When Alice logs into Coda, she will first need to get authentication tokens from the AS. In this case, she does a secure RPC using her password to generate the secret key $K_{A,AS}$ she shares with the AS, and which is used for mutual authentication as explained before. The AS returns two authentication tokens. A **clear token** $CT = [A, TID, K_S, T_{start}, T_{end}]$ identifying Alice and containing a token identifier TID, a session key K_S, and two timestamps T_{start} and T_{end} indicating when the token is valid. In addition, it sends a **secret token** $ST = K_{vice}([CT]^{*}_{K_{vice}})$, which is CT cryptographically sealed with the secret key K_{vice} that is shared between all Vice servers, and encrypted with that same key.

A Vice server is capable of decrypting ST revealing CT and thus the session key K_S. Also, because only Vice servers know K_{vice}, such a server can easily check whether CT has been tampered with by doing an integrity check (of which the computation requires K_{vice}).

Whenever Alice wants to set up a secure channel with a Vice server, she uses the secret token ST to identify herself as shown in Fig. 10-33. The session key K_S that was handed to her by the AS in the clear token is used to encrypt the challenge R_A she sends to the server.

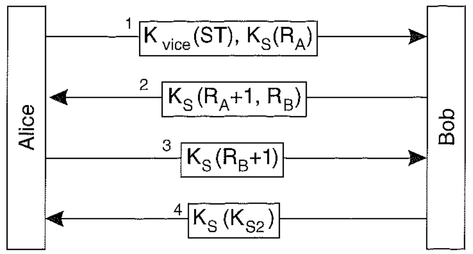

Figure 10-33. Setting up a secure channel between a (Venus) client and a Vice server in Coda.

The server, in turn, will first decrypt ST using the shared secret key K_{vice}, giving CT. It will then find K_S, which it subsequently uses to complete the authentication protocol. Of course, the server will also do an integrity check on CT and proceed only if the token is currently valid.

A problem occurs when a client needs to be authenticated before it can access files, but is currently disconnected. Authentication cannot be done because the server cannot be contacted. In this case, authentication is postponed and access is tentatively granted. When the client reconnects to the server(s), authentication takes place before entering the *REINTEGRATION* state.

Access Control

Let us briefly consider protection in Coda. As in AFS, Coda uses access control lists to ensure that only authorized processes have access to files. For reasons of simplicity and scalability, a Vice file server associates an access control list only with directories and not with files. All normal files in the same directory (i.e., excluding subdirectories) share the same protection rights.

Coda distinguishes access rights with respect to the types of operations shown in Fig. 10-34. Note that there is no right with respect to executing a file. There is simple reason for this omission: execution of files takes place at clients and is thus out of the scope of a Vice file server. Once a client has downloaded a file there is no way for Vice to even tell whether the client is executing it or just reading it.

Operation	Description
Read	Read any file in the directory
Write	Modify any file in the directory
Lookup	Look up the status of any file
Insert	Add a new file to the directory
Delete	Delete an existing file
Administer	Modify the ACL of the directory

Figure 10-34. Classification of file and directory operations recognized by Coda with respect to access control.

Coda maintains information on users and groups. Besides listing the rights a user or group has, Coda also supports the listing of negative rights. In other words, it is possible to explicitly state that a specific user is *not* permitted certain access rights. This approach has shown to be convenient in the light of immediately revoking access rights of a misbehaving user, without having to first remove that user from all groups.

10.3 OTHER DISTRIBUTED FILE SYSTEMS

There are many more distributed file systems than the ones we have discussed so far, although most have many similarities with NFS or Coda. In this section, we take a brief look at three systems that each are different in one or more respects. We first discuss Plan 9, in which every resource is treated as a file. XFS is an example of a serverless file system. Our last example is SFS, a file system in which file names also contain security information.

In contrast to the description of the NFS and Coda, we deliberately leave out many details of the three systems discussed in this section. Instead, we concentrate mainly on the general principles underlying each of the systems with the purpose of giving an impression of alternative approaches to distributed file systems.

10.3.1 Plan 9: Resources Unified to Files

Starting in the 1980s, there has been a trend to replace the centralized timesharing systems with networks of powerful workstations, leading to the network operating systems as we discussed in Chap. 1. One of the problems with this approach is that transparency is often lost, which, in turn, has led to developments such as distributed (operating) systems.

Plan 9 was developed in reaction to network operating systems by bringing back the idea of having a few centralized servers and numerous client machines. However, rather than having mainframes or minicomputers as servers, servers are assumed to be powerful and relatively cheap computers deploying microcomputer technology. The servers are centrally managed as before. Client machines, on the other hand, are assumed to be simple and have only a few tasks.

The system was largely conceived by the same group of people that were responsible for developing UNIX at Bell Labs. The project started in the late 1980s. Considering that many modern local distributed systems are organized with relatively simple clients and a number of powerful servers, the seemingly old-fashioned idea of having a centrally administered timesharing system is still a valid one today. An overview of Plan 9 is given in (Pike et al., 1995). Detailed information can be found in the Plan 9 programmer's manual (Bell Labs, 2000).

Plan 9 is not so much a distributed file system, but rather a *file-based distributed system*. All resources are accessed in the same way, namely with file-like syntax and operations, including even resources such as processes and network interfaces. This idea is inherited from UNIX, which also attempts to offer file-like interfaces to resources, but it has been exploited much further and more consistently in Plan 9. Each server offers a hierarchical name space to the resources it controls. A client can locally mount a name space as offered by a server, thus building its own private name space analogous to the approach followed in NFS. To allow sharing, part of the name space is standardized.

This approach leads to the organization shown in Fig. 10-35. A Plan 9 system consists of a collection of servers offering resources to clients in the form of local name spaces. To access a server's resources, a client mounts the server's name space into its own name space. Although we make a distinction between clients and servers, it should be noted that such a distinction is not always so clear in Plan 9. For example, servers often act as clients of other machines, whereas clients can export their resources to servers.

Communication

Communication across the network takes place through a standard protocol called **9P**, which is a protocol tailored to file-oriented operations. For example, 9P offers operations to open and close files, read and write chunks of data, and operations to traverse a hierarchical name space. 9P runs on a reliable transport protocol. For local-area networks, it uses a reliable datagram protocol called **Internet Link (IL)**. IL is a reliable, message-based protocol that provides FIFO-ordered message delivery. For wide-area networks, 9P uses TCP.

An interesting aspect of Plan 9 is the way it handles communication at the level of network interfaces. Network interfaces are represented by a file system,

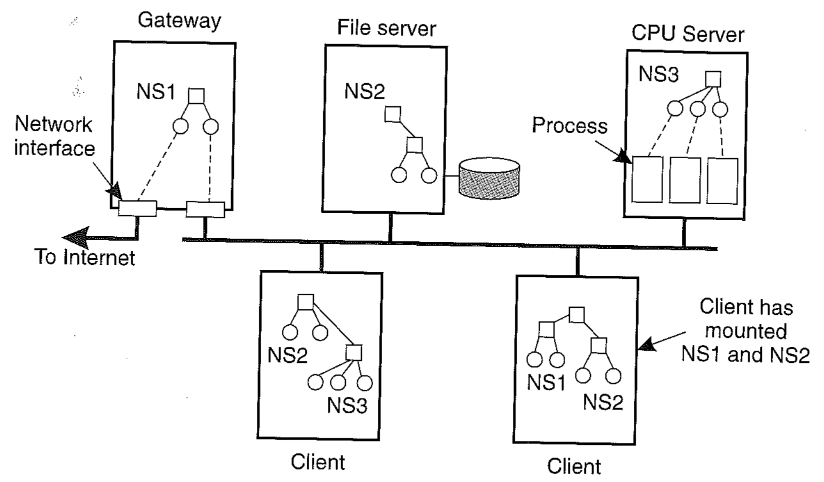

Figure 10-35. General organization of Plan 9.

in this case consisting of a collection of special files. This approach is similar to UNIX, although network interfaces in UNIX are represented by files and not file systems. (Note that a file system in this context is again the logical block device containing all the data and metadata that comprise a collection of files.) In Plan 9, for example, an individual TCP connection is represented by a subdirectory consisting of the files shown in Fig. 10-36.

File	Description
ctl	Used to write protocol-specific control commands
data	Used to read and write data
listen	Used to accept incoming connection setup requests
local	Provides information on the caller's side of the connection
remote	Provides information on the other side of the connection
status	Provides diagnostic information on the current status of the connection

Figure 10-36. Files associated with a single TCP connection in Plan 9.

The file *ctl* is used to send control commands to the connection. For example, to open a Telnet session to a machine with IP address 192.31.231.42 using port 23, requires that the sender writes the text string "connect 192.31.231.42!23" to file *ctl*. The receiver would previously have written the string "announce 23" to its own *ctl* file, indicating that it can accept incoming session requests.

The *data* file is used to exchange data by simply performing read and write operations. These operations follow the usual UNIX semantics for file operations. For example, to write data to a connection, a process simply calls the operation

```
res = write(fd, buf, nbytes);
```

where *fd* is the file descriptor returned after opening the data file, *buf* is a pointer to a buffer containing the data to be written, and *nbytes* is the number of bytes that should be extracted from the buffer. The number of bytes actually written is returned and stored in the variable *res*.

The file *listen* is used to wait for connection setup requests. After a process has announced its willingness to accept new connections, it can do a blocking read on file *listen*. If a request comes in, the call returns a file descriptor to a new *ctl* file corresponding to a newly created connection directory.

Processes

There are various servers in Plan 9. Each server implements a hierarchical name space. The simplest example is the Plan 9 file server, which is a stand-alone system running on a dedicated machine. The naming graph it exports is a tree representing a collection of files stored on multiple disks. Logically, the server is organized as a three-layer storage system as shown in Fig. 10-37.

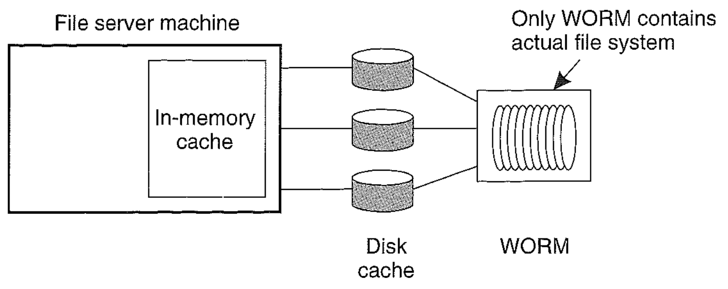

Figure 10-37. The Plan 9 file server.

The lowest layer is formed by a **Write-Once, Read-Many (WORM)** device providing bulk storage. In effect, it is like a CD-recordable, but much bigger. The file system as stored on this device forms the actual file system as seen by clients.

The middle layer consists of a collection of magnetic disks that act as a large caching system for the WORM device. When a file is accessed, it is read from the WORM device and saved on disk. Modifications are temporarily stored on disk as well, similar to traditional buffer caching in UNIX systems. Once a day, all modifications are written to the WORM device, which thus provides an incremental backup of the entire file system on a daily basis.

The highest layer is formed by a large collection of memory buffers that operates as an in-memory cache for the magnetic disks. Again, files are read in from disk and modifications are stored in the in-memory buffers that are occasionally flushed to disk.

A completely different example of a Plan 9 server is a server for the 8½ window system (Pike, 1991). Conceptually, this server offers a collection of files to client programs that correspond to devices such as a mouse or screen. The interfaces to these devices are like files; the devices are not implemented as files, but just appear to be like files. For example, a program running in a window is offered a file called */dev/mouse* to capture the current position of the mouse in its window. Each window is offered its own private version of */dev/mouse*. Likewise, keyboard input for a specific window is read from a file named */dev/cons*, which is also private to each window. Similarly, other files provide access to other services such as window control functions and graphical output.

The approach followed in Plan 9 offers interesting alternatives to offering services. For example, there is server called *exportfs* that allows a machine to export (part of) its local name space across the network. In effect, this server accepts 9P messages and translates those messages into local system calls. As an example, assume a machine *M* offers multiple network interfaces that are accessible through a local directory called */net*. Let */net/inet* denote the network interface offering low-level access to an external network. If *M* exports */net*, a client can use *M* as a gateway by locally mounting */net* and subsequently opening */net/inet*. By writing data to that file, the client will effectively be sending messages to the external network although the client machine itself has no interface to that network.

In a similar fashion, a client can export its own files to a remote compute server. In this case, the client can start a program on a remote host. By making its own files locally available to that program, the actual computing takes place on that host and not on the client machine. However, to the program, it appears as if it is executing while using the client's local name space. Note that this approach effectively turns the client machine into a file server for the program running on the remote host.

Naming

As we have said, each process in Plan 9 has its own private name space that is constructed by locally mounting remote name spaces. An interesting feature of naming in Plan 9, is that multiple name spaces can be mounted at the same mount point, leading to what is called a **union directory**. In such a directory, the different file systems appear to be Boolean ORed (although there is an ordering imposed as we discuss shortly). For example, assume that file system FS_A has subdirectories */home* and */usr*, whereas file system FS_B has subdirectories */bin*, */src*, and */lib*. As shown in Fig. 10-38, if a client mounts these two file systems at the same mount point, say */remote*, this directory will then subsequently appear to contain five subdirectories.

Creating union directories proceeds by mounting file systems in a specific order. This ordering is needed to solve name clashes. For example, if file system FS_A also had a subdirectory */bin*, then mounting FS_B *after* FS_A places subdirectory */bin* of FS_B behind that of FS_A. The effect is that when looking up a name

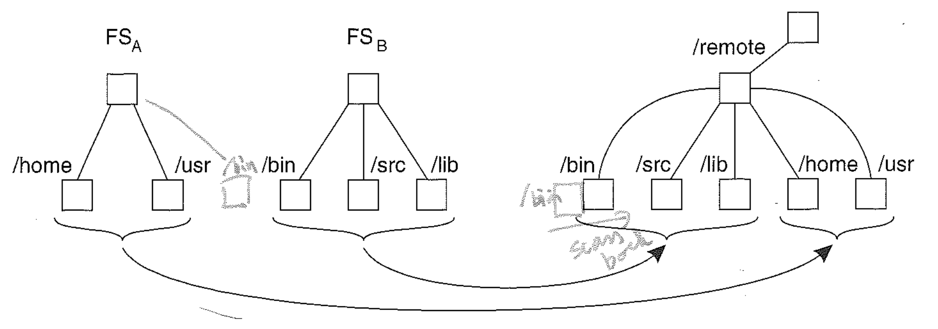

Figure 10-38. A union directory in Plan 9.

/remote/bin/prog, a file identifier for the file stored in FS_A will be returned if it exists; otherwise, the lookup operation continues searching the */bin* directory of FS_B.

When a file is created by a file server or device, it is labeled by two integers. A **path** is a unique file number that identifies the file relative to the server or device. Its **version** number is incremented each time the file is modified. To identify a file systemwide, a client that has mounted a file system maintains a type and device number for the server or device from which it mounted the file system. These numbers correspond the major and minor device numbers as used in UNIX. As a consequence, all files are globally and uniquely identified by four numbers: two to identify the server or device hosting the file, and two to uniquely identify the file relative to its host.

Synchronization

To improve efficiency, many distributed file systems transfer (part of) a file to the client when the file is opened. As a consequence, it may occur that several copies of a file exist that are modified concurrently by different clients. In the case of session semantics, the updates of the last client who closes a file are put into effect; updates from other clients are lost.

Plan 9 implements UNIX file sharing semantics by letting the file server always keep a copy of the file. All update operations are always forwarded to the server. Concurrent clients have their operations processed in an order determined by the server, but updates are never lost.

Caching and Replication

Plan 9 provides minimal support for caching and replication. Clients may cache files using a special user-level server called *cfs*, which is normally called automatically when a client machine boots. *Cfs* implements a write-through caching policy; update operations are always immediately sent to the server.

To avoid unnecessary file transfer, a client can use a previously cached copy provided it is still valid. Validity is checked by comparing the cached file's version number to that of the file at the server. If the file had been modified between the last time it was cached and the client's current access, the version numbers would differ. In that case, the client invalidates its cache and fetches the new copy from the server.

Security

In Plan 9, users are authenticated using an approach similar to that in the Needham-Schroeder protocol. A user is required to get a ticket from an authentication service to set up a secure channel to another user. In principle, users are authenticated, not machines. To let a machine or service act on behalf of a user, Plan 9 also supports delegation.

Files are protected the same way as in UNIX. What is unusual in comparison to other file systems, is Plan 9's notion of a group. In Plan 9, a group is considered to be just another user, possibly with an identified leader. A group leader has special rights, such as being allowed to change group permissions on a file. If there is no leader, then all group members are considered equal.

10.3.2 XFS: Serverless File System

Our next example of a somewhat unusual distributed file system is **xFS** (Anderson et al., 1996), which has been developed as part of the Berkeley NOW project (Anderson et al., 1995). XFS is unusual for its *serverless* design; the entire file system is distributed across multiple machines, including the clients. This approach contrasts with most other file systems, which are normally organized in a centralized fashion, even when there are multiple servers used for the distribution and replication of files. For example, AFS and Coda fall into the latter category.

We point out that there is a completely different distributed file system, called XFS (i.e., with a capital "X") that has been developed around the same time as xFS (Sweeney et al., 1996). In the following, we use "xFS" and "XFS" to refer only to the system developed as part of the NOW project.

Overview of xFS

XFS is designed to operate on a local-area network in which machines are interconnected through high-speed links. Many modern networks meet this requirement, notably clusters of workstations (which we briefly discussed in Chap. 1). By fully distributing data and control across the machines in a local-are network, the designers of xFS aimed at achieving higher scalability and fault tolerance then is possible with traditional distributed file systems that use a