12

DISTRIBUTED FILE SYSTEMS

A distributed file system enables programs to store and access remote files exactly as they do local ones, allowing users to access files from any computer on a network. The performance and reliability experienced for access to files stored at a server should be comparable to that for files stored on local disks.

In this chapter we define a simple architecture for file systems and describe two basic distributed file service implementations with contrasting designs that have been in widespread use for over two decades:

- the Sun Network File System, NFS;
- the Andrew File System, AFS.

Each emulates the UNIX file system interface, with differing degrees of scalability, fault tolerance and deviation from the strict UNIX one-copy file update semantics.

Several related file systems that exploit new modes of data organization on disk or across multiple servers to achieve high-performance, fault-tolerant and scalable file systems are also reviewed. Other types of distributed storage system are described elsewhere in the book. These include peer-to-peer storage systems (Chapter 10), replicated file systems (Chapter 18), multimedia data servers (Chapter 20) and the particular style of storage service required to support Internet search and other large-scale, data-intensive applications (Chapter 21).

12.1 Introduction

In Chapters 1 and 2, we identified the sharing of resources as a key goal for distributed systems. The sharing of stored information is perhaps the most important aspect of distributed resource sharing. Mechanisms for data sharing take many forms and are described in several parts of this book. Web servers provide a restricted form of data sharing in which files stored locally, in file systems at the server or in servers on a local network, are made available to clients throughout the Internet. The design of large-scale wide area read-write file storage systems poses problems of load balancing, reliability, availability and security, whose resolution is the goal of the peer-to-peer file storage systems described in Chapter 10. Chapter 18 focuses on replicated storage systems that are suitable for applications requiring reliable access to data stored on systems where the availability of individual hosts cannot be guaranteed. In Chapter 20 we describe a media server that is designed to serve streams of video data to large numbers of users in real time. Chapter 21 describes a file system designed to support large-scale, data-intensive applications such as Internet search.

The requirements for sharing within local networks and intranets lead to a need for a different type of service – one that supports the persistent storage of data and programs of all types on behalf of clients and the consistent distribution of up-to-date data. The purpose of this chapter is to describe the architecture and implementation of these *basic* distributed file systems. We use the word 'basic' here to denote distributed file systems whose primary purpose is to emulate the functionality of a non-distributed file system for client programs running on multiple remote computers. They do not maintain multiple persistent replicas of files, nor do they support the bandwidth and timing guarantees required for multimedia data streaming – those requirements are addressed in later chapters. Basic distributed file systems provide an essential underpinning for organizational computing based on intranets.

File systems were originally developed for centralized computer systems and desktop computers as an operating system facility providing a convenient programming interface to disk storage. They subsequently acquired features such as access-control and file-locking mechanisms that made them useful for the sharing of data and programs. Distributed file systems support the sharing of information in the form of files and hardware resources in the form of persistent storage throughout an intranet. A well-designed file service provides access to files stored at a server with performance and reliability similar to, and in some cases better than, files stored on local disks. Their design is adapted to the performance and reliability characteristics of local networks, and hence they are most effective in providing shared persistent storage for use in intranets. The first file servers were developed by researchers in the 1970s [Birrell and Needham 1980, Mitchell and Dion 1982, Leach *et al.* 1983], and Sun's Network File System became available in the early 1980s [Sandberg *et al.* 1985, Callaghan 1999].

A file service enables programs to store and access remote files exactly as they do local ones, allowing users to access their files from any computer in an intranet. The concentration of persistent storage at a few servers reduces the need for local disk storage and (more importantly) enables economies to be made in the management and archiving of the persistent data owned by an organization. Other services, such as the name service, the user authentication service and the print service, can be more easily

Figure 12.1 Storage systems and their properties

	Sharing	*Persistence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	×	×	×	1	RAM
File system	×	✓	×	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	×	Web server
Distributed shared memory	✓	×	✓	✓	Ivy (DSM, Ch. 6)
Remote objects (RMI/ORB)	✓	×	×	1	CORBA
Persistent object store	✓	✓	×	1	CORBA Persistent State Service
Peer-to-peer storage system	✓	✓	✓	2	OceanStore (Ch. 10)

Types of consistency:
1: strict one-copy ✓: slightly weaker guarantees 2: considerably weaker guarantees

implemented when they can call upon the file service to meet their needs for persistent storage. Web servers are reliant on filing systems for the storage of the web pages that they serve. In organizations that operate web servers for external and internal access via an intranet, the web servers often store and access the material from a local distributed file system.

With the advent of distributed object-oriented programming, a need arose for the persistent storage and distribution of shared objects. One way to achieve this is to serialize objects (in the manner described in Section 4.3.2) and to store and retrieve the serialized objects using files. But this method for achieving persistence and distribution is impractical for rapidly changing objects, so several more direct approaches have been developed. Java remote object invocation and CORBA ORBs provide access to remote, shared objects, but neither of these ensures the persistence of the objects, nor are the distributed objects replicated.

Figure 12.1 provides an overview of types of storage system. In addition to those already mentioned, the table includes distributed shared memory (DSM) systems and persistent object stores. DSM was described in Chapter 6. It provides an emulation of a shared memory by the replication of memory pages or segments at each host, but it does not necessarily provide automatic persistence. Persistent object stores were introduced in Chapter 5. They aim to provide persistence for distributed shared objects. Examples include the CORBA Persistent State Service (see Chapter 8) and persistent extensions to Java [Jordan 1996, java.sun.com VIII]. Some research projects have developed in platforms that support the automatic replication and persistent storage of objects (for example, PerDiS [Ferreira *et al*. 2000] and Khazana [Carter *et al*. 1998]). Peer-to-peer storage systems offer scalability to support client loads much larger than the systems described in this chaper, but they incur high performance costs in providing secure access control and consistency between updatable replicas.

Figure 12.2 File system modules

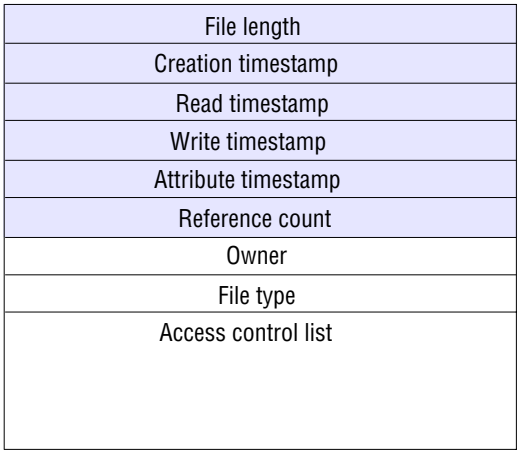
Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	performs disk I/O and buffering

The *consistency* column indicates whether mechanisms exist for the maintenance of consistency between multiple copies of data when updates occur. Virtually all storage systems rely on the use of caching to optimize the performance of programs. Caching was first applied to main memory and non-distributed file systems, and for those the consistency is strict (denoted by a '1', for one-copy consistency in Figure 12.1) – programs cannot observe any discrepancies between cached copies and stored data after an update. When distributed replicas are used, strict consistency is more difficult to achieve. Distributed file systems such as Sun NFS and the Andrew File System cache copies of portions of files at client computers, and they adopt specific consistency mechanisms to maintain an approximation to strict consistency – this is indicated by a tick (✓) in the consistency column of Figure 12.1. We discuss these mechanisms and the degree to which they deviate from strict consistency in Sections 12.3 and 12.4.

The Web uses caching extensively both at client computers and at proxy servers maintained by user organizations. The consistency between the copies stored at web proxies and client caches and the original server is only maintained by explicit user actions. Clients are not notified when a page stored at the original server is updated; they must perform explicit checks to keep their local copies up-to-date. This serves the purposes of web browsing adequately, but it does not support the development of cooperative applications such as a shared distributed whiteboard. The consistency mechanisms used in DSM systems are discussed in depth on the companion web site to the book [www.cdk5.net]. Persistent object systems vary considerably in their approach to caching and consistency. The CORBA and Persistent Java schemes maintain single copies of persistent objects, and remote invocation is required to access them, so the only consistency issue is between the persistent copy of an object on disk and the active copy in memory, which is not visible to remote clients. The PerDiS and Khazana projects that we mentioned above maintain cached replicas of objects and employ quite elaborate consistency mechanisms to produce forms of consistency similar to those found in DSM systems.

Having introduced some wider issues relating to storage and distribution of persistent and non-persistent data, we now return to the main topic of this chapter – the design of basic distributed file systems. We describe some relevant characteristics of (non-distributed) file systems in Section 12.1.1 and the requirements for distributed file systems in Section 12.1.2. Section 12.1.3 introduces the case studies that will be used throughout the chapter. In Section 12.2, we define an abstract model for a basic

Figure 12.3 File attribute record structure



distributed file service, including a set of programming interfaces. Sun NFS is described in Section 12.3; it shares many of the features of the abstract model. In Section 12.4 we describe the Andrew File System, a widely used system that employs substantially different caching and consistency mechanisms. Section 12.5 reviews some recent developments in the design of file services.

The systems described in this chapter do not cover the full spectrum of distributed file and data management systems. Several systems with more advanced characteristics will be described later in the book. Chapter 18 includes a description of Coda, a distributed file system that maintains persistent replicas of files for reliability, availability and disconnected working. Bayou, a distributed data management system that provides a weakly consistent form of replication for high availability, is also covered in Chapter 18. Chapter 20 covers the Tiger video file server, which is designed to provide timely delivery of streams of data to large numbers of clients. Chapter 21 describes the Google File System (GFS), a file system designed specifically to support large-scale, data-intensive applications including Internet search.

12.1.1 Characteristics of file systems

File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files. They provide a programming interface that characterizes the file abstraction, freeing programmers from concern with the details of storage allocation and layout. Files are stored on disks or other non-volatile storage media.

Files contain both *data* and *attributes*. The data consist of a sequence of data items (typically 8-bit bytes), accessible by operations to read and write any portion of the sequence. The attributes are held as a single record containing information such as the length of the file, timestamps, file type, owner's identity and access control lists. A typical attribute record structure is illustrated in Figure 12.3. The shaded attributes are managed by the file system and are not normally updatable by user programs.

Figure 12.4 UNIX file system operations

filedes = open(name, mode) *filedes = creat(name, mode)*	Opens an existing file with the given *name*. Creates a new file with the given *name*. Both operations deliver a file descriptor referencing the open file. The *mode* is *read*, *write* or both.
status = close(filedes)	Closes the open file *filedes*.
count = read(filedes, buffer, n) *count = write(filedes, buffer, n)*	Transfers *n* bytes from the file referenced by *filedes* to *buffer*. Transfers *n* bytes to the file referenced by *filedes* from *buffer*. Both operations deliver the number of bytes actually transferred and advance the read-write pointer.
pos = lseek(filedes, offset, whence)	Moves the read-write pointer to *offset* (relative or absolute, depending on *whence*).
status = unlink(name)	Removes the file *name* from the directory structure. If the file has no other names, it is deleted.
status = link(name1, name2)	Adds a new name (*name2*) for a file (*name1*).
status = stat(name, buffer)	Puts the file attributes for file *name* into *buffer*.

File systems are designed to store and manage large numbers of files, with facilities for creating, naming and deleting files. The naming of files is supported by the use of directories. A *directory* is a file, often of a special type, that provides a mapping from text names to internal file identifiers. Directories may include the names of other directories, leading to the familiar hierarchic file-naming scheme and the multi-part *pathnames* for files used in UNIX and other operating systems. File systems also take responsibility for the control of access to files, restricting access to files according to users' authorizations and the type of access requested (reading, updating, executing and so on).

The term *metadata* is often used to refer to all of the extra information stored by a file system that is needed for the management of files. It includes file attributes, directories and all the other persistent information used by the file system.

Figure 12.2 shows a typical layered module structure for the implementation of a non-distributed file system in a conventional operating system. Each layer depends only on the layers below it. The implementation of a distributed file service requires all of the components shown there, with additional components to deal with client-server communication and with the distributed naming and location of files.

File system operations • Figure 12.4 summarizes the main operations on files that are available to applications in UNIX systems. These are the system calls implemented by the kernel; application programmers usually access them through procedure libraries such as the C Standard Input/Output Library or the Java file classes. We give the primitives here as an indication of the operations that file services are expected to support and for comparison with the file service interfaces that we shall introduce below.

The UNIX operations are based on a programming model in which some file state information is stored by the file system for each running program. This consists of a list of currently open files with a read-write pointer for each, giving the position within the file at which the next read or write operation will be applied.

The file system is responsible for applying access control for files. In local file systems such as UNIX, it does so when each file is opened, checking the rights allowed for the user's identity in the access control list against the *mode* of access requested in the *open* system call. If the rights match the mode, the file is opened and the *mode* is recorded in the open file state information.

12.1.2 Distributed file system requirements

Many of the requirements and potential pitfalls in the design of distributed services were first observed in the early development of distributed file systems. Initially, they offered access transparency and location transparency; performance, scalability, concurrency control, fault tolerance and security requirements emerged and were met in subsequent phases of development. We discuss these and related requirements in the following subsections.

Transparency • The file service is usually the most heavily loaded service in an intranet, so its functionality and performance are critical. The design of the file service should support many of the transparency requirements for distributed systems identified in Section 1.5.7. The design must balance the flexibility and scalability that derive from transparency against software complexity and performance. The following forms of transparency are partially or wholly addressed by current file services:

Access transparency: Client programs should be unaware of the distribution of files. A single set of operations is provided for access to local and remote files. Programs written to operate on local files are able to access remote files without modification.

Location transparency: Client programs should see a uniform file name space. Files or groups of files may be relocated without changing their pathnames, and user programs see the same name space wherever they are executed.

Mobility transparency: Neither client programs nor system administration tables in client nodes need to be changed when files are moved. This allows file mobility – files or, more commonly, sets or volumes of files may be moved, either by system administrators or automatically.

Performance transparency: Client programs should continue to perform satisfactorily while the load on the service varies within a specified range.

Scaling transparency: The service can be expanded by incremental growth to deal with a wide range of loads and network sizes.

Concurrent file updates • Changes to a file by one client should not interfere with the operation of other clients simultaneously accessing or changing the same file. This is the well-known issue of concurrency control, discussed in detail in Chapter 16. The need for concurrency control for access to shared data in many applications is widely accepted and techniques are known for its implementation, but they are costly. Most current file

services follow modern UNIX standards in providing advisory or mandatory file- or record-level locking.

File replication • In a file service that supports replication, a file may be represented by several copies of its contents at different locations. This has two benefits – it enables multiple servers to share the load of providing a service to clients accessing the same set of files, enhancing the scalability of the service, and it enhances fault tolerance by enabling clients to locate another server that holds a copy of the file when one has failed. Few file services support replication fully, but most support the caching of files or portions of files locally, a limited form of replication. The replication of data is discussed in Chapter 18, which includes a description of the Coda replicated file service.

Hardware and operating system heterogeneity • The service interfaces should be defined so that client and server software can be implemented for different operating systems and computers. This requirement is an important aspect of openness.

Fault tolerance • The central role of the file service in distributed systems makes it essential that the service continue to operate in the face of client and server failures. Fortunately, a moderately fault-tolerant design is straightforward for simple servers. To cope with transient communication failures, the design can be based on *at-most-once* invocation semantics (see Section 5.3.1); or it can use the simpler *at-least-once* semantics with a server protocol designed in terms of *idempotent* operations, ensuring that duplicated requests do not result in invalid updates to files. The servers can be *stateless*, so that they can be restarted and the service restored after a failure without any need to recover previous state. Tolerance of disconnection or server failures requires file replication, which is more difficult to achieve and will be discussed in Chapter 18.

Consistency • Conventional file systems such as that provided in UNIX offer *one-copy update semantics*. This refers to a model for concurrent access to files in which the file contents seen by all of the processes accessing or updating a given file are those that they would see if only a single copy of the file contents existed. When files are replicated or cached at different sites, there is an inevitable delay in the propagation of modifications made at one site to all of the other sites that hold copies, and this may result in some deviation from one-copy semantics.

Security • Virtually all file systems provide access-control mechanisms based on the use of access control lists. In distributed file systems, there is a need to authenticate client requests so that access control at the server is based on correct user identities and to protect the contents of request and reply messages with digital signatures and (optionally) encryption of secret data. We discuss the impact of these requirements in the case studies later in this chapter.

Efficiency • A distributed file service should offer facilities that are of at least the same power and generality as those found in conventional file systems and should achieve a comparable level of performance. Birrell and Needham [1980] expressed their design aims for the Cambridge File Server (CFS) in these terms:

> We would wish to have a simple, low-level file server in order to share an expensive resource, namely a disk, whilst leaving us free to design the filing system most appropriate to a particular client, but we would wish also to have available a high-level system shared between clients.

The changed economics of disk storage have reduced the significance of their first goal, but their perception of the need for a range of services addressing the requirements of clients with different goals remains and can best be addressed by a modular architecture of the type outlined above.

The techniques used for the implementation of file services are an important part of the design of distributed systems. A distributed file system should provide a service that is comparable with, or better than, local file systems in performance and reliability. It must be convenient to administer, providing operations and tools that enable system administrators to install and operate the system conveniently.

12.1.3 Case studies

We have constructed an abstract model for a file service to act as an introductory example, separating implementation concerns and providing a simplified model. We describe the Sun Network File System in some detail, drawing on our simpler abstract model to clarify its architecture. The Andrew File System is then described, providing a view of a distributed file system that takes a different approach to scalability and consistency maintenance.

File service architecture • This is an abstract architectural model that underpins both NFS and AFS. It is based upon a division of responsibilities between three modules – a client module that emulates a conventional file system interface for application programs, and server modules, that perform operations for clients on directories and on files. The architecture is designed to enable a *stateless* implementation of the server module.

SUN NFS • Sun Microsystems's *Network File System* (NFS) has been widely adopted in industry and in academic environments since its introduction in 1985. The design and development of NFS were undertaken by staff at Sun Microsystems in 1984 [Sandberg *et al.* 1985, Sandberg 1987, Callaghan 1999]. Although several distributed file services had already been developed and used in universities and research laboratories, NFS was the first file service that was designed as a product. The design and implementation of NFS have achieved success both technically and commercially.

To encourage its adoption as a standard, the definitions of the key interfaces were placed in the public domain [Sun 1989], enabling other vendors to produce implementations, and the source code for a reference implementation was made available to other computer vendors under licence. It is now supported by many vendors, and the NFS protocol (version 3) is an Internet standard, defined in RFC 1813 [Callaghan *et al.* 1995]. Callaghan's book on NFS [Callaghan 1999] is an excellent source on the design and development of NFS and related topics.

NFS provides transparent access to remote files for client programs running on UNIX and other systems. The client-server relationship is symmetrical: each computer in an NFS network can act as both a client and a server, and the files at every machine can be made available for remote access by other machines. Any computer can be a server, exporting some of its files, and a client, accessing files on other machines. But it is common practice to configure larger installations with some machines as dedicated servers and others as workstations.

An important goal of NFS is to achieve a high level of support for hardware and operating system heterogeneity. The design is operating system–independent: client and server implementations exist for almost all operating systems and platforms, including all versions of Windows, Mac OS, Linux and every other version of UNIX. Implementations of NFS on high-performance multiprocessor hosts have been developed by several vendors, and these are widely used to meet storage requirements in intranets with many concurrent users.

Andrew File System • Andrew is a distributed computing environment developed at Carnegie Mellon University (CMU) for use as a campus computing and information system [Morris *et al.* 1986]. The design of the Andrew File System (henceforth abbreviated AFS) reflects an intention to support information sharing on a large scale by minimizing client-server communication. This is achieved by transferring whole files between server and client computers and caching them at clients until the server receives a more up-to-date version. We shall describe AFS-2, the first 'production' implementation, following the descriptions by Satyanarayanan [1989a, 1989b]. More recent descriptions can be found in Campbell [1997] and [Linux AFS].

AFS was initially implemented on a network of workstations and servers running BSD UNIX and the Mach operating system at CMU and was subsequently made available in commercial and public-domain versions. A public-domain implementation of AFS is available in the Linux operating system [Linux AFS]. AFS was adopted as the basis for the DCE/DFS file system in the Open Software Foundation's Distributed Computing Environment (DCE) [www.opengroup.org]. The design of DCE/DFS went beyond AFS in several important respects, which we outline in Section 12.5.

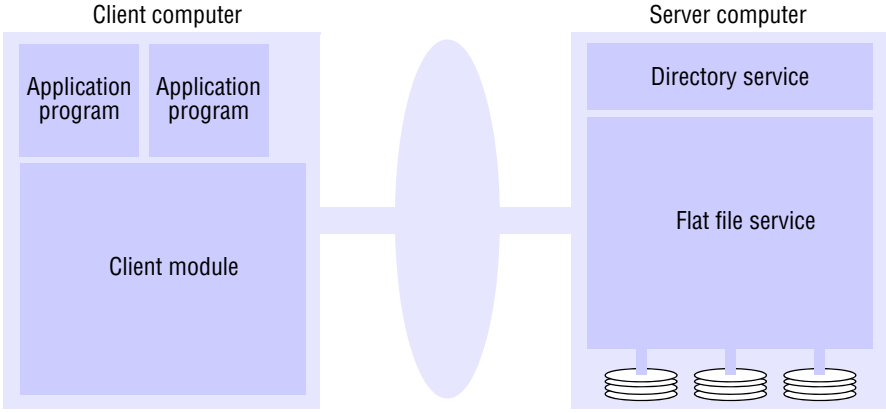
12.2 File service architecture

An architecture that offers a clear separation of the main concerns in providing access to files is obtained by structuring the file service as three components – a *flat file service*, a *directory service* and a *client module*. The relevant modules and their relationships are shown in Figure 12.5. The flat file service and the directory service each export an interface for use by client programs, and their RPC interfaces, taken together, provide a comprehensive set of operations for access to files. The client module provides a single programming interface with operations on files similar to those found in conventional file systems. The design is *open* in the sense that different client modules can be used to implement different programming interfaces, simulating the file operations of a variety of different operating systems and optimizing the performance for different client and server hardware configurations.

The division of responsibilities between the modules can be defined as follows:

Flat file service • The flat file service is concerned with implementing operations on the contents of files. *Unique file identifiers* (UFIDs) are used to refer to files in all requests for flat file service operations. The division of responsibilities between the file service and the directory service is based upon the use of UFIDs. UFIDs are long sequences of bits chosen so that each file has a UFID that is unique among all of the files in a

Figure 12.5 File service architecture



distributed system. When the flat file service receives a request to create a file, it generates a new UFID for it and returns the UFID to the requester.

Directory service • The directory service provides a mapping between *text names* for files and their UFIDs. Clients may obtain the UFID of a file by quoting its text name to the directory service. The directory service provides the functions needed to generate directories, to add new file names to directories and to obtain UFIDs from directories. It is a client of the flat file service; its directory files are stored in files of the flat file service. When a hierarchic file-naming scheme is adopted, as in UNIX, directories hold references to other directories.

Client module • A client module runs in each client computer, integrating and extending the operations of the flat file service and the directory service under a single application programming interface that is available to user-level programs in client computers. For example, in UNIX hosts, a client module would be provided that emulates the full set of UNIX file operations, interpreting UNIX multi-part file names by iterative requests to the directory service. The client module also holds information about the network locations of the flat file server and directory server processes. Finally, the client module can play an important role in achieving satisfactory performance through the implementation of a cache of recently used file blocks at the client.

Flat file service interface • Figure 12.6 contains a definition of the interface to a flat file service. This is the RPC interface used by client modules. It is not normally used directly by user-level programs. A *FileId* is invalid if the file that it refers to is not present in the server processing the request or if its access permissions are inappropriate for the operation requested. All of the procedures in the interface except *Create* throw exceptions if the *FileId* argument contains an invalid UFID or the user doesn't have sufficient access rights. These exceptions are omitted from the definition for clarity.

The most important operations are those for reading and writing. Both the *Read* and the *Write* operation require a parameter *i* specifying a position in the file. The *Read* operation copies the sequence of *n* data items beginning at item *i* from the specified file

Figure 12.6 Flat file service operations

Read(FileId, i, n) → *Data* — throws *BadPosition*	If $1 \le i \le Length(File)$: Reads a sequence of up to *n* items from a file starting at item *i* and returns it in *Data*.
Write(FileId, i, Data) — throws *BadPosition*	If $1 \le i \le Length(File)+1$: Writes a sequence of *Data* to a file, starting at item *i*, extending the file if necessary.
Create() → *FileId*	Creates a new file of length 0 and delivers a UFID for it.
Delete(FileId)	Removes the file from the file store.
GetAttributes(FileId) → *Attr*	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes (only those attributes that are not shaded in Figure 12.3).

into *Data*, which is then returned to the client. The *Write* operation copies the sequence of data items in *Data* into the specified file beginning at item *i*, replacing the previous contents of the file at the corresponding position and extending the file if necessary.

Create creates a new, empty file and returns the UFID that is generated. *Delete* removes the specified file.

GetAttributes and *SetAttributes* enable clients to access the attribute record. *GetAttributes* is normally available to any client that is allowed to read the file. Access to the *SetAttributes* operation would normally be restricted to the directory service that provides access to the file. The values of the length and timestamp portions of the attribute record are not affected by *SetAttributes*; they are maintained separately by the flat file service itself.

Comparison with UNIX: Our interface and the UNIX file system primitives are functionally equivalent. It is a simple matter to construct a client module that emulates the UNIX system calls in terms of our flat file service and the directory service operations described in the next section.

In comparison with the UNIX interface, our flat file service has no *open* and *close* operations – files can be accessed immediately by quoting the appropriate UFID. The *Read* and *Write* requests in our interface include a parameter specifying a starting point within the file for each transfer, whereas the equivalent UNIX operations do not. In UNIX, each *read* or *write* operation starts at the current position of the read-write pointer, and the read-write pointer is advanced by the number of bytes transferred after each *read* or *write*. A *seek* operation is provided to enable the read-write pointer to be explicitly repositioned.

The interface to our flat file service differs from the UNIX file system interface mainly for reasons of fault tolerance:

Repeatable operations: With the exception of *Create*, the operations are *idempotent*, allowing the use of *at-least-once* RPC semantics – clients may repeat calls to which they receive no reply. Repeated execution of *Create* produces a different new file for each call.

Stateless servers: The interface is suitable for implementation by *stateless* servers. Stateless servers can be restarted after a failure and resume operation without any need for clients or the server to restore any state.

The UNIX file operations are neither idempotent nor consistent with the requirement for a stateless implementation. A read-write pointer is generated by the UNIX file system whenever a file is opened, and it is retained, together with the results of access-control checks, until the file is closed. The UNIX *read* and *write* operations are not idempotent; if an operation is accidentally repeated, the automatic advance of the read-write pointer results in access to a different portion of the file in the repeated operation. The read-write pointer is a hidden, client-related state variable. To mimic it in a file server, *open* and *close* operations would be needed, and the read-write pointer's value would have to be retained by the server as long as the relevant file is open. By eliminating the read-write pointer, we have eliminated most of the need for the file server to retain state information on behalf of specific clients.

Access control • In the UNIX file system, the user's access rights are checked against the access *mode* (read or write) requested in the *open* call (Figure 12.4 shows the UNIX file system API) and the file is opened only if the user has the necessary rights. The user identity (UID) used in the access rights check is retrieved during the user's earlier authenticated login and cannot be tampered with in non-distributed implementations. The resulting access rights are retained until the file is closed, and no further checks are required when subsequent operations on the same file are requested.

In distributed implementations, access rights checks have to be performed at the server because the server RPC interface is an otherwise unprotected point of access to files. A user identity has to be passed with requests, and the server is vulnerable to forged identities. Furthermore, if the results of an access rights check were retained at the server and used for future accesses, the server would no longer be stateless. Two alternative approaches to the latter problem can be adopted:

- An access check is made whenever a file name is converted to a UFID, and the results are encoded in the form of a capability (see Section 11.2.4), which is returned to the client for submission with subsequent requests.
- A user identity is submitted with every client request, and access checks are performed by the server for every file operation.

Both methods enable stateless server implementation, and both have been used in distributed file systems. The second is more common; it is used in both NFS and AFS. Neither of these approaches overcomes the security problem concerning forged user identities, but we saw in Chapter 11 that this can be addressed by the use of digital signatures. Kerberos is an effective authentication scheme that has been applied to both NFS and AFS.

In our abstract model, we make no assumption about the method by which access control is implemented. The user identity is passed as an implicit parameter and can be used whenever it is needed.

Directory service interface • Figure 12.7 contains a definition of the RPC interface to a directory service. The primary purpose of the directory service is to provide a service for translating text names to UFIDs. In order to do so, it maintains directory files containing

Figure 12.7 Directory service operations

Lookup(Dir, Name) → *FileId* — throws *NotFound*	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, FileId) — throws *NameDuplicate*	If *Name* is not in the directory, adds *(Name, File)* to the directory and updates the file's attribute record. If *Name* is already in the directory, throws an exception.
UnName(Dir, Name) — throws *NotFound*	If *Name* is in the directory, removes the entry containing *Name* from the directory. If *Name* is not in the directory, throws an exception.
GetNames(Dir, Pattern) → *NameSeq*	Returns all the text names in the directory that match the regular expression *Pattern*.

the mappings between text names for files and UFIDs. Each directory is stored as a conventional file with a UFID, so the directory service is a client of the file service.

We define only operations on individual directories. For each operation, a UFID for the file containing the directory is required (in the *Dir* parameter). The *Lookup* operation in the basic directory service performs a single *Name* → *UFID* translation. It is a building block for use in other services or in the client module to perform more complex translations, such as the hierarchic name interpretation found in UNIX. As before, exceptions caused by inadequate access rights are omitted from the definitions.

There are two operations for altering directories: *AddName* and *UnName*. *AddName* adds an entry to a directory and increments the reference count field in the file's attribute record.

UnName removes an entry from a directory and decrements the reference count. If this causes the reference count to reach zero, the file is removed. *GetNames* is provided to enable clients to examine the contents of directories and to implement pattern-matching operations on file names such as those found in the UNIX shell. It returns all or a subset of the names stored in a given directory. The names are selected by pattern matching against a regular expression supplied by the client.

The provision of pattern matching in the *GetNames* operation enables users to determine the names of one or more files by giving an incomplete specification of the characters in the names. A regular expression is a specification for a class of strings in the form of an expression containing a combination of literal substrings and symbols denoting variable characters or repeated occurrences of characters or substrings.

Hierarchic file system • A hierarchic file system such as the one that UNIX provides consists of a number of directories arranged in a tree structure. Each directory holds the names of the files and other directories that are accessible from it. Any file or directory can be referenced using a *pathname* – a multi-part name that represents a path through

the tree. The root has a distinguished name, and each file or directory has a name in a directory. The UNIX file-naming scheme is not a strict hierarchy – files can have several names, and they can be in the same or different directories. This is implemented by a *link* operation, which adds a new name for a file to a specified directory.

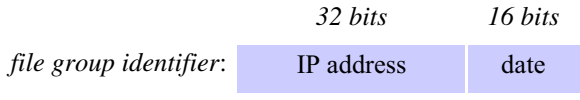
A UNIX-like file-naming system can be implemented by the client module using the flat file and directory services that we have defined. A tree-structured network of directories is constructed with files at the leaves and directories at the other nodes of the tree. The root of the tree is a directory with a 'well-known' UFID. Multiple names for files can be supported using the *AddName* operation and the reference count field in the attribute record.

A function can be provided in the client module that gets the UFID of a file given its pathname. The function interprets the pathname starting from the root, using *Lookup* to obtain the UFID of each directory in the path.

In a hierarchic directory service, the file attributes associated with files should include a type field that distinguishes between ordinary files and directories. This is used when following a path to ensure that each part of the name, except the last, refers to a directory.

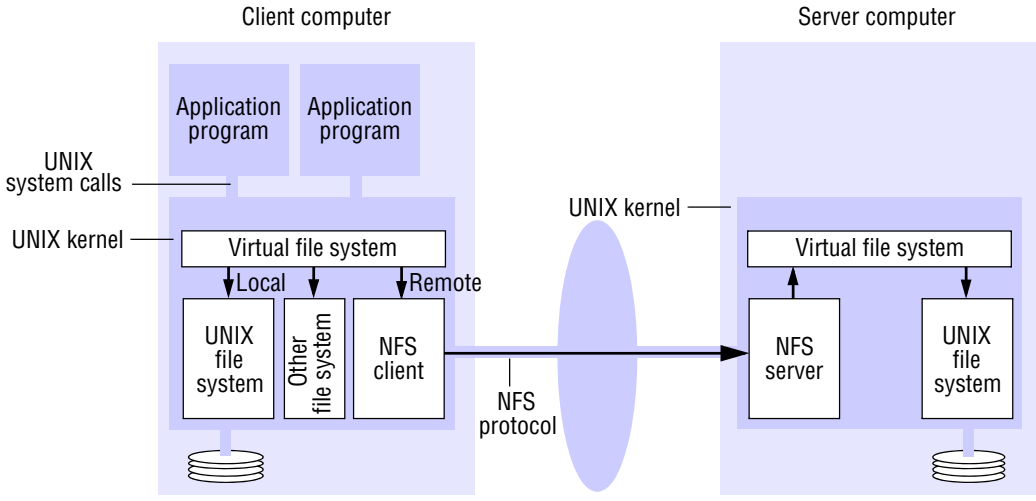
File groups • A *file group* is a collection of files located on a given server. A server may hold several file groups, and groups can be moved between servers, but a file cannot change the group to which it belongs. A similar construct called a *filesystem* is used in UNIX and in most other operating systems. (Terminology note: the single word *filesystem* refers to the set of files held in a storage device or partition, whereas the words *file system* refer to a software component that provides access to files.) File groups were originally introduced to support facilities for moving collections of files stored on removable media between computers. In a distributed file service, file groups support the allocation of files to file servers in larger logical units and enable the service to be implemented with files stored on several servers. In a distributed file system that supports file groups, the representation of UFIDs includes a file group identifier component, enabling the client module in each client computer to take responsibility for dispatching requests to the server that holds the relevant file group.

File group identifiers must be unique throughout a distributed system. Since file groups can be moved and distributed systems that are initially separate can be merged to form a single system, the only way to ensure that file group identifiers will always be distinct in a given system is to generate them with an algorithm that ensures global uniqueness. For example, whenever a new file group is created, a unique identifier can be generated by concatenating the 32-bit IP address of the host creating the new group with a 16-bit integer derived from the date, producing a unique 48-bit integer:



Note that the IP address *cannot* be used for the purpose of locating the file group, since it may be moved to another server. Instead, a mapping between group identifiers and servers should be maintained by the file service.

Figure 12.8 NFS architecture



12.3 Case study: Sun Network File System

Figure 12.8 shows the architecture of Sun NFS. It follows the abstract model defined in the preceding section. All implementations of NFS support the NFS protocol – a set of remote procedure calls that provide the means for clients to perform operations on a remote file store. The NFS protocol is operating system–independent but was originally developed for use in networks of UNIX systems, and we shall describe the UNIX implementation the NFS protocol (version 3).

The *NFS server* module resides in the kernel on each computer that acts as an NFS server. Requests referring to files in a remote file system are translated by the client module to NFS protocol operations and then passed to the NFS server module at the computer holding the relevant file system.

The NFS client and server modules communicate using remote procedure calls. Sun's RPC system, described in Section 5.3.3, was developed for use in NFS. It can be configured to use either UDP or TCP, and the NFS protocol is compatible with both. A port mapper service is included to enable clients to bind to services in a given host by name. The RPC interface to the NFS server is open: any process can send requests to an NFS server; if the requests are valid and they include valid user credentials, they will be acted upon. The submission of signed user credentials can be required as an optional security feature, as can the encryption of data for privacy and integrity.

Virtual file system • Figure 12.8 makes it clear that NFS provides access transparency: user programs can issue file operations for local or remote files without distinction. Other distributed file systems may be present that support UNIX system calls, and if so, they could be integrated in the same way.

The integration is achieved by a virtual file system (VFS) module, which has been added to the UNIX kernel to distinguish between local and remote files and to translate between the UNIX-independent file identifiers used by NFS and the internal file identifiers normally used in UNIX and other file systems. In addition, VFS keeps track of the filesystems that are currently available both locally and remotely, and it passes each request to the appropriate local system module (the UNIX file system, the NFS client module or the service module for another file system).

The file identifiers used in NFS are called *file handles*. A file handle is opaque to clients and contains whatever information the server needs to distinguish an individual file. In UNIX implementations of NFS, the file handle is derived from the file's *i-node number* by adding two extra fields as follows (the i-node number of a UNIX file is a number that serves to identify and locate the file within the file system in which the file is stored):

File handle:	Filesystem identifier	i-node number of file	i-node generation number

NFS adopts the UNIX mountable filesystem as the unit of file grouping defined in the preceding section. The *filesystem identifier* field is a unique number that is allocated to each filesystem when it is created (and in the UNIX implementation is stored in the superblock of the file system). The *i-node generation number* is needed because in the conventional UNIX file system i-node numbers are reused after a file is removed. In the VFS extensions to the UNIX file system, a generation number is stored with each file and is incremented each time the i-node number is reused (for example, in a UNIX *creat* system call). The client obtains the first file handle for a remote file system when it mounts it. File handles are passed from server to client in the results of *lookup*, *create* and *mkdir* operations (see Figure 12.9) and from client to server in the argument lists of all server operations.

The virtual file system layer has one VFS structure for each mounted file system and one *v-node* per open file. A VFS structure relates a remote file system to the local directory on which it is mounted. The v-node contains an indicator to show whether a file is local or remote. If the file is local, the v-node contains a reference to the index of the local file (an i-node in a UNIX implementation). If the file is remote, it contains the file handle of the remote file.

Client integration • The NFS client module plays the role described for the client module in our architectural model, supplying an interface suitable for use by conventional application programs. But unlike our model client module, it emulates the semantics of the standard UNIX file system primitives precisely and is integrated with the UNIX kernel. It is integrated with the kernel and not supplied as a library for loading into client processes so that:

- user programs can access files via UNIX system calls without recompilation or reloading;
- a single client module serves all of the user-level processes, with a shared cache of recently used blocks (described below);

Figure 12.9 NFS server operations (NFS version 3 protocol, simplified)

lookup(dirfh, name) → *fh, attr*	Returns file handle and attributes for the file *name* in the directory *dirfh*.
create(dirfh, name, attr) → *newfh, attr*	Creates a new file *name* in directory *dirfh* with attributes *attr* and returns the new file handle and attributes.
remove(dirfh, name) → *status*	Removes file *name* from directory *dirfh*.
getattr(fh) → *attr*	Returns file attributes of file *fh*. (Similar to the UNIX *stat* system call.)
setattr(fh, attr) → *attr*	Sets the attributes (mode, user ID, group ID, size, access time and modify time of a file). Setting the size to 0 truncates the file.
read(fh, offset, count) → *attr, data*	Returns up to *count* bytes of data from a file starting at *offset*. Also returns the latest attributes of the file.
write(fh, offset, count, data) → *attr*	Writes *count* bytes of data to a file starting at *offset*. Returns the attributes of the file after the write has taken place.
rename(dirfh, name, todirfh, toname) → *status*	Changes the name of file *name* in directory *dirfh* to *toname* in directory *todirfh*.
link(newdirfh, newname, fh) → *status*	Creates an entry *newname* in the directory *newdirfh* that refers to the file *or* directory *fh*.
symlink(newdirfh, newname, string) → *status*	Creates an entry *newname* in the directory *newdirfh* of type *symbolic link* with the value *string*. The server does not interpret the *string* but makes a symbolic link file to hold it.
readlink(fh) → *string*	Returns the string that is associated with the symbolic link file identified by *fh*.
mkdir(dirfh, name, attr) → *newfh, attr*	Creates a new directory *name* with attributes *attr* and returns the new file handle and attributes.
rmdir(dirfh, name) → *status*	Removes the empty directory *name* from the parent directory *dirfh*. Fails if the directory is not empty.
readdir(dirfh, cookie, count) → *entries*	Returns up to *count* bytes of directory entries from the directory *dirfh*. Each entry contains a file name, a file handle and an opaque pointer to the next directory entry, called a *cookie*. The *cookie* is used in subsequent *readdir* calls to start reading from the following entry. If the value of *cookie* is 0, reads from the first entry in the directory.
statfs(fh) → *fsstats*	Returns file system information (such as block size, number of free blocks and so on) for the file system containing a file *fh*.

- the encryption key used to authenticate user IDs passed to the server (see below) can be retained in the kernel, preventing impersonation by user-level clients.

The NFS client module cooperates with the virtual file system in each client machine. It operates in a similar manner to the conventional UNIX file system, transferring blocks of files to and from the server and caching the blocks in the local memory whenever possible. It shares the same buffer cache that is used by the local input-output system.

But since several clients in different host machines may simultaneously access the same remote file, a new and significant cache consistency problem arises.

Access control and authentication • Unlike the conventional UNIX file system, the NFS server is stateless and does not keep files open on behalf of its clients. So the server must check the user's identity against the file's access permission attributes afresh on each request, to see whether the user is permitted to access the file in the manner requested. The Sun RPC protocol requires clients to send user authentication information (for example, the conventional UNIX 16-bit user ID and group ID) with each request and this is checked against the access permission in the file attributes. These additional parameters are not shown in our overview of the NFS protocol in Figure 12.9; they are supplied automatically by the RPC system.

In its simplest form, there is a security loophole in this access-control mechanism. An NFS server provides a conventional RPC interface at a well-known port on each host and any process can behave as a client, sending requests to the server to access or update a file. The client can modify the RPC calls to include the user ID of any user, impersonating the user without their knowledge or permission. This security loophole has been closed by the use of an option in the RPC protocol for the DES encryption of the user's authentication information. More recently, Kerberos has been integrated with Sun NFS to provide a stronger and more comprehensive solution to the problems of user authentication and security; we describe this below.

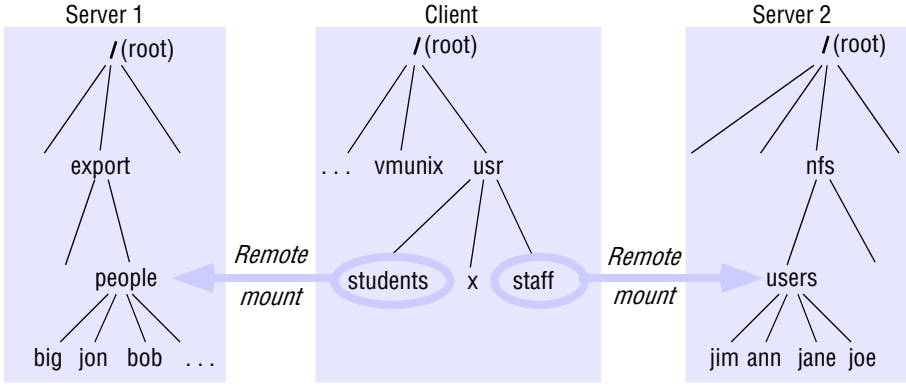
NFS server interface • A simplified representation of the RPC interface provided by NFS version 3 servers (defined in RFC 1813 [Callaghan *et al.* 1995]) is shown in Figure 12.9. The NFS file access operations *read*, *write*, *getattr* and *setattr* are almost identical to the *Read*, *Write*, *GetAttributes* and *SetAttributes* operations defined for our flat file service model (Figure 12.6). The *lookup* operation and most of the other directory operations defined in Figure 12.9 are similar to those in our directory service model (Figure 12.7).

The file and directory operations are integrated in a single service; the creation and insertion of file names in directories is performed by a single *create* operation, which takes the text name of the new file and the file handle for the target directory as arguments. The other NFS operations on directories are *create*, *remove*, *rename*, *link*, *symlink*, *readlink*, *mkdir*, *rmdir*, *readdir* and *statfs*. They resemble their UNIX counterparts with the exception of *readdir*, which provides a representation-independent method for reading the contents of directories, and *statfs,* which gives the status information on remote file systems.

Mount service • The mounting of subtrees of remote filesystems by clients is supported by a separate *mount service* process that runs at user level on each NFS server computer. On each server, there is a file with a well-known name (*/etc/exports*) containing the names of local filesystems that are available for remote mounting. An access list is associated with each filesystem name indicating which hosts are permitted to mount the filesystem.

Clients use a modified version of the UNIX *mount* command to request mounting of a remote filesystem, specifying the remote host's name, the pathname of a directory in the remote filesystem and the local name with which it is to be mounted. The remote directory may be any subtree of the required remote filesystem, enabling clients to mount any part of the remote filesystem. The modified *mount* command communicates

Figure 12.10 Local and remote filesystems accessible on an NFS client



Note: The file system mounted at */usr/students* in the client is actually the subtree located at */export/people* in Server 1; the filesystem mounted at */usr/staff* in the client is actually the subtree located at /nfs/users in Server 2.

with the mount service process on the remote host using a *mount protocol*. This is an RPC protocol and includes an operation that takes a directory pathname and returns the file handle of the specified directory if the client has access permission for the relevant filesystem. The location (IP address and port number) of the server and the file handle for the remote directory are passed on to the VFS layer and the NFS client.

Figure 12.10 illustrates a *Client* with two remotely mounted file stores. The nodes *people* and *users* in filesystems at *Server 1* and *Server 2* are mounted over nodes *students* and *staff* in *Client*'s local file store. The meaning of this is that programs running at *Client* can access files at *Server 1* and *Server 2* by using pathnames such as */usr/students/jon* and */usr/staff/ann*.

Remote filesystems may be *hard-mounted* or *soft-mounted* in a client computer. When a user-level process accesses a file in a filesystem that is hard-mounted, the process is suspended until the request can be completed, and if the remote host is unavailable for any reason the NFS client module continues to retry the request until it is satisfied. Thus in the case of a server failure, user-level processes are suspended until the server restarts and then they continue just as though there had been no failure. But if the relevant filesystem is soft-mounted, the NFS client module returns a failure indication to user-level processes after a small number of retries. Properly constructed programs will then detect the failure and take appropriate recovery or reporting actions. But many UNIX utilities and applications do not test for the failure of file access operations, and these behave in unpredictable ways in the case of failure of a soft-mounted filesystem. For this reason, many installations use hard mounting exclusively, with the consequence that programs are unable to recover gracefully when an NFS server is unavailable for a significant period.

Pathname translation • UNIX file systems translate multi-part file pathnames to i-node references in a step-by-step process whenever the *open*, *creat* or *stat* system calls are used. In NFS, pathnames cannot be translated at a server, because the name may cross a

'mount point' at the client – directories holding different parts of a multi-part name may reside in filesystems at different servers. So pathnames are parsed, and their translation is performed in an iterative manner by the client. Each part of a name that refers to a remote-mounted directory is translated to a file handle using a separate *lookup* request to the remote server.

The *lookup* operation looks for a single part of a pathname in a given directory and returns the corresponding file handle and file attributes. The file handle returned in the previous step is used as a parameter in the next *lookup* step. Since file handles are opaque to NFS client code, the virtual file system is responsible for resolving file handles to a local or a remote directory and performing the necessary indirection when it references a local mount point. Caching of the results of each step in pathname translations alleviates the apparent inefficiency of this process, taking advantage of locality of reference to files and directories; users and programs typically access files in only one or a small number of directories.

Automounter • The automounter was added to the UNIX implementation of NFS in order to mount a remote directory dynamically whenever an 'empty' mount point is referenced by a client. The original implementation of the automounter ran as a user-level UNIX process in each client computer. Later versions (called *autofs*) were implemented in the kernel for Solaris and Linux. We describe the original version here.

The automounter maintains a table of mount points (pathnames) with a reference to one or more NFS servers listed against each. It behaves like a local NFS server at the client machine. When the NFS client module attempts to resolve a pathname that includes one of these mount points, it passes to the local automounter a *lookup*() request that locates the required filesystem in its table and sends a 'probe' request to each server listed. The filesystem on the first server to respond is then mounted at the client using the normal mount service. The mounted filesystem is linked to the mount point using a symbolic link, so that accesses to it will not result in further requests to the automounter. File access then proceeds in the normal way without further reference to the automounter unless there are no references to the symbolic link for several minutes. In the latter case, the automounter unmounts the remote filesystem.

The later kernel implementations replaced the symbolic links with real mounts, avoiding some problems that arose with applications that cached the temporary pathnames used in user-level automounters [Callaghan 1999].

A simple form of read-only replication can be achieved by listing several servers containing identical copies of a filesystem or file subtree against a name in the automounter table. This is useful for heavily used file systems that change infrequently, such as UNIX system binaries. For example, copies of the */usr/lib* directory and its subtree might be held on more than one server. On the first occasion that a file in */usr/lib* is opened at a client, all of the servers will be sent probe messages, and the first to respond will be mounted at the client. This provides a limited degree of fault tolerance and load balancing, since the first server to respond will be one that has not failed and is likely to be one that is not heavily occupied with servicing other requests.

Server caching • Caching in both the client and the server computer are indispensable features of NFS implementations in order to achieve adequate performance.

In conventional UNIX systems, file pages, directories and file attributes that have been read from disk are retained in a main memory *buffer cache* until the buffer space

is required for other pages. If a process then issues a read or a write request for a page that is already in the cache, it can be satisfied without another disk access. *Read-ahead* anticipates read accesses and fetches the pages following those that have most recently been read, and *delayed-write* optimizes writes: when a page has been altered (by a write request), its new contents are written to disk only when the buffer page is required for another page. To guard against loss of data in a system crash, the UNIX *sync* operation flushes altered pages to disk every 30 seconds. These caching techniques work in a conventional UNIX environment because all read and write requests issued by user-level processes pass through a single cache that is implemented in the UNIX kernel space. The cache is always kept up-to-date, and file accesses cannot bypass the cache.

NFS servers use the cache at the server machine just as it is used for other file accesses. The use of the server's cache to hold recently read disk blocks does not raise any consistency problems; but when a server performs write operations, extra measures are needed to ensure that clients can be confident that the results of the write operations are persistent, even when server crashes occur. In version 3 of the NFS protocol, the *write* operation offers two options for this (not shown in Figure 12.9):

1. Data in *write* operations received from clients is stored in the memory cache at the server *and* written to disk before a reply is sent to the client. This is called *write-through* caching. The client can be sure that the data is stored persistently as soon as the reply has been received.

2. Data in *write* operations is stored only in the memory cache. It will be written to disk when a *commit* operation is received for the relevant file. The client can be sure that the data is persistently stored only when a reply to a *commit* operation for the relevant file has been received. Standard NFS clients use this mode of operation, issuing a *commit* whenever a file that was open for writing is closed.

Commit is an additional operation provided in version 3 of the NFS protocol; it was added to overcome a performance bottleneck caused by the write-through mode of operation in servers that receive large numbers of *write* operations.

The requirement for write-through in distributed file systems is an instance of the independent failure modes discussed in Chapter 1 – clients continue to operate when a server fails, and application programs may take actions on the assumption that the results of previous writes are committed to disk storage. This is unlikely to occur in the case of local file updates, because the failure of a local file system is almost certain to result in the failure of all the application processes running on the same computer.

Client caching • The NFS client module caches the results of *read*, *write*, *getattr*, *lookup* and *readdir* operations in order to reduce the number of requests transmitted to servers. Client caching introduces the potential for different versions of files or portions of files to exist in different client nodes, because writes by a client do not result in the immediate updating of cached copies of the same file in other clients. Instead, clients are responsible for polling the server to check the currency of the cached data that they hold.

A timestamp-based method is used to validate cached blocks before they are used. Each data or metadata item in the cache is tagged with two timestamps:

Tc is the time when the cache entry was last validated.

Tm is the time when the block was last modified at the server.

A cache entry is valid at time T if $T - Tc$ is less than a freshness interval t, or if the value for Tm recorded at the client matches the value of Tm at the server (that is, the data has not been modified at the server since the cache entry was made). Formally, the validity condition is:

$$(T - Tc < t) \vee (Tm_{client} = Tm_{server})$$

The selection of a value for t involves a compromise between consistency and efficiency. A very short freshness interval will result in a close approximation to one-copy consistency, at the cost of a relatively heavy load of calls to the server to check the value of Tm_{server}. In Sun Solaris clients, t is set adaptively for individual files to a value in the range 3 to 30 seconds, depending on the frequency of updates to the file. For directories the range is 30 to 60 seconds, reflecting the lower risk of concurrent updates.

There is one value of Tm_{server} for all the data blocks in a file and another for the file attributes. Since NFS clients cannot determine whether a file is being shared or not, the validation procedure must be used for all file accesses. A validity check is performed whenever a cache entry is used. The first half of the validity condition can be evaluated without access to the server. If it is true, then the second half need not be evaluated; if it is false, the current value of Tm_{server} is obtained (by means of a *getattr* call to the server) and compared with the local value Tm_{client}. If they are the same, then the cache entry is taken to be valid and the value of T_c for that cache entry is updated to the current time. If they differ, then the cached data has been updated at the server and the cache entry is invalidated, resulting in a request to the server for the relevant data.

Several measures are used to reduce the traffic of *getattr* calls to the server:

- Whenever a new value of Tm_{server} is received at a client, it is applied to all cache entries derived from the relevant file.
- The current attribute values are sent 'piggybacked' with the results of every operation on a file, and if the value of Tm_{server} has changed the client uses it to update the cache entries relating to the file.
- The adaptive algorithm for setting freshness interval t outlined above reduces the traffic considerably for most files.

The validation procedure does not guarantee the same level of consistency of files that is provided in conventional UNIX systems, since recent updates are not always visible to clients sharing a file. There are two sources of time lag; the delay after a write before the updated data leaves the cache in the updating client's kernel and the three-second 'window' for cache validation. Fortunately, most UNIX applications do not depend critically upon the synchronization of file updates, and few difficulties have been reported from this source.

Writes are handled differently. When a cached page is modified it is marked as 'dirty' and is scheduled to be flushed to the server asynchronously. Modified pages are flushed when the file is closed or a *sync* occurs at the client, and they are flushed more frequently if bio-daemons are in use (see below). This does not provide the same persistence guarantee as the server cache, but it emulates the behaviour for local writes.

To implement read-ahead and delayed-write, the NFS client needs to perform some reads and writes asynchronously. This is achieved in UNIX implementations of

NFS by the inclusion of one or more *bio–daemon* processes at each client. (*Bio* stands for block input-output; the term *daemon* is often used to refer to user-level processes that perform system tasks.) The role of the bio-daemons is to perform read-ahead and delayed-write operations. A bio-daemon is notified after each read request, and it requests the transfer of the following file block from the server to the client cache. In the case of writing, the bio-daemon will send a block to the server whenever a block has been filled by a client operation. Directory blocks are sent whenever a modification has occurred.

Bio-daemon processes improve performance, ensuring that the client module does not block waiting for *read*s to return or *write*s to commit at the server. They are not a logical requirement, since in the absence of read-ahead, a *read* operation in a user process will trigger a synchronous request to the relevant server, and the results of *write*s in user processes will be transferred to the server when the relevant file is closed or when the virtual file system at the client performs a *sync* operation.

Other optimizations • The Sun file system is based on the UNIX BSD Fast File System which uses 8-kbyte disk blocks, resulting in fewer file system calls for sequential file access than previous UNIX systems. The UDP packets used for the implementation of Sun RPC are extended to 9 kilobytes, enabling an RPC call containing an entire block as an argument to be transferred in a single packet and minimizing the effect of network latency when reading files sequentially. In NFS version 3, there is no limit on the maximum size of file blocks that can be handled in *read* and *write* operations; clients and servers can negotiate sizes larger than 8 kbytes if both are able to handle them.

As mentioned above, the file status information cached at clients must be updated at least every three seconds for active files. To reduce the consequential server load resulting from *getattr* requests, all operations that refer to files or directories are taken as implicit *getattr* requests, and the current attribute values are 'piggybacked' along with the other results of the operation.

Securing NFS with Kerberos • In Section 11.6.2 we described the Kerberos authentication system developed at MIT, which has become an industry standard for securing intranet servers against unauthorized access and imposter attacks. The security of NFS implementations has been strengthened by the use of the Kerberos scheme to authenticate clients. In this subsection, we describe the 'Kerberization' of NFS as carried out by the designers of Kerberos.

In the original standard implementation of NFS, the user's identity is included in each request in the form of an unencrypted numeric identifier. (The identifier can be encrypted in later versions of NFS.) NFS does not take any further steps to check the authenticity of the identifier supplied. This implies a high degree of trust in the integrity of the client computer and its software by NFS, whereas the aim of Kerberos and other authentication-based security systems is to reduce to a minimum the range of components in which trust is assumed. Essentially, when NFS is used in a 'Kerberized' environment it should accept requests only from clients whose identity can be shown to have been authenticated by Kerberos.

One obvious solution considered by the Kerberos developers was to change the nature of the credentials required by NFS to be a full-blown Kerberos ticket and authenticator. But because NFS is implemented as a stateless server, each individual file access request is handled on its face value and the authentication data would have to be

included in each request. This was considered unacceptably expensive in terms of the time required to perform the necessary encryptions and because it would have entailed adding the Kerberos client library to the kernel of all workstations.

Instead, a hybrid approach was adopted in which the NFS mount server is supplied with full Kerberos authentication data for the users when their home and root filesystems are mounted. The results of this authentication, including the user's conventional numerical identifier and the address of the client computer, are retained by the server with the mount information for each filesystem. (Although the NFS server does not retain state relating to individual client processes, it does retain the current mounts at each client computer.)

On each file access request, the NFS server checks the user identifier and the sender's address and grants access only if they match those stored at the server for the relevant client at mount time. This hybrid approach involves only minimal additional cost and is safe against most forms of attack, provided that only one user at a time can log in to each client computer. At MIT, the system is configured so that this is the case. Recent NFS implementations include Kerberos authentication as one of several options for authentication, and sites that also run Kerberos servers are advised to use this option.

Performance • Early performance figures reported by Sandberg [1987] showed that the use of NFS did not normally impose a performance penalty in comparison with access to files stored on local disks. He identified two remaining problem areas:

- frequent use of the *getattr* call in order to fetch timestamps from servers for cache validation;
- relatively poor performance of the *write* operation because write-through was used at the server.

He noted that writes are relatively infrequent in typical UNIX workloads (about 5% of all calls to the server), and the cost of write-through is therefore tolerable except when large files are written to the server. Further, the version of NFS that he tested did not include the *commit* mechanism outlined above, which has resulted in a substantial improvement in write performance in current versions. His results also show that the *lookup* operation accounts for almost 50% of server calls. This is a consequence of the step-by-step pathname translation method necessitated by UNIX's file-naming semantics.

Measurements are taken regularly by Sun and other NFS implementors using an updated version of an exhaustive set of benchmark programs known as LADDIS [Keith and Wittle 1993]. Current and past results are available at the SPEC web site [www.spec.org]. Performance is summarized there for NFS server implementations from many vendors and different hardware configurations. Single-CPU implementations based on PC hardware but with dedicated operating systems achieve throughputs in excess of 12,000 server operations per second and large multi-processor configurations with many disks and controllers have achieved throughputs of up to 300,000 server operations per second. These figures indicate that NFS offers a very effective solution to distributed storage needs in intranets of most sizes and types of use, ranging for example from a traditional UNIX load of development by several hundred software engineers to a battery of web servers serving material from an NFS server.

NFS summary • Sun NFS closely follows our abstract model. The resulting design provides good location and access transparency if the NFS mount service is used properly to produce similar name spaces at all clients. NFS supports heterogeneous hardware and operating systems. The NFS server implementation is stateless, enabling clients and servers to resume execution after a failure without the need for any recovery procedures. Migration of files or filesystems is not supported, except at the level of manual intervention to reconfigure mount directives after the movement of a filesystem to a new location.

The performance of NFS is much enhanced by the caching of file blocks at each client computer. This is important for the achievement of satisfactory performance but results in some deviation from strict UNIX one-copy file update semantics.

The other design goals of NFS and the extent to which they have been achieved are discussed below.

Access transparency: The NFS client module provides an application programming interface to local processes that is identical to the local operating system's interface. Thus in a UNIX client, accesses to remote files are performed using the normal UNIX system calls. No modifications to existing programs are required to enable them to operate correctly with remote files.

Location transparency: Each client establishes a file name space by adding mounted directories in remote filesystems to its local name space. File systems have to be *exported* by the node that holds them and *remote-mounted* by a client before they can be accessed by processes running in the client (see Figure 12.10). The point in a client's name hierarchy at which a remote-mounted file system appears is determined by the client; NFS does not enforce a single network-wide file name space – each client sees a set of remote filesystems that is determined locally, and remote files may have different pathnames on different clients, but a uniform name space can be established with appropriate configuration tables in each client, achieving the goal of location transparency.

Mobility transparency: Filesystems (in the UNIX sense, that is, subtrees of files) may be moved between servers, but the remote mount tables in each client must then be updated separately to enable the clients to access the filesystems in their new locations, thus migration transparency is not fully achieved by NFS.

Scalability: The published performance figures show that NFS servers can be built to handle very large real-world loads in an efficient and cost-effective manner. The performance of a single server can be increased by the addition of processors, disks and controllers. When the limits of that process are reached, additional servers must be installed and the filesystems must be reallocated between them. The effectiveness of that strategy is limited by the existence of 'hot spot' files – single files that are accessed so frequently that the server reaches a performance limit. When loads exceed the maximum performance available with that strategy, a distributed file system that supports replication of updatable files (such as Coda, described in Chapter 18), or one such as AFS that reduces the protocol traffic by the caching of whole files, may offer a better solution. We discuss other approaches to scalability in Section 12.5.

File replication: Read-only file stores can be replicated on several NFS servers, but NFS does not support file replication with updates. The Sun Network Information Service (NIS) is a separate service available for use with NFS that supports the replication of simple databases organized as key-value pairs (for example, the UNIX system files */etc/passwd* and */etc/hosts*). It manages the distribution of updates and accesses to the replicated files based on a simple master–slave replication model (also known as the *primary copy* model, discussed further in Chapter 18) with provision for the replication of part or all of the database at each site. NIS provides a shared repository for system information that changes infrequently and does not require updates to occur simultaneously at all sites.

Hardware and operating system heterogeneity: NFS has been implemented for almost every known operating system and hardware platform and is supported by a variety of filing systems.

Fault tolerance: The stateless and idempotent nature of the NFS file access protocol ensures that the failure modes observed by clients when accessing remote files are similar to those for local file access. When a server fails, the service that it provides is suspended until the server is restarted, but once it has been restarted user-level client processes proceed from the point at which the service was interrupted, unaware of the failure (except in the case of access to *soft-mounted* remote file systems). In practice, hard mounting is used in most instances, and this tends to impede application programs handling server failures gracefully.

The failure of a client computer or a user-level process in a client has no effect on any server that it may be using, since servers hold no state on behalf of their clients.

Consistency: We have described the update behaviour in some detail. It provides a close approximation to one-copy semantics and meets the needs of the vast majority of applications, but the use of file sharing via NFS for communication or close coordination between processes on different computers cannot be recommended.

Security: The need for security in NFS emerged with the connection of most intranets to the Internet. The integration of Kerberos with NFS was a major step forward. Other recent developments include the option to use a secure RPC implementation (RPCSEC_GSS, documented in RFC 2203 [Eisler *et al.* 1997]) for authentication and to ensure the privacy and security of the data transmitted with read and write operations. Installations that have not deployed these mechanisms abound, though, and they are insecure.

Efficiency: The measured performance of several implementations of NFS and its widespread adoption for use in situations that generate very heavy loads are clear indications of the efficiency with which the NFS protocol can be implemented.

12.4 Case study: The Andrew File System

Like NFS, AFS provides transparent access to remote shared files for UNIX programs running on workstations. Access to AFS files is via the normal UNIX file primitives, enabling existing UNIX programs to access AFS files without modification or recompilation. AFS is compatible with NFS. AFS servers hold 'local' UNIX files, but the filing system in the servers is NFS-based, so files are referenced by NFS-style file handles rather than i-node numbers, and the files may be remotely accessed via NFS.

AFS differs markedly from NFS in its design and implementation. The differences are primarily attributable to the identification of scalability as the most important design goal. AFS is designed to perform well with larger numbers of active users than other distributed file systems. The key strategy for achieving scalability is the caching of whole files in client nodes. AFS has two unusual design characteristics:

Whole-file serving: The entire contents of directories and files are transmitted to client computers by AFS servers (in AFS-3, files larger than 64 kbytes are transferred in 64-kbyte chunks).

Whole-file caching: Once a copy of a file or a chunk has been transferred to a client computer it is stored in a cache on the local disk. The cache contains several hundred of the files most recently used on that computer. The cache is permanent, surviving reboots of the client computer. Local copies of files are used to satisfy clients' *open* requests in preference to remote copies whenever possible.

Scenario • Here is a simple scenario illustrating the operation of AFS:

1. When a user process in a client computer issues an *open* system call for a file in the shared file space and there is not a current copy of the file in the local cache, the server holding the file is located and is sent a request for a copy of the file.
2. The copy is stored in the local UNIX file system in the client computer. The copy is then *opened* and the resulting UNIX file descriptor is returned to the client.
3. Subsequent *read*, *write* and other operations on the file by processes in the client computer are applied to the local copy.
4. When the process in the client issues a *close* system call, if the local copy has been updated its contents are sent back to the server. The server updates the file contents and the timestamps on the file. The copy on the client's local disk is retained in case it is needed again by a user-level process on the same workstation.

We discuss the observed performance of AFS below, but we can make some general observations and predictions here based on the design characteristics described above:

- For shared files that are infrequently updated (such as those containing the code of UNIX commands and libraries) and for files that are normally accessed by only a single user (such as most of the files in a user's home directory and its subtree), locally cached copies are likely to remain valid for long periods – in the first case because they are not updated and in the second because if they are updated, the updated copy will be in the cache on the owner's workstation. These classes of file account for the overwhelming majority of file accesses.

- The local cache can be allocated a substantial proportion of the disk space on each workstation – say, 100 megabytes. This is normally sufficient for the establishment of a working set of the files used by one user. The provision of sufficient cache storage for the establishment of a working set ensures that files in regular use on a given workstation are normally retained in the cache until they are needed again.
- The design strategy is based on some assumptions about average and maximum file size and locality of reference to files in UNIX systems. These assumptions are derived from observations of typical UNIX workloads in academic and other environments [Satyanarayanan 1981, Ousterhout *et al.* 1985, Floyd 1986]. The most important observations are:
  - Files are small; most are less than 10 kilobytes in size.
  - Read operations on files are much more common than writes (about six times more common).
  - Sequential access is common, and random access is rare.
  - Most files are read and written by only one user. When a file is shared, it is usually only one user who modifies it.
  - Files are referenced in bursts. If a file has been referenced recently, there is a high probability that it will be referenced again in the near future.

  These observations were used to guide the design and optimization of AFS, *not* to restrict the functionality seen by users.
- AFS works best with the classes of file identified in the first point above. There is one important type of file that does not fit into any of these classes – databases are typically shared by many users and are often updated quite frequently. The designers of AFS have explicitly excluded the provision of storage facilities for databases from their design goals, stating that the constraints imposed by different naming structures (that is, content-based access) and the need for fine-grained data access, concurrency control and atomicity of updates make it difficult to design a distributed database system that is also a distributed file system. They argue that the provision of facilities for distributed databases should be addressed separately [Satyanarayanan 1989a].

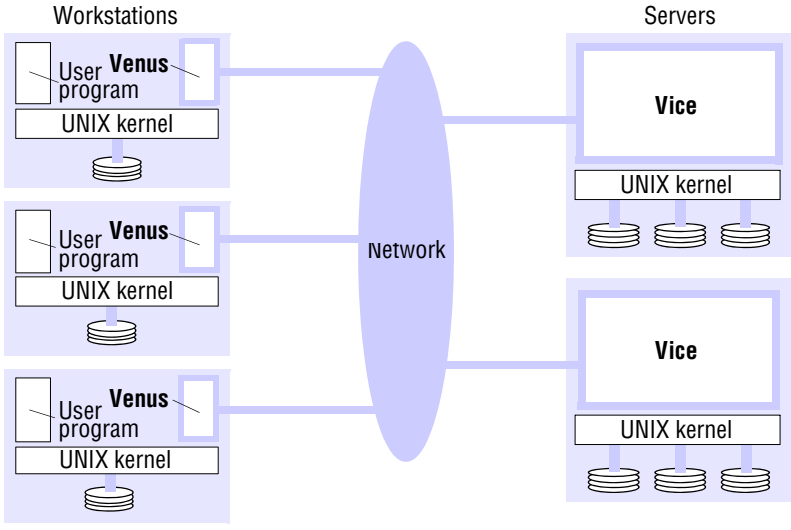
12.4.1 Implementation

The above scenario illustrates AFS's operation but leaves many questions about its implementation unanswered. Among the most important are:

- How does AFS gain control when an *open* or *close* system call referring to a file in the shared file space is issued by a client?
- How is the server holding the required file located?
- What space is allocated for cached files in workstations?
- How does AFS ensure that the cached copies of files are up-to-date when files may be updated by several clients?

We answer these questions below.

Figure 12.11 Distribution of processes in the Andrew File System



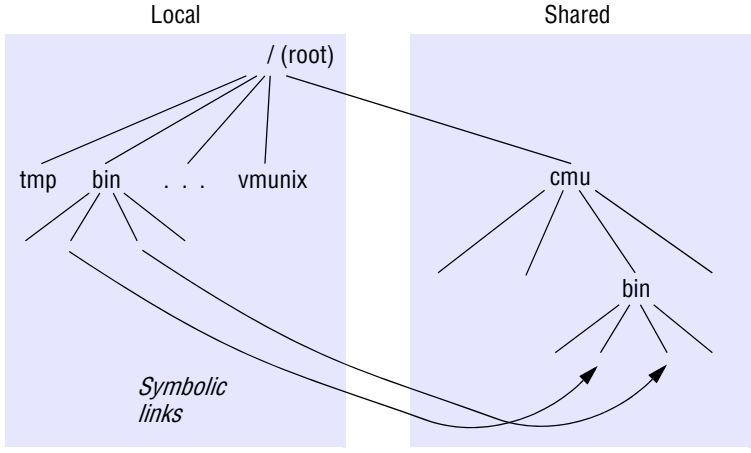
AFS is implemented as two software components that exist as UNIX processes called *Vice* and *Venus*. Figure 12.11 shows the distribution of Vice and Venus processes. Vice is the name given to the server software that runs as a user-level UNIX process in each server computer, and Venus is a user-level process that runs in each client computer and corresponds to the client module in our abstract model.

The files available to user processes running on workstations are either *local* or *shared*. Local files are handled as normal UNIX files. They are stored on a workstation's disk and are available only to local user processes. Shared files are stored on servers, and copies of them are cached on the local disks of workstations. The name space seen by user processes is illustrated in Figure 12.12. It is a conventional UNIX directory hierarchy, with a specific subtree (called *cmu*) containing all of the shared files. This splitting of the file name space into local and shared files leads to some loss of location transparency, but this is hardly noticeable to users other than system administrators. Local files are used only for temporary files (*/tmp*) and processes that are essential for workstation startup. Other standard UNIX files (such as those normally found in */bin*, */lib* and so on) are implemented as symbolic links from local directories to files held in the shared space. Users' directories are in the shared space, enabling users to access their files from any workstation.

The UNIX kernel in each workstation and server is a modified version of BSD UNIX. The modifications are designed to intercept *open*, *close* and some other file system calls when they refer to files in the shared name space and pass them to the Venus process in the client computer (illustrated in Figure 12.13). One other kernel modification is included for performance reasons, and this is described later.

One of the file partitions on the local disk of each workstation is used as a cache, holding the cached copies of files from the shared space. Venus manages the cache, removing the least recently used files when a new file is acquired from a server to make

Figure 12.12 File name space seen by clients of AFS

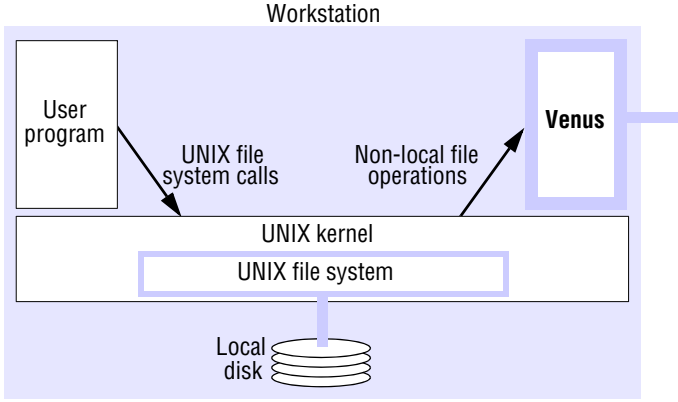


the required space if the partition is full. The workstation cache is usually large enough to accommodate several hundred average-sized files, rendering the workstation largely independent of the Vice servers once a working set of the current user's files and frequently used system files has been cached.

AFS resembles the abstract file service model described in Section 12.2 in these respects:

- A flat file service is implemented by the Vice servers, and the hierarchic directory structure required by UNIX user programs is implemented by the set of Venus processes in the workstations.
- Each file and directory in the shared file space is identified by a unique, 96-bit file identifier (*fid*) similar to a UFID. The Venus processes translate the pathnames issued by clients to *fid*s.

Figure 12.13 System call interception in AFS



Files are grouped into *volumes* for ease of location and movement. Volumes are generally smaller than the UNIX filesystems, which are the unit of file grouping in NFS. For example, each user's personal files are generally located in a separate volume. Other volumes are allocated for system binaries, documentation and library code.

The representation of *fid*s includes the volume number for the volume containing the file (*cf.* the *file group identifier* in UFIDs), an NFS file handle identifying the file within the volume (*cf.* the *file number* in UFIDs) and a *uniquifier* to ensure that file identifiers are not reused:

32 bits	*32 bits*	*32 bits*
Volume number	File handle	Uniquifier

User programs use conventional UNIX pathnames to refer to files, but AFS uses *fid*s in the communication between the Venus and Vice processes. The Vice servers accept requests only in terms of *fid*s. Venus translates the pathnames supplied by clients into *fid*s using a step-by-step lookup to obtain the information from the file directories held in the Vice servers.

Figure 12.14 describes the actions taken by Vice, Venus and the UNIX kernel when a user process issues each of the system calls mentioned in our outline scenario above. The *callback promise* mentioned here is a mechanism for ensuring that cached copies of files are updated when another client closes the same file after updating it. This mechanism is discussed in the next section.

12.4.2 Cache consistency

When Vice supplies a copy of a file to a Venus process it also provides a *callback promise* – a token issued by the Vice server that is the custodian of the file, guaranteeing that it will notify the Venus process when any other client modifies the file. Callback promises are stored with the cached files on the workstation disks and have two states: *valid* or *cancelled*. When a server performs a request to update a file it notifies all of the Venus processes to which it has issued callback promises by sending a *callback* to each – a callback is a remote procedure call from a server to a Venus process. When the Venus process receives a callback, it sets the *callback promise* token for the relevant file to *cancelled*.

Whenever Venus handles an *open* on behalf of a client, it checks the cache. If the required file is found in the cache, then its token is checked. If its value is *cancelled*, then a fresh copy of the file must be fetched from the Vice server, but if the token is *valid*, then the cached copy can be opened and used without reference to Vice.

When a workstation is restarted after a failure or a shutdown, Venus aims to retain as many as possible of the cached files on the local disk, but it cannot assume that the callback promise tokens are correct, since some callbacks may have been missed. Before the first use of each cached file or directory after a restart, Venus therefore generates a cache validation request containing the file modification timestamp to the server that is the custodian of the file. If the timestamp is current, the server responds with *valid* and the token is reinstated. If the timestamp shows that the file is out of date, then the server responds with *cancelled* and the token is set to *cancelled*. Callbacks must be renewed

Figure 12.14 Implementation of file system calls in AFS

User process	*UNIX kernel*	*Venus*	*Net*	*Vice*
open(FileName, mode)	If *FileName* refers to a file in shared file space, pass the request to Venus.	Check list of files in local cache. If not present or there is no valid *callback promise*, send a request for the file to the Vice server that is custodian of the volume containing the file.	→	Transfer a copy of the file and a *callback promise* to the workstation. Log the callback promise.
	Open the local file and return the file descriptor to the application.	Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.	←	
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy.			
close(FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice server that is the custodian of the file.	→	Replace the file contents and send a *callback* to all other clients holding *callback promises* on the file.

before an *open* if a time T (typically on the order of a few minutes) has elapsed since the file was cached without communication from the server. This is to deal with possible communication failures, which can result in the loss of callback messages.

This callback-based mechanism for maintaining cache consistency was adopted as offering the most scalable approach, following the evaluation in the prototype (AFS-1) of a timestamp-based mechanism similar to that used in NFS. In AFS-1, a Venus process holding a cached copy of a file interrogates the Vice process on each *open* to determine whether the timestamp on the local copy agrees with that on the server. The callback-based approach is more scalable because it results in communication between client and server and activity in the server only when the file has been updated, whereas the timestamp approach results in a client-server interaction on each *open*, even when there is a valid local copy. Since the majority of files are not accessed concurrently, and *read* operations predominate over *writes* in most applications, the *callback* mechanism results in a dramatic reduction in the number of client-server interactions.

The callback mechanism used in AFS-2 and later versions of AFS requires Vice servers to maintain some state on behalf of their Venus clients, unlike AFS-1, NFS and our file service model. The client-dependent state required consists of a list of the Venus

Figure 12.15 The main components of the Vice service interface

Fetch(fid) → *attr, data*	Returns the attributes (status) and, optionally, the contents of the file identified by *fid* and records a callback promise on it.
Store(fid, attr, data)	Updates the attributes and (optionally) the contents of a specified file.
Create() → *fid*	Creates a new file and records a callback promise on it.
Remove(fid)	Deletes the specified file.
SetLock(fid, mode)	Sets a lock on the specified file or directory. The mode of the lock may be shared or exclusive. Locks that are not removed expire after 30 minutes.
ReleaseLock(fid)	Unlocks the specified file or directory.
RemoveCallback(fid)	Informs the server that a Venus process has flushed a file from its cache.
BreakCallback(fid)	Call made by a Vice server to a Venus process; cancels the callback promise on the relevant file.

Note: Directory and administrative operations (*Rename*, *Link*, *Makedir*, *Removedir*, *GetTime*, *CheckToken* and so on) are not shown.

processes to which callback promises have been issued for each file. These callback lists must be retained over server failures – they are held on the server disks and are updated using atomic operations.

Figure 12.15 shows the RPC calls provided by AFS servers for operations on files (that is, the interface provided by AFS servers to Venus processes).

Update semantics • The goal of this cache-consistency mechanism is to achieve the best approximation to one-copy file semantics that is practicable without serious performance degradation. A strict implementation of one-copy semantics for UNIX file access primitives would require that the results of each *write* to a file be distributed to all sites holding the file in their cache before any further accesses can occur. This is not practicable in large-scale systems; instead, the callback promise mechanism maintains a well-defined approximation to one-copy semantics.

For AFS-1, the update semantics can be formally stated in very simple terms. For a client *C* operating on a file *F* whose custodian is a server *S*, the following guarantees of currency for the copies of *F* are maintained:

after a successful *open*:	*latest*(*F, S*)
after a failed *open*:	*failure*(*S*)
after a successful *close*:	*updated*(*F, S*)
after a failed *close*:	*failure*(*S*)

where *latest*(*F, S*) denotes a guarantee that the current value of *F* at *C* is the same as the value at *S*, *failure*(*S*) denotes that the *open* or *close* operation has not been performed at

S (and the failure can be detected by *C*), and *updated*(*F, S*) denotes that *C*'s value of *F* has been successfully propagated to *S*.

For AFS-2, the currency guarantee for *open* is slightly weaker, and the corresponding formal statement of the guarantee is more complex. This is because a client may open an old copy of a file after it has been updated by another client. This occurs if a callback message is lost, for example as a result of a network failure. But there is a maximum time, *T*, for which a client can remain unaware of a newer version of a file. Hence we have the following guarantee:

after a successful *open*: *latest*(*F, S, 0*)
or (*lostCallback*(*S, T*) *and inCache*(*F*) *and latest*(*F, S, T*))

where *latest*(*F, S, T*) denotes that the copy of *F* seen by the client is no more than *T* seconds out of date, *lostCallback*(*S, T*) denotes that a callback message from *S* to *C* has been lost at some time during the last *T* seconds, and *inCache*(*F*) indicates that the file *F* was in the cache at *C* before the *open* operation was attempted. The above formal statement expresses the fact that the cached copy of *F* at *C* after an *open* operation is the most recent version in the system or a callback message has been lost (due to a communication failure) and the version that was already in the cache has been used; the cached version will be no more than *T* seconds out of date. (*T* is a system constant representing the interval at which callback promises must be renewed. At most installations, the value of *T* is about 10 minutes.)

In line with its goal – to provide a large-scale, UNIX-compatible distributed file service – AFS does not provide any further mechanism for the control of concurrent updates. The cache consistency algorithm described above comes into action only on *open* and *close* operations. Once a file has been opened, the client may access and update the local copy in any way it chooses without the knowledge of any processes on other workstations. When the file is closed, a copy is returned to the server, replacing the current version.

If clients in different workstations *open*, *write* and *close* the same file concurrently, all but the update resulting from the last *close* will be silently lost (no error report is given). Clients must implement concurrency control independently if they require it. On the other hand, when two client processes in the same workstation open a file, they share the same cached copy and updates are performed in the normal UNIX fashion – block by block.

Although the update semantics differ depending on the locations of the concurrent processes accessing a file and are not precisely the same as those provided by the standard UNIX file system, they are sufficiently close for the vast majority of existing UNIX programs to operate correctly.

12.4.3 Other aspects

AFS introduces several other interesting design developments and refinements that we outline here, together with a summary of performance evaluation results:

UNIX kernel modifications • We have noted that the Vice server is a user-level process running in the server computer and the server host is dedicated to the provision of an

AFS service. The UNIX kernel in AFS hosts is altered so that Vice can perform file operations in terms of file handles instead of the conventional UNIX file descriptors. This is the only kernel modification required by AFS, and it is necessary if Vice is not to maintain any client state (such as file descriptors).

Location database • Each server contains a copy of a fully replicated location database giving a mapping of volume names to servers. Temporary inaccuracies in this database may occur when a volume is moved, but they are harmless because forwarding information is left behind in the server from which the volume is moved.

Threads • The implementations of Vice and Venus make use of a non-preemptive threads package to enable requests to be processed concurrently at both the client (where several user processes may have file access requests in progress concurrently) and the server. In the client, the tables describing the contents of the cache and the volume database are held in memory that is shared between the Venus threads.

Read-only replicas • Volumes containing files that are frequently read but rarely modified, such as the UNIX */bin* and */usr/bin* directories of system commands and */man* directory of manual pages, can be replicated as read-only volumes at several servers. When this is done, there is only one read-write replica and all updates are directed to it. The propagation of the changes to the read-only replicas is performed after the update by an explicit operational procedure. Entries in the location database for volumes that are replicated in this way are one-to-many, and the server for each client request is selected on the bases of server loads and accessibility.

Bulk transfers • AFS transfers files between clients and servers in 64-kilobyte chunks. The use of such a large packet size is an important aid to performance, minimizing the effect of network latency. Thus the design of AFS enables the use of the network to be optimized.

Partial file caching • The need to transfer the entire contents of files to clients even when the application requirement is to read only a small portion of the file is an obvious source of inefficiency. Version 3 of AFS removed this requirement, allowing file data to be transferred and cached in 64-kbyte blocks while still retaining the consistency semantics and other features of the AFS protocol.

Performance • The primary goal of AFS is scalability, so its performance with large numbers of users is of particular interest. Howard *et al.* [1988] give details of extensive comparative performance measurements, which were undertaken using a specially developed *AFS benchmark* that has subsequently been widely used for the evaluation of distributed file systems. Not surprisingly, whole-file caching and the callback protocol led to dramatically reduced loads on the servers. Satyanarayanan [1989a] states that a server load of 40% was measured with 18 client nodes running a standard benchmark, against a load of 100% for NFS running the same benchmark. Satyanarayanan attributes much of the performance advantage of AFS to the reduction in server load deriving from the use of callbacks to notify clients of updates to files, compared with the timeout mechanism used in NFS for checking the validity of pages cached at clients.

Wide area support: • Version 3 of AFS supports multiple administrative cells, each with its own servers, clients, system administrators and users. Each cell is a completely autonomous environment, but a federation of cells can cooperate in presenting users

with a uniform, seamless file name space. The resulting system was widely deployed by the Transarc Corporation, and a detailed survey of the resulting performance usage patterns was published [Spasojevic and Satyanarayanan 1996]. The system was installed on over 1000 servers at over 150 sites. The survey showed cache hit ratios in the range of 96 –98% for accesses to a sample of 32,000 file volumes holding 200 Gbytes of data.

12.5 Enhancements and further developments

Several advances have been made in the design of distributed file systems since the emergence of NFS and AFS. In this section, we describe advances that enhance the performance, availability and scalability of conventional distributed file systems. More radical advances are described elsewhere in the book, including the maintenance of consistency in replicated read-write filesystems to support disconnected operation and high availability in the Bayou and Coda systems (Sections 18.4.2 and 18.4.3) and a highly scalable architecture for the delivery of streams of real-time data with quality guarantees in the Tiger video file server (Section 20.6.1).

NFS enhancements • Several research projects have addressed the need for one-copy update semantics by extending the NFS protocol to include *open* and *close* operations and adding a callback mechanism to enable the server to notify clients of the need to invalidate cache entries. We describe two such efforts here; their results seem to indicate that these enhancements can be accommodated without undue complexity or extra communication costs.

Some recent efforts by Sun and other NFS developers have been directed at making NFS servers more accessible and useful in wide-area networks. While the HTTP protocol supported by web servers offers an effective and highly scalable method for making whole files available to clients throughout the Internet, it is less useful to application programs that require access to portions of large files or those that update portions of files. The WebNFS development (described below) makes it possible for application programs to become clients of NFS servers anywhere in the Internet (using the NFS protocol directly instead of indirectly through a kernel module). This, together with appropriate libraries for Java and other network programming languages, should offer the possibility of implementing Internet applications that share data directly, such as multi-user games or clients of large dynamic databases.

Achieving one-copy update semantics: The stateless server architecture of NFS brought great advantages in terms of robustness and ease of implementation, but it precluded the achievement of precise one-copy update semantics (the effects of concurrent writes by different clients to the same file are not guaranteed to be the same as they would be in a single UNIX system when multiple processes write to a local file). It also prevents the use of callbacks notifying clients of changes to files, and this results in frequent *getattr* requests from clients to check for file modification.

Two research systems have been developed that address these drawbacks. Spritely NFS [Srinivasan and Mogul 1989, Mogul 1994] is a version of the file system developed for the Sprite distributed operating system at Berkeley [Nelson *et al.* 1988]. Spritely NFS is an implementation of the NFS protocol with the addition of *open* and *close* calls.

Clients' modules must send an *open* operation whenever a local user-level process opens a file that is on the server. The parameters of the Sprite *open* operation specify a mode (read, write or both) and include counts of the number of local processes that currently have the file open for reading and for writing. Similarly, when a local process closes a remote file, a *close* operation is sent to the server with updated counts of readers and writers. The server records these numbers in an *open files table* with the IP address and port number of the client.

When the server receives an *open*, it checks the *open files table* for other clients that have the same file open and sends callback messages to those clients instructing them to modify their caching strategy. If the *open* specifies write mode, then it will fail if any other client has the file open for writing. Other clients that have the file open for reading will be instructed to invalidate any locally cached portions of the file.

For *open* operations that specify read mode, the server sends a callback message to any client that is writing, instructing it to stop caching (i.e., to use a strictly write-through mode of operation), and it instructs all clients that are reading to cease caching the file (so that all local read calls result in a request to the server).

These measures result in a file service that maintains the UNIX one-copy update semantics at the expense of carrying some client-related state at the server. They also enable some efficiency gains in the handling of cached writes. If the client-related state is held in volatile memory at the server, it is vulnerable to server crashes. Spritely NFS implements a recovery protocol that interrogates a list of clients that have recently opened files on the server to recover the full *open files table*. The list of clients is stored on disk, is updated relatively infrequently and is 'pessimistic' – it may safely include more clients than those that had files open at the time of a crash. Failed clients may also result in excess entries in the *open files table*, but these entries will be removed when the clients restart.

When Spritely NFS was evaluated against NFS version 2, it showed a modest performance improvement. This was due to the improved caching of writes. Changes in NFS version 3 would probably result in at least as great an improvement, but the results of the Spritely NFS project certainly indicate that it is possible to achieve one-copy update semantics without substantial loss of performance, albeit at the expense of some extra implementation complexity in the client and server modules and the need for a recovery mechanism to restore the state after a server crash.

NQNFS: The NQNFS (Not Quite NFS) project [Macklem 1994] had similar aims to Spritely NFS – to add more precise cache consistency to the NFS protocol and to improve performance through better use of caching. An NQNFS server maintains similar client-related state concerning open files, but it uses leases (Section 5.4.3) to aid recovery after a server crash. The server sets an upper bound on the time for which a client may hold a lease on an open file. If the client wishes to continue beyond that time, it must renew the lease. Callbacks are used in a similar manner to Spritely NFS to request clients to flush their caches when a write request occurs, but if the clients don't reply, the server simply waits until their leases expire before responding to the new write request.

WebNFS: The advent of the Web and Java applets led to the recognition by the NFS development team and others that some Internet applications could benefit from direct

access to NFS servers without many of the overheads associated with the emulation of UNIX file operations included in standard NFS clients.

The aim of WebNFS (described in RFCs 2055 and 2056 [Callaghan 1996a, 1996b]) is to enable web browsers and other applications to access files on an NFS server that 'publishes' them using a *public file handle* relative to a public root directory. This mode of use bypasses the *mount* service and the port mapper service (described in Chapter 5). WebNFS clients interact with an NFS server at a well-known port number (2049). To access files by pathname, they issue *lookup* requests using a public file handle. The public file handle has a well-known value that is interpreted specially by the virtual file system at the server. Because of the high latency of wide-area networks, a *multicomponent* variant of the *lookup* operation is used to look up a multi-part pathname in a single request.

Thus WebNFS enables clients to be written that access portions of files stored in NFS servers at remote sites with minimal setup overheads. There is provision for access control and authentication, but in many cases the client will require only read access to public files, and in that case the authentication option can be turned off. To read a portion of a single file located on an NFS server that supports WebNFS requires the establishment of a TCP connection and two RPC calls – a multicomponent *lookup* and a *read* operation. The size of the block of data read is not limited by the NFS protocol.

For example, a weather service might publish a file on its NFS server containing a large database of frequently updated weather data with a URL such as:

nfs://data.weather.gov/weatherdata/global.data

An interactive WeatherMap client, that displays weather maps could be constructed in Java or any other language that supports a WebNFS procedure library. The client reads only those portions of the */weatherdata/global.data* file that are needed to construct the particular maps requested by a user, whereas a similar application that used HTTP to access a weather data server either would have to transfer the entire database to the client or would require the support of a special-purpose server program to supply it with the data it requires.

NFS version 4: A new version of the NFS protocol was introduced in 2000. The goals of NFS version 4 are described in RFC 2624 [Shepler 1999] and in Brent Callaghan's book [Callaghan 1999]. Like WebNFS, it aims to make it practical to use NFS in wide-area networks and Internet applications. It includes the features of WebNFS, but the introduction of a new protocol also offers an opportunity to make more radical enhancements. (WebNFS was restricted to changes to the server that did not involve the addition of new operations to the protocol.)

NFS version 4 exploits results that have emerged from research in file server design over the past decade, such as the use of callbacks or leases to maintain consistency. NFS version 4 supports on-the-fly recovery from server faults by allowing file systems to be moved to new servers transparently. Scalability is improved by using proxy servers in a manner analogous to their use in the Web.

AFS enhancements • We have mentioned that DCE/DFS, the distributed file system included in the Open Software Foundation's Distributed Computing Environment [www.opengroup.org], was based on the Andrew File System. The design of DCE/DFS goes beyond AFS, particularly in its approach to cache consistency. In AFS, callbacks

are generated only when the server receives a *close* operation for a file that has been updated. DFS adopted a similar strategy to Spritely NFS and NQNFS to generating callbacks as soon as a file is updated. In order to update a file, a client must obtain a *write* token from the server, specifying a range of bytes in the file that the client is permitted to update. When a *write* token is requested, clients holding copies of the same file for reading receive revocation callbacks. Tokens of other types are used to achieve consistency for cached file attributes and other metadata. All tokens have an associated lifetime, and clients must renew them after their lifetime has expired.

Improvements in storage organization • There has been considerable progress in the organization of file data stored on disks. The impetus for much of this work arose from the increased loads and greater reliability that distributed file systems need to support, and they have resulted in file systems with substantially improved performance. The principal results of this work are:

Redundant Arrays of Inexpensive Disks (RAID): This is a mode of storage [Patterson *et al.* 1988, Chen *et al.* 1994] in which data blocks are segmented into fixed-size chunks and stored in 'stripes' across several disks, along with redundant error-correcting codes that enable the data blocks to be reconstructed completely and operation to continue normally in the event of disk failures. RAID also produces considerably better performance than a single disk, because the stripes that make up a block are read and written concurrently.

Log-structured file storage (LFS): Like Spritely NFS, this technique originated in the Sprite distributed operating system project at Berkeley [Rosenblum and Ousterhout 1992]. The authors observed that as larger amounts of main memory became available for caching in file servers, an increased level of cache hits resulted in excellent read performance, but write performance remained mediocre. This arose from the high latencies associated with writing individual data blocks to disk and associated updates to metadata blocks (that is, the blocks known as *i-nodes* that hold file attributes and a vector of pointers to the blocks in a file).

The LFS solution is to accumulate a set of writes in memory and then commit them to disk in large, contiguous, fixed-sized segments. These are called *log segments* because the data and metadata blocks are stored strictly in the order in which they were updated. A log segment is 1 Mbyte or larger in size and is stored in a single disk track, removing the disk head latencies associated with writing individual blocks. Fresh copies of updated data and metadata blocks are always written, requiring the maintenance of a dynamic map (in memory with a persistent backup) pointing to the i-node blocks. Garbage collection of stale blocks is also required, with compaction of 'live' blocks to leave contiguous areas of storage free for the storage of log segments. The latter is a fairly complex process; it is carried out as a background activity by a component called the *cleaner*. Some sophisticated cleaner algorithms have been developed for it based on the results of simulations.

Despite these extra costs, the overall performance gain is outstanding; Rosenblum and Ousterhout measured a write throughput as high as 70% of the available disk bandwidth, compared with less than 10% for a conventional UNIX file system. The log structure also simplifies recovery after server crashes. The Zebra file system [Hartman and Ousterhout 1995], developed as a follow-on to the original LFS

work, combines log-structured writes with a distributed RAID approach – the log segments are subdivided into sections with error-correcting data and written to disks on separate network nodes. Performance four to five times better than that of NFS is claimed for writing large files, with smaller gains for short files.

New design approaches • The availability of high-performance switched networks (such as ATM and switched high-speed Ethernet) have prompted several efforts to provide persistent storage systems that distribute file data in a highly scalable and fault-tolerant manner among many nodes on an intranet, separating the responsibilities for reading and writing data from the responsibilities for managing the metadata and servicing client requests. In the following, we outline two such developments.

These approaches scale better than the more centralized servers that we have described in the preceding sections. They generally demand a high level of trust among the computers that cooperate to provide the service, because they include a fairly low-level protocol for communication with the nodes holding data (somewhat analogous to a 'virtual disk' API). Hence their scope is likely to be limited to a single local network.

xFS: A group at the University of California, Berkeley, proposed a serverless network file system architecture and developed a prototype implementation called xFS [Anderson *et al.* 1996]. Their approach was motivated by three factors:

1. the opportunity provided by fast switched LANs for multiple file servers in a local network to transfer bulk data to clients concurrently;
2. increased demand for access to shared data;
3. the fundamental limitations of systems based on central file servers.

Concerning (3), they refer to the facts that the construction of high-performance NFS servers requires relatively costly hardware with multiple CPUs, disks and network controllers, and that there are limits to the process of partitioning the file space – i.e., placing shared files in separate filesystems mounted on different servers. They also point to the fact that a central server represents a single point of failure.

xFS is 'serverless' in the sense that it distributes file server processing responsibilities across a set of available computers in a local network at the granularity of individual files. Storage responsibilities are distributed independently of management and other service responsibilities: xFS implements a software RAID storage system, striping file data across disks on multiple computers (in this regard it is a precursor to the Tiger video file server described in Chapter 20), together with a log-structuring technique similar to the Zebra file system.

Responsibility for the management of each file can be allocated to any of the computers supporting the xFS service. This is achieved through a metadata structure called the *manager map*, which is replicated at all clients and servers. File identifiers include a field that acts as an index into the manager map, and each entry in the map identifies the computer that is currently responsible for managing the corresponding file. Several other metadata structures, similar to those found in other log-structured and RAID storage systems, are used for the management of the log-structured file storage and the striped disk storage.

Anderson *et al.* constructed a preliminary prototype of xFS and evaluated its performance. The prototype was incomplete at the time the evaluation was carried out –

the implementation of crash recovery was unfinished and the log-structured storage scheme lacked a cleaner component to recover space occupied by stale logs and compact files.

The performance evaluations carried out with this preliminary prototype used 32 single-processor and dual-processor Sun SPARCstations connected to a high-speed network. The evaluations compared the xFS file service running on up to 32 workstations with NFS and with AFS, each running on a single dual-processor Sun SPARCStation. The read and write bandwidths achieved with xFS with 32 servers exceeded those of NFS and AFS with a single dual-processor server by approximately a factor of 10. The difference in performance was much less marked when xFS was compared with NFS and AFS using the standard AFS benchmark. But overall, the results indicate that the highly distributed processing and storage architecture of xFS offers a promising direction for achieving better scalability in distributed file systems.

Frangipani: Frangipani is a highly scalable distributed file system developed and deployed at the Digital Systems Research Center (now Compaq Systems Research Center) [Thekkath *et al.* 1997]. Its goals are very similar to those of xFS, and like xFS, it approaches them with a design that separates persistent storage responsibilities from other file service actions. But Frangipani's service is structured as two totally independent layers. The lower layer is provided by the Petal distributed virtual disk system [Lee and Thekkath 1996].

Petal provides a distributed virtual disk abstraction across many disks located on multiple servers on a switched local network. The virtual disk abstraction tolerates most hardware and software failures with the aid of replicas of the stored data and autonomously balances the load on servers by relocating data. Petal virtual disks are accessed through a UNIX disk driver using standard block input-output operations, so they can be used to support most file systems. Petal adds between 10 and 100% to the latency of disk accesses, but the caching strategy results in read and write throughputs at least as good as those of the underlying disk drives.

Frangipani server modules run within the operating system kernel. As in xFS, the responsibility for managing files and associated tasks (including the provision of a file-locking service for clients) is assigned to hosts dynamically, and all machines see a unified file name space with coherent access (with approximately single-copy semantics) to shared updatable files. Data is stored in a log-structured and striped format in the Petal virtual disk store. The use of Petal relieves Frangipani of the need to manage physical disk space, resulting in a much simpler distributed file system implementation. Frangipani can emulate the service interfaces of several existing file services, including NFS and DCE/DFS. Frangipani's performance is at least as good as that of the Digital implementation of the UNIX file system.

12.6 Summary

The key design issues for distributed file systems are:

- the effective use of client caching to achieve performance equal to or better than that of local file systems;
- the maintenance of consistency between multiple cached client copies of files when they are updated;
- recovery after client or server failure;
- high throughput for reading and writing files of all sizes;
- scalability.

Distributed file systems are very heavily employed in organizational computing, and their performance has been the subject of much tuning. NFS has a simple stateless protocol, but it has maintained its early position as the dominant distributed file system technology with the help of some relatively minor enhancements to the protocol, tuned implementations and high-performance hardware support.

AFS demonstrated the feasibility of a relatively simple architecture using server state to reduce the cost of maintaining coherent client caches. AFS outperforms NFS in many situations. Recent advances have employed data striping across multiple disks and log-structured writing to further improve performance and scalability.

Current state-of-the-art distributed file systems are highly scalable, provide good performance across both local and wide-area networks, maintain one-copy file update semantics and tolerate and recover from failures. Future requirements include support for mobile users with disconnected operation, and automatic reintegration and quality of service guarantees to meet the need for the persistent storage and delivery of streams of multimedia and other time-dependent data. Solutions to these requirements are discussed in Chapters 18 and 20.

EXERCISES

12.1 Why is there no *open* or *close* operation in our interface to the flat file service or the directory service? What are the differences between our directory service *Lookup* operation and the UNIX *open*? *pages 532–534*

12.2 Outline methods by which a client module could emulate the UNIX file system interface using our model file service. *pages 532–534*

12.3 Write a procedure *PathLookup*(*Pathname, Dir*) → *UFID* that implements *Lookup* for UNIX-like pathnames based on our model directory service. *pages 532–534*

12.4 Why should UFIDs be unique across all possible file systems? How is uniqueness for UFIDs ensured? *page 535*

12.5 To what extent does Sun NFS deviate from one-copy file update semantics? Construct a scenario in which two user-level processes sharing a file would operate correctly in a single UNIX host but would observe inconsistencies when running in different hosts. *page 542*

12.6 Sun NFS aims to support heterogeneous distributed systems by the provision of an operating system–independent file service. What are the key decisions that the implementer of an NFS server for an operating system other than UNIX would have to take? What constraints should an underlying filing system obey to be suitable for the implementation of NFS servers? *page 536*

12.7 What data must the NFS client module hold on behalf of each user-level process? *pages 536–537*

12.8 Outline client module implementations for the UNIX *open()* and *read()* system calls, using the NFS RPC calls of Figure 12.9, (i) without and (ii) with a client cache. *pages 538, 542*

12.9 Explain why the RPC interface to early implementations of NFS is potentially insecure. The security loophole has been closed in NFS 3 by the use of encryption. How is the encryption key kept secret? Is the security of the key adequate? *pages 539, 544*

12.10 After the timeout of an RPC call to access a file on a hard-mounted file system the NFS client module does not return control to the user-level process that originated the call. Why? *page 539*

12.11 How does the NFS automounter help to improve the performance and scalability of NFS? *page 541*

12.12 How many lookup calls are needed to resolve a five-part pathname (for example, */usr/users/jim/code/xyz.c*) for a file that is stored on an NFS server? What is the reason for performing the translation step-by-step? *page 540*

12.13 What condition must be fulfilled by the configuration of the mount tables at the client computers for access transparency to be achieved in an NFS-based filing system? *page 540*

12.14 How does AFS gain control when an *open* or *close* system call referring to a file in the shared file space is issued by a client? *page 549*

12.15 Compare the update semantics of UNIX when accessing local files with those of NFS and AFS. Under what circumstances might clients become aware of the differences? *pages 542, 554*

12.16 How does AFS deal with the risk that callback messages may be lost? *page 552*

12.17 Which features of the AFS design make it more scalable than NFS? What are the limits on its scalability, assuming that servers can be added as required? Which recent developments offer greater scalability? *pages 545, 556, 561*