent flushing are apparent.

15.3.4 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 verifying 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 checking is the crux of this approach and determines the resulting consistency semantics. It can range from a check before every access to a check only on first access to a file (on file open, basically). Every access coupled with a validity check is delayed, compared with an access served immediately by the cache. Alternatively, checks can be initiated at fixed time intervals. Depending on its frequency, the validity check can load both the network and the server.

2. **Server-initiated approach**. The server records, for each client, the files (or parts of files) that it caches. When the server detects a potential inconsistency, it must react. A potential for inconsistency occurs when two different clients in conflicting modes cache a file. If UNIX semantics (Section 10.5.3) is implemented, we can resolve the potential inconsistency by having the server play an active role. The server must be notified whenever a file is opened, and the intended mode (read or write) must be indicated for every open. The server can then act when it detects that file has been opened simultaneously in conflicting modes by disabling caching for that particular file. Actually, disabling caching results in switching to a remote-service mode of operation.

15.3.5 A Comparison of Caching and Remote Service

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

- When caching is used, the local cache can handle a substantial number of the remote accesses efficiently. 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, when the remote-service method is used, every remote access is handled across the network. The penalty in network traffic, server load, and performance is obvious.

- Total network overhead is lower for transmitting big chunks of data (as is done in caching) than for transmitting series of responses to specific requests (as in the remote-service method). Furthermore, disk-access routines on the server may be better optimized if it is known that requests will always be for large, contiguous segments of data rather than for random disk blocks.

- The cache-consistency problem is the major drawback of caching. When access patterns 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 the client, rather than in response to the specific needs of a file operation, the lower-level intermachine interface is different from the upper-level user interface. The remote-service paradigm, in contrast, is just an extension of the local file-system interface across the network. Thus, the intermachine interface mirrors the user interface.

15.4 Stateful Versus Stateless Service

There are two approaches for storing server-side information when a client accesses remote files: 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 how those blocks are used. In the former case, the service provided is *stateful;* in the latter case, it is *stateless.*

The typical scenario of a **stateful file service** is as follows: A client must perform an open() operation on a file before accessing that file. The server fetches 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 that are no longer active. The key point regarding fault tolerance in a stateful service approach is that the server keeps main-memory information about its clients. AFS is a stateful file service.

A **stateless file service** avoids 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 through open() and close() operations. They are totally redundant, since each file operation stands on its own and is not considered part of a session. A client process would open a file, and that open would not result in the sending of a remote message. Reads and writes would take place as remote messages (or cache lookups). The final close by the client would again result in only a local operation. NFS is a stateless file service.

The advantage of a stateful over a stateless 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 knows whether a file is open for sequential access and can 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 distinction between stateful and stateless service becomes more evident when we consider the effects of a crash that occurs 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 computer 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, system-wide, 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 deleting 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 must store the current offset within a file. This requirement is why NFS, which employs a stateless service, does not use file descriptors and does include an explicit offset in every access.

Replication of files on different machines in a distributed file system is a useful redundancy for improving availability. Multimachine replication can benefit performance too: 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. Mapping a replicated file name to a particular replica is the task of the naming scheme. The existence of replicas should be invisible to higher levels. At lower levels, however, the replicas must be distinguished from one another by different lower-level names. Another transparency requirement 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, we may want 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 updating. 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. In this 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) circumstances of catastrophic failures for the sake of guaranteed progress. Locus, for example, employs replication extensively and sacrifices consistency in the case of network partition for the sake of availability of files for read and write accesses.

Ibis uses a variation of the primary-copy approach. The domain of the name mapping is a pair <*primary-replica-identifier, local-replica-identifier*>. If no local replica exists, a special value is used. 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, an automatic replication-control policy similar to whole-file caching. Under demand replication, reading of a nonlocal replica causes it to be cached locally, thereby generating a new nonprimary replica. Updates are performed only on the primary copy and cause all other replicas to be invalidated through the sending of 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.

15.5 An Example: AFS

Andrew is a distributed computing environment designed and implemented at Carnegie Mellon University. The Andrew file system (AFS) constitutes the underlying information-sharing mechanism among clients of the environment. The Transarc Corporation took over development of AFS, then was purchased by IBM. IBM has since produced several commercial implementations of AFS. AFS was subsequently chosen as the DFS for an industry coalition; the result was **Transarc DFS**, part of the distributed computing environment (DCE) from the OSF organization.

In 2000, IBM's Transarc Lab announced that AFS would be an open-source product (termed OpenAFS) available under the IBM public license and Transarc DFS was canceled as a commercial product. OpenAFS is available under most commercial versions of UNIX as well as Linux and Microsoft Windows systems. Many UNIX vendors, as well as Microsoft, support the DCE system and its DFS, which is based on AFS, and work is ongoing to make DCE a cross-platform, universally accepted DFS. As AFS and Transarc DFS are very similar, we describe AFS throughout this section, unless Transarc DFS is named specifically.

AFS seeks to solve many of the problems of the simpler DFSs, such as NFS, and is arguably the most feature-rich nonexperimental DFS. It features a uniform name space, location-independent file sharing, client-side caching with cache consistency, and secure authentication via Kerberos. It also includes server-side caching in the form of replicas, with high availability through automatic switchover to a replica if the source server is unavailable. One of the most formidable attributes of AFS is scalability: The Andrew system is targeted to span over 5,000 workstations. Between AFS and Transarc DFS, there are hundreds of implementations worldwide.

15.5.1 Overview

AFS distinguishes between *client machines* (sometimes referred to as *workstations*) and dedicated *server machines*. Servers and clients originally ran only 4.2 BSD UNIX, but AFS has been ported to many operating systems. The clients and servers are interconnected by a network of LANs or WANs.

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 each is required to have a local disk where it stores its 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 each cluster 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 is also based on considerations of scale. The basic heuristic is 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 (64 KB). 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 some additional issues concerning the design of AFS, which we briefly discuss here:

- **Client mobility**. Clients are able to access any file in the shared name space from any workstation. A client may notice some initial performance degradation due to the caching of files when accessing files from a workstation other than the usual one.

Security. The Vice interface is considered 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 replicated at each server.

Protection. AFS provides **access lists** for protecting directories and the regular UNIX bits for file protection. The access list may contain information about those users allowed to access a directory, as well as information about those users *not* allowed to access it. Thus, it is simple to specify that everyone except, say, Jim can access a directory. AFS 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 systems. So that heterogeneity is facilitated, some files in the local */bin* directory are symbolic links pointing to machine-specific executable files residing in Vice.

15.5.2 The Shared Name Space

AFS's shared name space is made up of component units called **volumes**. The 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 AFS 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. The aggregation of files into volumes 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 can still 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.

15.5.3 File Operations and Consistency Semantics

The fundamental architectural principle in AFS 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. Reading and writing files do not cause remote interaction (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 to record this caching. We say that the client has a callback on that file. The server notifies the client before allowing another client to modify the file. 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's intervention on the cached copy. Venus regains control when the file is closed. If the file has been modified locally, it updates the file on the appropriate server. Thus, the only occasions on 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, AFS 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 it 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. If the callback state maintained by Venus gets out of sync with the corresponding state maintained by the servers, some inconsistency may result.

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. Venus does lookups on the fetched directories locally, using fids. No requests are forwarded 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 exception to the caching policy is a modification to a directory that is made directly on the server responsible for that directory for reasons of integrity. The Vice interface has well-defined operations for such purposes. Venus reflects the changes in its cached copy to avoid re-fetching the directory.

15.5.4 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 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 AFS 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. DFS uses its own journaling file system to improve performance and reliability over UFS.

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 non-preemptible scheduling to service many client requests concurrently. The RPC package is integrated with the lightweight-process package, thereby allowing the file server to concurrently make or service one RPC per lightweight process. The RPC package is built on top of a low-level datagram abstraction. Whole-file transfer is implemented as a side effect of the RPC calls. One RPC connection exists 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 negative side, a crash of a single server process has the disastrous effect of paralyzing this particular server.

15.6

A DFS is a file-service system whose clients, servers, and storage devices are dispersed among the 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 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 the client's environment to the site where the 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 system-wide name. Another approach, popularized by NFS, provides a 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 the 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 the client accesses, or it simply provides blocks as the client requests them 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.

AFS is a feature-rich DFS characterized by location independence and location transparency. It also imposes significant consistency semantics. Caching and replication are used to improve performance.

Exercises

15.1 What are the benefits of a DFS compared with a file system in a centralized system?

15.2 Which of the example DFSs discussed in this chapter would handle a large, multiclient database application most efficiently? Explain your answer.

15.3 Under what circumstances would a client prefer a location-transparent DFS? Under what circumstances would she prefer a location-independent DFS? Discuss the reasons for these preferences.

15.4 What aspects of a distributed system would you select for a system running on a totally reliable network?

15.5 Compare and contrast the techniques of caching disk blocks locally, on a client system, and remotely, on a server.

15.6 AFS is designed to support a large number of clients. Discuss three techniques used to make AFS a scalable system.

15.7 What are the benefits of mapping objects into virtual memory, as Apollo Domain does? What are the drawbacks?

15.8 Describe some of the fundamental differences between AFS and NFS (see Chapter 11).

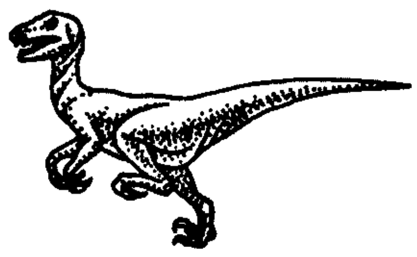
Bibliographical Notes

Discussions concerning consistency and recovery control for replicated files were offered by Davcev and Burkhard [1985]. Management of replicated files in a UNIX environment was covered by Brereton [1986] and Purdin et al. [1987]. Wah [1984] discussed the issue of file placement on distributed computer systems. A detailed survey of mainly centralized file servers was given in Svobodova [1984].

Sun's network file system (NFS) was presented by Callaghan [2000] and Sandberg et al. [1985]. The AFS system was discussed by Morris et al. [1986], Howard et al. [1988], and Satyanarayanan [1990]. Information about OpenAFS is available from http://www.openafs.org

Many different and interesting DFSs were not covered in detail in this text, including UNIX United, Sprite, and Locus. 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]. Distributed file systems for mobile storage devices were discussed in Kistler and Satyanarayanan [1992] and Sobti et al. [2004]. Considerable research has also been performed on cluster-based distributed file systems (Anderson et al. [1995], Lee and Thekkath [1996], Thekkath et al. [1997], and Anderson et al. [2000]). Distributed storage systems for large-scale, wide-area settings were presented in Dabek et al. [2001] and Kubiatowicz et al. [2000].

CHAPTER



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 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 centralized synchronization mechanisms can be extended to a distributed environment. We also discuss methods for handling deadlocks in a distributed system.

16.1

In a centralized system, we can always determine the order in which two events occurred, since the system has a single common memory and clock. Many applications may require us 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. A distributed system, however, has 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.

16.1.1 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):

> If A and B are events in the same process, and A was executed before B, then A $\rightarrow$ B.

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.

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.

A space–time diagram, such as that in Figure 16.1, can best illustrate the definitions of concurrency and *happened-before*. 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. Events are concurrent if and only if no path exists between them.

For example, these are some of the events related by the *happened-before* relation in Figure 16.1:

$p_1 \rightarrow q_2$
$r_0 \rightarrow q_4$
$q_3 \rightarrow r_4$
$p_1 \rightarrow q_4$ (since $p_1 \rightarrow q_2$ and $q_2 \rightarrow q_4$)

These are some of the concurrent events in the system:

q_0 and p_2
r_0 and q_3
r_0 and p_3
q_3 and p_3

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

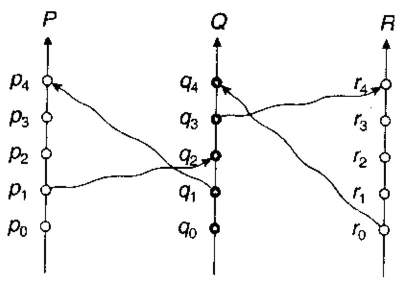


Figure 16.1 Relative time for three concurrent processes.

happened first. It is important only that any processes that care about the order of two concurrent events agree on some order.

16.1.2 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 neither of these is available in a distributed system, 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 $\rightarrow$ B, then the timestamp of A is less than the timestamp of B. (Below, we will see that the converse need not 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 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 $LC_i(\text{A}) < LC_i(\text{B})$. The timestamp for an event is the value of the logical clock for that event. 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, 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 $LC_1(\text{A}) = 200$, and P_2 receives the message (event B) with $LC_2(\text{B}) = 195$ (because the processor for P_2 is slower than the processor for P_1, its logical clock ticks more slowly). This situation violates our requirement, since A $\rightarrow$ 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 so 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 so that $LC_2(B) = 201$.

Finally, to realize a total ordering, we need only 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 further discussed in Section 16.4.2.

16.2 [illegible]

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 a one-to-one mapping exists between processes and processors (that is, each process has its own processor).

16.2.1 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 (FCFS) 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 16.6, we describe some 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 can resume.

16.2.2 Fully Distributed Approach

If we want to distribute the decision making across the entire system, then the solution is far more complicated. One approach, described next, uses an algorithm based on the event-ordering scheme described in Section 16.1.

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 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:

If process P_i is in its critical section, then it defers its reply to P_j.

If process P_i does *not* want to enter its critical section, then it sends a reply immediately to P_j.

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 of the incoming request made by process P_j. If its own *request* timestamp is greater than that of the incoming request, 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 FCFS 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 16.1. 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* message 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 from 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.

Because this scheme requires the participation of all the processes in the system, however, it has three undesirable consequences:

- 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 Bibliographical Notes for more details.

- If one process fails, then the entire scheme collapses. We can resolve this difficulty by continuously monitoring the state of all 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.

Processes that have not entered their critical section must pause frequently to assure other processes that they intend to enter the critical section.

Because of these difficulties, this protocol is best suited for small, stable sets of cooperating processes.

16.2.3 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, 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 need not 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.

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. In Section 16.6, we present an election algorithm; others are possible. The development of an algorithm for reconstructing the ring is left to you in Exercise 16.7.

16.3

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, ensuring the atomicity of a transaction becomes much more complicated than 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 the sites, may result in erroneous computations.

Ensuring that the execution of transactions in the distributed system preserves atomicity is the function of the **transaction coordinator**. Each site has its own local transaction coordinator, which 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 transactions being committed at all sites or aborted at all sites

We assume that each local site maintains a log for recovery purposes.

16.3.1 The Two-Phase Commit Protocol

For atomicity to be ensured, all the sites in which a transaction T has 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 next.

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 the sites at which T has executed. On receiving the 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 has received responses to the *prepare (T)* message from all the sites, or when a pre-specified 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 has 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 has 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. A site can make such a promise only when 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 with *abort (T)*. Note that the coordinator site S_i can decide unilaterally to abort T, as it is one of the sites at which T has executed. 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 has received the *acknowledge (T)* message from all the sites, it adds the record <complete T> to the log.

16.3.2 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, and a decision either to commit or to abort T may have to be postponed until C_i recovers.

16.3.2.1 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. How will S_k deal with T? We consider each of the possible alternatives:

- 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 out 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 (T)* message to the other sites. It does so until a site recovers that contains the needed information. 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 means that it could not have sent such a response, by our algorithm, C_i must have aborted T. Hence, S_k must execute undo(T).

16.3.2.2 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 these sites must 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. Rather than wait for C_i to recover, it is preferable to abort T in this case.
- If none of the preceding cases holds, then all the 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, if so, what that decision is—until the coordinator recovers. Thus, the active sites must wait for C_i to recover. As long as 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 hours or days may pass 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 amount 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.

16.3.2.3 Failure of the Network

When a link fails, the messages in the process of being routed through the link do not arrive at their destinations intact. From the viewpoint of the sites connected throughout that link, the other sites appear to 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.

16.4

We move next to the issue of concurrency control. In this section, we show how certain of the concurrency-control schemes discussed in Chapter 6 can be modified for use in a distributed environment.

The transaction manager of a distributed database system manages the execution of those transactions (or subtransactions) that access data stored in a local site. Each such transaction may be either a local transaction (that is, a transaction that executes only 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.

16.4.1 Locking Protocols

The two-phase locking protocols described in Chapter 6 can be used in a distributed environment. The only change needed is in the way the lock manager is implemented. Here, we present several possible schemes. The first deals with the case where no data replication is allowed. The others apply to the more general case where data can be replicated in several sites. As in Chapter 6, we assume the existence of the **shared** and **exclusive lock modes**.

16.4.1.1 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 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.

This 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; these modifications are discussed in Section 16.5.

16.4.1.2 Single-Coordinator Approach

Several concurrency-control schemes can be used in systems that allow data replication. 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; and at that 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` operation, 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, and the lock managers reside in different sites. 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 a single site.

16.4.1.3 Majority Protocol

The majority protocol is a modification of the nonreplicated data scheme presented earlier. The system maintains a lock manager at each site. Each manager controls the locks for all the data or replicas of data stored at that site. When a transaction wishes to lock a data item Q that 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 *ch18/18*.

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 (Section 16.5). In addition, a deadlock can 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.

16.4.1.4 Biased Protocol

The biased protocol is similar to 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 *ch18/18.*
- **Exclusive locks**. When a transaction needs to lock data item Q, it requests a lock on Q from the lock manager at each site containing a replica of *ch18/18.*

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 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.

16.4.1.5 Primary Copy

Yet another alternative is to 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 *ch18/18.* As before, the response to the request is delayed until the request can be granted.

This scheme enables us to handle concurrency control for replicated data in much the same way as for unreplicated data. Implementation of the method is simple. However, if the primary site of Q fails, Q is inaccessible even though other sites containing a replica may be accessible.

16.4.2 Timestamping

The principal idea behind the timestamping scheme discussed in Section 6.9 is that each transaction is given a *unique* timestamp, which is used to decide the serialization order. Our first task, then, in generalizing the centralized scheme to a distributed scheme is to develop a method for generating unique timestamps. Our previous protocols can then be applied directly to the nonreplicated environment.

16.4.2.1 Generation of Unique Timestamps

Two primary methods are used to generate unique timestamps; one is centralized, and one is 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 local unique timestamp using either a logical counter or the local clock. The global unique timestamp is obtained by concatenation of the local unique timestamp with the site identifier, which must be unique (Figure 16.2). 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 16.1.2 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 those of other sites. Therefore, all timestamps generated by the fast site will be larger than those generated by other sites. A mechanism is needed 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 local timestamp (see Section 16.1.2). 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.

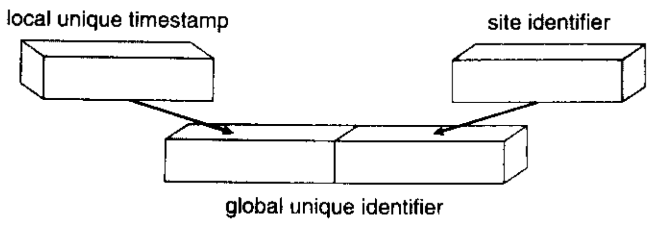


Figure 16.2 Generation of unique timestamps.

16.4.2.2 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 16.3 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 $\text{TS}(T_j) < \text{TS}(T_i)$. Similarly, a `write`($x$) operation by T_i must be delayed if there exists a transaction T_j that will perform either a `read`(x) or a `write`(x) operation and $\text{TS}(T_j) < \text{TS}(T_i)$. Various methods are available for ensuring this property. One such method, called the **conservative timestamp-ordering scheme**, requires each site to maintain a read queue and a write queue consisting of all the `read` and `write` requests 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. Again, we leave the development of the algorithm to you.

16.5

The deadlock-prevention, deadlock-avoidance, and deadlock-detection algorithms presented in Chapter 7 can be extended so that they can be used in a distributed system. In this section, we describe several of these distributed algorithms.

16.5.1 Deadlock Prevention and Avoidance

The deadlock-prevention and deadlock-avoidance algorithms presented in Chapter 7 can 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 channelled through the banker.

The global resource-ordering deadlock-prevention scheme is simple to implement in a distributed environment and requires little overhead. The banker's algorithm can also be implemented easily, but it may require too much overhead. The banker may become a bottleneck, since the number o 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.

We turn next to a new deadlock-prevention scheme based on a timestamp-ordering approach with resource preemption. Although this approach can handle any deadlock situation that may arise in a distributed system, 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 priorities 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 approach. 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 *wounded* 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 has 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 has requested a resource it holds. When P_i is restarted and requests the resource now being held by P_j, P_i waits. Thus, fewer rollbacks occur in the wound–wait scheme.

The major problem with both schemes is that unnecessary rollbacks may occur.

16.5.2 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 illustrate this problem by describing several common techniques to deal with this issue. These schemes require each site to keep a *local* wait-for graph. The nodes of the graph correspond to all the processes (local as well as nonlocal) currently holding or requesting any of the resources local to that site. For example, in Figure 16.3 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 S_1 needs a resource held by process P_j in site S_2, a request message is sent by P_i to site S_2. The edge $P_i \rightarrow P_j$ is then inserted in the local wait-for graph of site S_2.

Clearly, if any local wait-for graph has a cycle, deadlock has occurred. The fact that we find no cycles in any of the local wait-for graphs does not mean that there are no deadlocks, however. To illustrate this problem, we consider the system depicted in Figure 16.3. 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 (Figure 16.4) that we obtain by taking the union of the two wait-for graphs of Figure 16.3 does indeed contain a cycle, implying that the system is in a deadlocked state.

A number of methods are available to organize the wait-for graph in a distributed system. We describe several common schemes here.

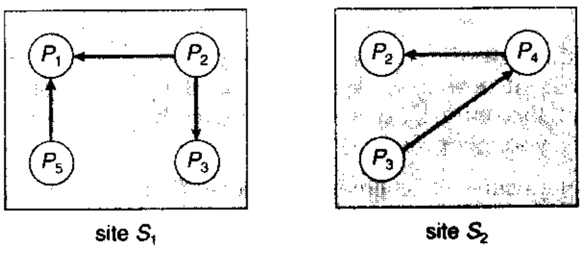


Figure 16.3 Two local wait-for graphs.

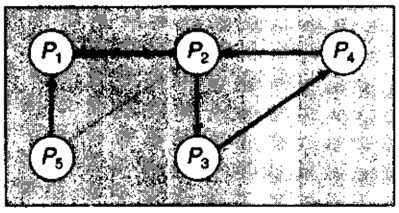


Figure 16.4 Global wait-for graph for Figure 16.3.

16.5.2.1 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. The constructed graph must be generated so that, whenever the detection algorithm is invoked, the reported results are correct. By *correct* we mean the following:

- If a deadlock exists, then it is reported properly.
- If a deadlock is reported, then the system is indeed in a deadlocked state.

As we shall show, it is not easy to construct such correct algorithms.

The wait-for graph may be constructed at three different points in time:

1. Whenever a new edge is inserted in or removed from one of the local wait-for graphs
2. Periodically, when a number of changes have occurred in a wait-for graph
3. Whenever the deadlock-detection coordinator needs to invoke the cycle-detection algorithm

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.

Let us consider option 1. Whenever an edge is either inserted in or removed from 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 (option 2), 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 16.4. When site S_2 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 S_1 deletes the edge $P_5 \rightarrow P_1$

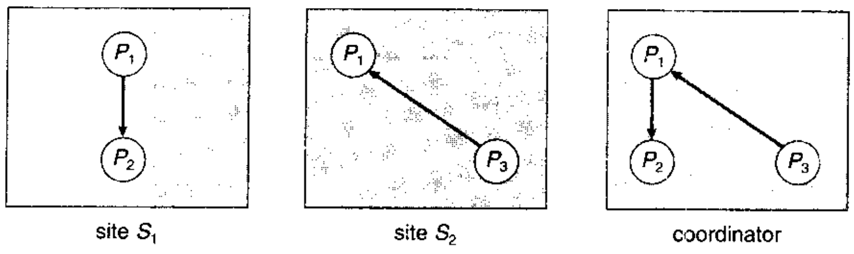


Figure 16.5 Local and global wait-for graphs.

because P_1 has released a resource that was requested by P_5, an appropriate message is sent to the coordinator.

Note that no matter which option is used, unnecessary rollbacks may occur, as a result of two situations:

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 16.5. Suppose that P_2 releases the resource it is holding in site S_1, resulting in the deletion of the edge $P_1 \rightarrow P_2$ in site S_1. Process P_2 then requests a resource held by P_3 at site S_2, resulting in the addition of the edge $P_2 \rightarrow P_3$ in site S_2. If the *insert* $P_2 \rightarrow P_3$ message from site S_2 arrives before the *delete* $P_1 \rightarrow P_2$ message from site S_1, 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 has been aborted for reasons unrelated to the deadlock (as when a process has exceeded its allocated time). For example, suppose that site S_1 in Figure 16.3 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.

Let us now consider a centralized deadlock-detection algorithm using option 3 that 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 (or timestamps). When process P_i, at site S_1, requests a resource from P_j, at site S_2, 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 graph of S_1. This edge is inserted in the local wait-for graph of site S_2 only if site S_2 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 as follows:

The controller sends an initiating message to each site in the system.

On receiving this message, a site sends its local wait-for graph to the coordinator. 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.

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* there is an edge $P_i \rightarrow P_j$ in one of the wait-for graphs *or* an edge $P_i \rightarrow P_j$ with some label *TS* in more than one wait-for graph.

If the constructed graph contains a cycle, then the system is in a deadlocked state. If the constructed graph does not contain a cycle, then the system was not in a deadlocked state when the detection algorithm was invoked as result of the initiating messages sent by the coordinator (in step 1).

16.5.2.2 Fully Distributed Approach

In the **fully distributed deadlock-detection algorithm**, all controllers share equally the responsibility for detecting deadlock. 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 a process at another site is waiting to acquire a resource currently being held by P_j in this local site.

To illustrate this situation, we consider again the two local wait-for graphs of Figure 16.3. The addition of the node P_{ex} in both graphs results in the local wait-for graphs shown in Figure 16.6.

If a local wait-for graph contains a cycle that does not involve node P_{ex}, then the system is in a deadlocked state. If, however, a local graph contains a cycle involving P_{ex}, then this implies the *possibility* of a deadlock.

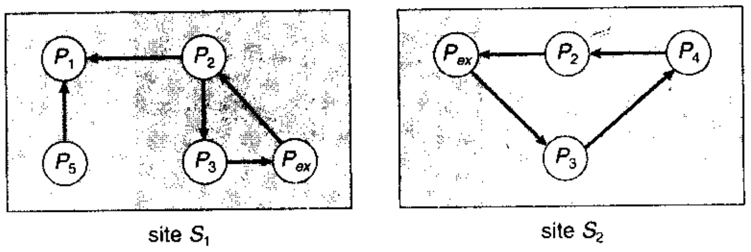


Figure 16.6 Augmented local wait-for graphs of Figure 16.3.

To 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

$$P_{ex} \rightarrow P_{k_1} \rightarrow P_{k_2} \rightarrow \ldots \rightarrow P_{k_n} \rightarrow P_{ex},$$

which indicates that process 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 16.6. 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 16.7. 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 deadlocked 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.

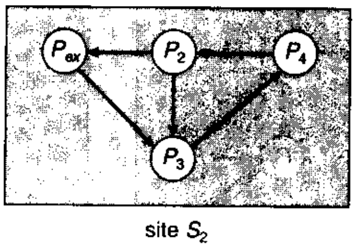


Figure 16.7 Augmented local wait-for graph in site S_2 of Figure 16.6.

To reduce message traffic, we assign to each process P_i a unique identifier, which we denote $ID(P_i)$. When site S_k discovers that its local wait-for graph contains a cycle involving node P_{ex} of the form

$$P_{ex} \rightarrow P_{K_1} \rightarrow P_{K_2} \rightarrow \ldots \rightarrow P_{K_n} \rightarrow P_{ex},$$

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 16.6. 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.

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 deadlocked state.

16.6 Election Algorithms

As we pointed out in Section 16.3, 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, and 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 examples of election algorithms for two different configurations of distributed systems. The first algorithm applies to systems where every process can send a message to every other process in the system. The second algorithm applies 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 has indeed failed and thus takes appropriate actions to rejoin the set of active processes.

16.6.1 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. (That is, 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:

- P_j is the new coordinator ($j > i$). Process P_i, in turn, records this information.
- P_j has 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**.

We can 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:

All processes are active; P_4 is the coordinator process.

P_1 and P_4 fail. P_2 determines that 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.

P_3 receives the request, responds to P_2, and begins its own algorithm by sending an election request to P_4.

P_2 receives P_3's response and begins waiting for an interval T'.

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. (P_1 does not receive the number, since it has failed.)

Later, when P_1 recovers, it sends an election request to P_2, P_3, and P_4.

P_2 and P_3 respond to P_1 and begin their own election algorithms. P_3 will again be elected, through the same events as before.

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.)

16.6.2 The Ring Algorithm

The **ring algorithm** assumes that the links are unidirectional and that each process sends its messages to the neighbor on the right. 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:

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.

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 requires 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.

16.7 [illegible]

For a system to be reliable, we need a mechanism that allows a set of processes to agree on a common *value*. Such an agreement may not take place, for several reasons. 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.

The **Byzantine generals problem** provides an analogy for this situation. Several divisions of the Byzantine army, each commanded by its own general, surround an enemy camp. The Byzantine generals must reach 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. The generals may not be able to reach agreement for at least two major reasons:

- 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 16.7.1.
- 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 16.7.2.

16.7.1 Unreliable Communications

Let us first 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 S_1, which has sent a message to process P_j at site S_2, 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 *foo* if P_j has received its message or to compute a function *boo* 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 14.7.1. 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 wait for an acknowledgment. This procedure continues until P_i either gets the acknowledgment message back or is notified by the system that site S_2 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, so that it can decide how to proceed with its computation. For example, P_j may want to compute *foo* only if it is assured that P_i got its acknowledgment. In other words, P_i and P_j will compute *foo* 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.

To prove this claim, let us suppose that a minimal sequence of message transfers exists such that, after the messages have been delivered, both processes agree to compute *foo*. 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 *foo* 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 *foo*.

16.7.2 Faulty Processes

Now 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 $X_i = (A_{i,1}, A_{i,2}, ..., A_{i,n})$ such that the following conditions exist:

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, and they 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:

- Each process sends its private value to the other three processes.
- 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 the value was sent by the faulty 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:

- $A_{i,i} = V_i$.
- 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 $A_{i,j}$.

16.8 [illegible]

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 a distributed system. Timestamps can be used to provide a consistent event ordering.

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 networks, is the token-passing approach.

For atomicity to be ensured, all the sites in which a transaction T has 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, we need only change the way the lock manager is implemented. In the case of timestamping and validation schemes, the only change needed 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. Methods for organizing the wait-for graph include a centralized approach and a fully distributed approach.

Some distributed algorithms require the use of a coordinator. If the coordinator fails because of 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 can do 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 the ring algorithm, can be used to elect a new coordinator in case of failures.

[illegible]

16.1 Discuss the advantages and disadvantages of the two methods we presented for generating globally unique timestamps.

16.2 The logical clock timestamp scheme presented in this chapter provides the following guarantee: If event A happens before event B, then the timestamp of A is less than the timestamp of B. Note, however, that one cannot order two events based only on their timestamps. The fact that an event C has a timestamp that is less than the timestamp of event D does not necessarily mean that event C happened before event D; C and D could be concurrent events in the system. Discuss ways in which the logical clock timestamp scheme could be extended to distinguish concurrent events from events that can be ordered by the *happens-before* relationship.

16.3 Why is deadlock detection much more expensive in a distributed environment than in a centralized environment?

16.4 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.

16.5 Consider the centralized and the fully distributed approaches to deadlock detection. Compare the two algorithms in terms of message complexity.

16.6 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 non-leaf 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 S_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.

16.7 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?

16.8 Consider a failure that occurs during 2PC for a transaction. For each possible failure, explain how 2PC ensures transaction atomicity despite the failure.

The distributed algorithm for extending the *happened-before* relation to a consistent total ordering of all the events in the system was developed by Lamport [1978b]. Further discussions of using logical time to characterize the behavior of distributed systems can be found in Fidge [1991], Raynal and Singhal [1996], Babaoglu and Marzullo [1993], Schwarz and Mattern [1994], and Mattern [1988].

The first general algorithm for implementing mutual exclusion in a distributed environment was also developed by Lamport [1978b]. 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 is presented in Section 16.2.2. A square-root algorithm for distributed mutual exclusion was described by Maekawa [1985]. The token-passing algorithm for ring-structured systems presented in Section 16.2.3 was developed by Lann [1977]. Carvalho and Roucairol [1983] discussed mutual exclusion in computer networks, and Agrawal and Abbadi [1991] described an efficient and fault-tolerant solution of distributed mutual exclusion. 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 [1978b], Lamport [1978a], 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 assumptions 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 of 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 16.5.2 was developed by Obermarck [1982]. The hierarchical deadlock-detection scheme of Exercise 16.3 appeared in Menasce and Muntz [1979]. Knapp [1987] and Singhal [1989] offered surveys of deadlock detection in distributed systems. Deadlocks can also be detected by taking global snapshots of a distributed system, as discussed in Chandy and Lamport [1985].

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], and the election algorithm for a ring-structured system was written by Lann [1977].