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

5.1. DISTRIBUTED FILE SYSTEM DESIGN

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

5.1.1. The File Service Interface

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

On mainframes, however, many types of files exist, each with different properties. A file can be structured as a sequence of records, for example, with operating system calls to read or write a particular record. The record can usually be specified by giving either its record number (i.e., position within the file) or the value of some field. In the latter case, the operating system either maintains the file as a B-tree or other suitable data structure, or uses hash tables to locate records quickly. Since most distributed systems are intended for UNIX or MS-DOS environments, most file servers support the notion of a file as a sequence of bytes rather than as a sequence of keyed records.

A files can have **attributes**, which are pieces of information about the file but which are not part of the file itself. Typical attributes are the owner, size, creation date, and access permissions. The file service usually provides primitives to read and write some of the attributes. For example, it may be possible to change the access permissions but not the size (other than by appending data to the file). In a few advanced systems, it may be possible to create and manipulate user-defined attributes in addition to the standard ones.

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

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

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

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

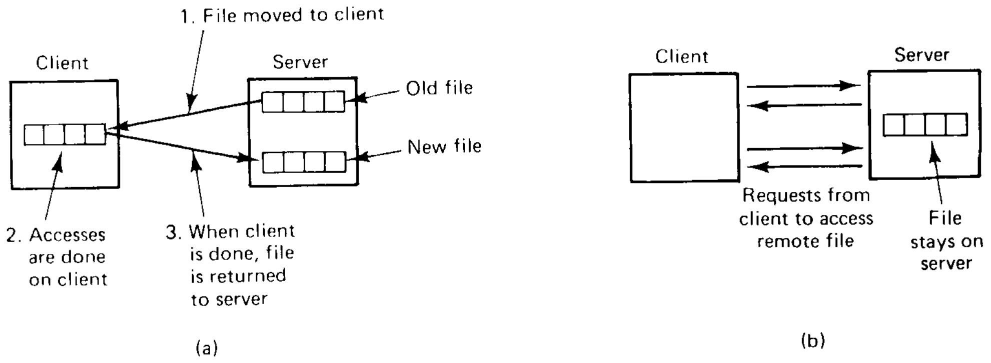
1. File moved to client
Client
Server
Old file
New file
2. Accesses are done on client
3. When client is done, file is returned to server
(a)
Client
Server
Requests from client to access remote file
File stays on server
(b)

Fig. 5-1. (a) The upload/download model. (b) The remote access model.

The advantage of the upload/download model is its conceptual simplicity. Application programs fetch the files they need, then use them locally. Any modified files or newly created files are written back when the program finishes. No complicated file service interface has to be mastered to use this model. Furthermore, whole file transfer is highly efficient. However, enough storage must be available on the client to store all the files required. Furthermore, if only a fraction of a file is needed, moving the entire file is wasteful.

The other kind of file service is the **remote access model**, as illustrated in Fig. 5-1(b). In this model, the file service provides a large number of operations for opening and closing files, reading and writing parts of files, moving around within files (LSEEK), examining and changing file attributes, and so on. Whereas in the upload/download model, the file service merely provides physical storage and transfer, here the file system runs on the servers, not on the clients. It has the advantage of not requiring much space on the clients, as well as eliminating the need to pull in entire files when only small pieces are needed.

5.1.2. The Directory Server Interface

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

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

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

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

The nature of the difficulty can be seen by looking at the directory graph of

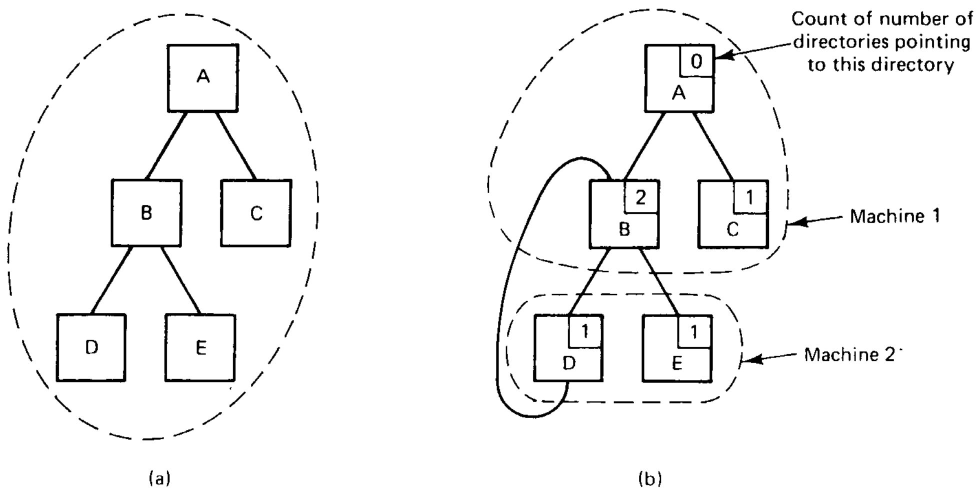

Fig. 5-2. (a) A directory tree contained on one machine. (b) A directory graph on two machines.

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

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

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

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

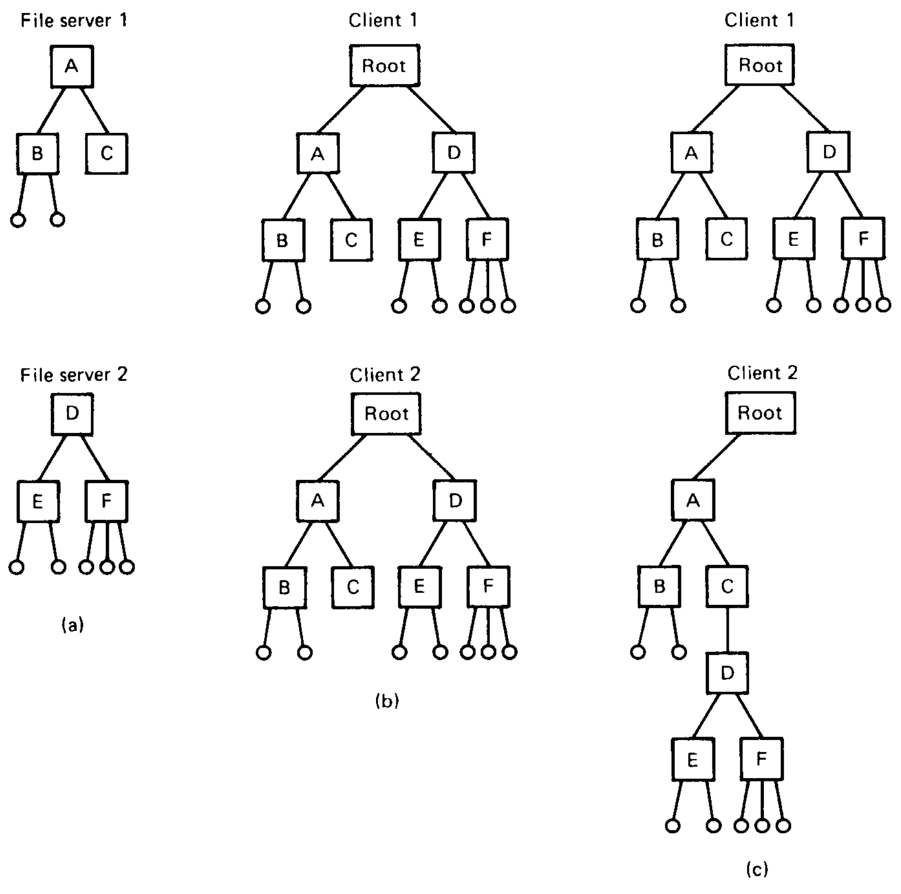

Fig. 5-3. (a) Two file servers. The squares are directories and the circles are files. (b) A system in which all clients have the same view of the file system. (c) A system in which different clients may have different views of the file system.

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

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

Naming Transparency

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

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

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

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

The first two are easy to implement, especially as a way to connect up existing