In this section we discuss the first distributed file systems, developed in the 1980s by software companies

and universities. The systems covered are the Network File System developed by Sun Microsystems

in 1984, the Andrew File System developed at Carnegie Mellon University as part of the Andrew

project, and the Sprite Network File System developed by John Osterhout’s group at UC Berkeley as

a component of the Unix-like distributed operating system called Sprite. Other systems developed at

about the same time are Locus [365], Apollo [211], and the Remote File System (RFS) [35]. The main

concerns in the design of these systems were scalability, performance, and security (see Table 8.1.)

In the 1980s many organizations, including research centers, universities, financial institutions, and

design centers, considered networks of workstations to be an ideal environment for their operations.

Diskless workstations were appealing due to reduced hardware costs and because of lower maintenance

and system administration costs. Soon it became obvious that a distributed file system could be very

useful for the management of a large number of workstations. Sun Microsystems, one of the main

promoters of distributed systems based on workstations, proceeded to develop theNFSin the early 1980s.

Network File System (NFS). NFSwas the first widely used distributed file system; the development

of this application based on the client-server model was motivated by the need to share a file system

among a number of clients interconnected by a local area network.

A majority of workstations were running under Unix; thus, many design decisions for the NFS were

influenced by the design philosophy of the Unix File System (UFS). It is not surprising that the NFS

designers aimed to:

• Provide the same semantics as a local UFS to ensure compatibility with existing applications.

• Facilitate easy integration into existing UFS.

• Ensure that the system would be widely used and thus support clients running on different operating

systems.

• Accept a modest performance degradation due to remote access over a network with a bandwidth

of several Mbps.

Before we examine NFS in more detail, we have to analyze three important characteristics of the

Unix File System that enabled the extension from local to remote file management:

• The layered design provides the necessary flexibility for the file system; layering allows separation

of concerns and minimization of the interaction among the modules necessary to implement the

system. The addition of the vnode layer allowed the Unix File System to treat local and remote file

access uniformly.

• The hierarchical design supports scalability of the file system; indeed, it allows grouping of files

into special files called directories and supports multiple levels of directories and collections of

directories and files, the so-called file systems. The hierarchical file structure is reflected by the

file-naming convention.

• The metadata supports a systematic rather than an ad hoc design philosophy of the file system. The

so called inodes contain information about individual files and directories. The inodes are kept on

persistent media, together with the data. Metadata includes the file owner, the access rights, the

creation time or the time of the last modification of the file, the file size, and information about

the structure of the file and the persistent storage device cells where data is stored. Metadata also

supports device independence, a very important objective due to the very rapid pace of storage

technology development.

The logical organization of a file reflects the data model – the view of the data from the perspective

of the application. The physical organization reflects the storage model and describes the manner in

which the file is stored on a given storage medium. The layered design allows UFS to separate concerns

for the physical file structure from the logical one.

Recall that a file is a linear array of cells stored on a persistent storage device. The file pointer

identifies a cell used as a starting point for a read or write operation. This linear array is viewed

by an application as a collection of logical records; the file is stored on a physical device as a set of

physical records, or blocks, of a size dictated by the physical media.

The lower three layers of the UFS hierarchy – the block, the file, and the inode layer – reflect

the physical organization. The block layer allows the system to locate individual blocks on the physical

device; the file layer reflects the organization of blocks into files; and the inode layer provides

the metadata for the objects (files and directories). The upper three layers – the path name, the

absolute path name, and the symbolic path name layer – reflect the logical organization. The filename

layer mediates between the machine-oriented and the user-oriented views of the file system (see

Figure 8.3).

Several control structures maintained by the kernel of the operating system support file handling by

a running process. These structures are maintained in the user area of the process address space and can

only be accessed in kernel mode. To access a file, a process must first establish a connection with the

file system by opening the file. At that time a new entry is added to the file description table, and the

meta-information is brought into another control structure, the open file table.

A path specifies the location of a file or directory in a file system; a relative path specifies this

location relative to the current/working directory of the process, whereas a full path, also called an

absolute path, specifies the location of the file independently of the current directory, typically relative

to the root directory. A local file is uniquely identified by a file descriptor (fd), generally an index in the

open file table.

The Network File System is based on the client-server paradigm. The client runs on the local host

while the server is at the site of the remote file system, and they interact by means of remote procedure

calls (RPCs) (see Figure 8.4). The API interface of the local file system distinguishes file operations

on a local file from the ones on a remote file and, in the latter case, invokes the RPC client. Figure 8.5

shows the API for a Unix File System, with the calls made by the RPC client in response to API calls

issued by a user program for a remote file system as well as some of the actions carried out by the NFS server in response to an RPC call. NFS uses a vnode layer to distinguish between operations on local

and remote files, as shown in Figure 8.4.

A remote file is uniquely identified by a file handle (fh) rather than a file descriptor. The file handle

is a 32-byte internal name, a combination of the file system identification, an inode number, and a

generation number. The file handle allows the system to locate the remote file system and the file on

that system; the generation number allows the system to reuse the inode numbers and ensures correct

semantics when multiple clients operate on the same remote file.

Although many RPC calls, such as read, are idempotent,3 communication failures could sometimes

lead to unexpected behavior. Indeed, if the network fails to deliver the response to a read RPC, then

the call can be repeated without any side effects. By contrast, when the network fails to deliver the

response to the rmdir RPC, the second call returns an error code to the user if the call was successful

the first time. If the server fails to execute the first call, the second call returns normally. Note also that

there is no close RPC because this action only makes changes in the process open file structure and

does not affect the remote file.

The NFS has undergone significant transformations over the years. It has evolved from Version 2

[314], discussed in this section, to Version 3 [286] in 1994 and then to Version 4 [287] in 2000 (see

Section 8.11).

Andrew File System (AFS). AFS is a distributed file system developed in the late 1980s at Carnegie

Mellon University (CMU) in collaboration with IBM [250]. The designers of the system envisioned

a very large number of workstations interconnected with a relatively small number of servers; it was

anticipated that each individual at CMU would have an Andrew workstation, so the system would

connect up to 10,000 workstations. The set of trusted servers in AFS forms a structure called Vice.

The OS on a workstation, 4.2 BSD Unix, intercepts file system calls and forwards them to a user-level

process called Venus, which caches files from Vice and stores modified copies of files back on the

servers they came from. Reading and writing from/to a file are performed directly on the cached copy

and bypass Venus; only when a file is opened or closed does Venus communicate with Vice.

The emphasis of the AFS design was on performance, security, and simple management of the file

system [170]. To ensure scalability and to reduce response time, the local disks of the workstations are

used as persistent cache. The master copy of a file residing on one of the servers is updated only when

the file is modified. This strategy reduces the load placed on the servers and contributes to better system

performance.

Another major objective of the AFS design was improved security. The communications between

clients and servers are encrypted, and all file operations require secure network connections. When a

user signs into a workstation, the password is used to obtain security tokens from an authentication

server. These tokens are then used every time a file operation requires a secure network connection.

The AFS uses access control lists (ACLs) to allow control sharing of the data. An ACL specifies

the access rights of an individual user or group of users. A set of tools supports ACL management.

Another facet of the effort to reduce user involvement in file management is location transparency. The

files can be accessed from any location and can be moved automatically or at the request of system

administrators without user involvement and/or inconvenience. The relatively small number of servers

drastically reduces the efforts related to system administration because operations, such as backups,

affect only the servers, whereas workstations can be added, removed, or moved from one location to

another without administrative intervention.

Sprite Network File System (SFS). SFS is a component of the Sprite network operating system

[165]. SFS supports non-write-through caching of files on the client as well as the server systems [255].

Processes running on all workstations enjoy the same semantics for file access as they would if they

were run on a single system. This is possible due to a cache consistency mechanism that flushes portions

of the cache and disables caching for shared files opened for read/write operations.

Caching not only hides the network latency, it also reduces server utilization and obviously improves

performance by reducing response time. A file access request made by a client process could be satisfied

at different levels. First, the request is directed to the local cache; if it’s not satisfied there, it is passed

to the local file system of the client. If it cannot be satisfied locally then the request is sent to the remote

server. If the request cannot be satisfied by the remote server’s cache, it is sent to the file system running

on the server.

The design decisions for the Sprite system were influenced by the resources available at a time when a

typical workstation had a 1–2 MIPS processor and 4–14 Mbytes of physical memory. The main-memory caches allowed diskless workstations to be integrated into the system and enabled the development of

unique caching mechanisms and policies for both clients and servers. The results of a file-intensive

benchmark report [255] show that SFS was 30–35% faster than either NFS or AFS.

The file cache is organized as a collection of 4 KB blocks; a cache block has a virtual address

consisting of a unique file identifier supplied by the server and a block number in the file. Virtual

addressing allows the clients to create new blocks without the need to communicate with the server. File

servers map virtual to physical disk addresses. Note that the page size of the virtual memory in Sprite

is also 4K.

The size of the cache available to an SFS client or a server system changes dynamically as a function

of the needs. This is possible because the Sprite operating system ensures optimal sharing of the physical

memory between file caching by SFS and virtual memory management.

The file system and the virtual memory manage separate sets of physical memory pages and maintain

a time of last access for each block or page, respectively. Virtual memory uses a version of the clock

algorithm [254] to implement a least recently used (LRU) page replacement algorithm, and the file

system implements a strict LRU order, since it knows the time of each read and write operation.

Whenever the file system or the virtual memory management experiences a file cache miss or a page

fault, it compares the age of its oldest cache block or page, respectively, with the age of the oldest one

of the other system; the oldest cache block or page is forced to release the real memory frame.

An important design decision related to the SFS was to delay write-backs; this means that a block

is first written to cache, and the writing to the disk is delayed for a time of the order of tens of seconds.

This strategy speeds up writing and avoids writing when the data is discarded before the time to write

it to the disk. The obvious drawback of this policy is that data can be lost in case of a system failure.

Write-through is the alternative to the delayed write-back; it guarantees reliability because the block

is written to the disk as soon as it is available on the cache, but it increases the time for a write

operation.

Most network file systems guarantee that once a file is closed, the server will have the newest version

on persistent storage. As far as concurrency is concerned, we distinguish sequential write sharing, when

a file cannot be opened simultaneously for reading and writing by several clients, from concurrent write

sharing, when multiple clients can modify the file at the same time. Sprite allows both modes of

concurrency and delegates the cache consistency to the servers. In case of concurrent write sharing,

the client caching for the file is disabled; all reads and writes are carried out through the server.

Table 8.1 presents a comparison of caching, writing strategy, and consistency of NFS, AFS [250],

Sprite [165], Locus [365], Apollo [211], and the Remote File System (RFS) [35].