Distributed File Systems: Concepts and Examples

ELIEZER LEVY and ABRAHAM SILBERSCHATZ

Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712-1188

The purpose of a distributed file system (DFS) is to allow users of physically distributed computers to share data and storage resources by using a common file system. A typical configuration for a DFS is a collection of workstations and mainframes connected by a local area network (LAN). A DFS is implemented as part of the operating system of each of the connected computers. This paper establishes a viewpoint that emphasizes the dispersed structure and decentralization of both data and control in the design of such systems. It defines the concepts of transparency, fault tolerance, and scalability and discusses them in the context of DFSs. The paper claims that the principle of distributed operation is fundamental for a fault tolerant and scalable DFS design. It also presents alternatives for the semantics of sharing and methods for providing access to remote files. A survey of contemporary UNIX®-based systems, namely, UNIX United, Locus, Sprite, Sun's Network File System, and ITC's Andrew, illustrates the concepts and demonstrates various implementations and design alternatives. Based on the assessment of these systems, the paper makes the point that a departure from the approach of extending centralized file systems over a communication network is necessary to accomplish sound distributed file system design.

Categories and Subject Descriptors: C.2.4 **[Computer-Communication Networks]**: Distributed Systems—*distributed applications*; *network operating systems*; D.4.2 **[Operating Systems]**: Storage Management—*allocation/deallocation strategies*; *storage hierarchies*; D.4.3 **[Operating Systems]**: File Systems Management—*directory structures*; *distributed file systems*; *file organization*; *maintenance*; D.4.4 **[Operating Systems]**: Communication Management—*buffering*; *network communication*; D.4.5 **[Operating Systems]**: Reliability—*fault tolerance*; D.4.7 **[Operating Systems]**: Organization and Design—*distributed systems*; F.5 **[Files]**: Organization/structure

General Terms: Design, Reliability

Additional Key Words and Phrases: Caching, client-server communication, network transparency, scalability, UNIX

INTRODUCTION

The need to share resources in a computer system arises due to economics or the nature of some applications. In such cases, it is necessary to facilitate sharing long-term storage devices and their data. This paper discusses Distributed File Systems (DFSs) as the means of sharing storage space and data.

A *file system* is a subsystem of an operating system whose purpose is to provide *long-term storage*. It does so by implementing *files*—named objects that exist from their explicit creation until their explicit destruction and are immune to temporary

® UNIX is a trademark of AT&T Bell Laboratories.

CONTENTS

failures in the system. A DFS is a distributed implementation of the classical time-sharing model of a file system, where multiple users share files and storage resources. The UNIX time-sharing file system is usually regarded as the model [Ritchie and Thompson 1974]. The purpose of a DFS is to support the same kind of sharing when users are physically dispersed in a distributed system. A *distributed system* is a collection of loosely coupled machines—either a mainframe or a workstation—interconnected by a communication network. Unless specified otherwise, the *network* is a local area network (LAN). From the point of view of a specific machine in a distributed system, the rest of the machines and their respective resources are *remote* and the machine's own resources are *local.*

To explain the structure of a DFS, we need to define service, server, and client [Mitchell 1982]. A *service* is a software entity running on one or more machines and providing a particular type of function to a priori unknown clients. A *server* is the service software running on a single machine. A *client* is a process that can invoke a service using a set of operations that form its *client interface* (see below). Sometimes, a lower level interface is defined for the actual cross-machine interaction. When the need arises, we refer to this interface as the *intermachine interface.* Clients implement interfaces suitable for higher level applications or direct access by humans.

Using the above terminology, we say a file system provides file services to clients. A client interface for a file service is formed by a set of *file operations.* The most primitive operations are Create a file, Delete a file, Read from a file, and Write to a file. The primary hardware component a file server controls is a set of secondary storage devices (i.e., magnetic disks) on which files are stored and from which they are retrieved according to the client's requests. We often say that a server, or a machine, stores a file, meaning the file resides on one of its attached devices. We refer to the file system offered by a uniprocessor, time-sharing operating system (e.g., UNIX 4.2 BSD) as a *conventional file system.*

A DFS is a file system, whose clients, servers, and storage devices are dispersed among the machines of a distributed system. Accordingly, service activity has to be carried out across the network, and instead of a single centralized data repository there are multiple and independent storage devices. As will become evident, the concrete configuration and implementation of a DFS may vary. There are configurations where servers run on dedicated machines, as well as configurations where a machine can be both a server and a client. A DFS can be implemented as part of a distributed operating system or, alternatively, by a software layer whose task is to manage the communication between conventional operating systems and file systems. The distinctive features of a DFS are the multiplicity and autonomy of clients and servers in the system.

The paper is divided into two parts. In the first part, which includes Sections 1 to 6, the basic concepts underlying the design of a DFS are discussed. In particular, alternatives and trade-offs regarding the design of a DFS are pointed out. The second part surveys five DFSs: UNIX United [Brownbridge et al. 1982; Randell 1983], Locus [Popek and Walker 1985; Walker et al. 1983], Sun's Network File System (NFS) [Sandberg et al. 1985; Sun Microsystems Inc. 1988], Sprite [Nelson et al., 1988; Ousterhout et al. 1988], and Andrew [Howard et al. 1988; Morris et al. 1986; Satyanarayanan et al. 1985]. These systems exemplify the concepts and observations mentioned in the first part and demonstrate various implementations. A point in the first part is often illustrated by referring to a later section covering one of the surveyed systems.

The fundamental concepts of a DFS can be studied without paying significant attention to the actual operating system of which it is a component. The first part of the paper adopts this approach. The second part reviews actual DFS architectures that serve to demonstrate approaches to integration of a DFS with an operating system and a communication network. To complement our discussion, we refer the reader to the survey paper by Tanenbaum and Van Renesse [1985], where the broader context of distributed operating systems and communication primitives are discussed.

In light of the profusion of UNIX-based DFSs and the dominance of the UNIX file system model, five UNIX-based systems are surveyed. The first part of the paper is independent of this choice as much as possible. Since a vast majority of the actual DFSs (and all systems surveyed and mentioned in this paper) have some relation to UNIX, however, it is inevitable that the concepts are understood best in the UNIX context. The choice of the five systems and the order of their presentation demonstrate the evolution of DFSs in the last decade.

Section 1 presents the terminology and concepts of transparency, fault tolerance, and scalability. Section 2 discusses transparency and how it is expressed in naming schemes in greater detail. Section 3 introduces notions that are important for the semantics of sharing files, and Section 4 compares methods of caching and remote service. Sections 5 and 6 discuss issues related to fault tolerance and scalability, respectively, pointing out observations based on the designs of the surveyed systems. Sections 7–11 describe each of the five systems mentioned above, including distinctive features of a system not related to the issues presented in the first part. Each description is followed by a summary of the prominent features of the corresponding system. A table compares the five systems and concludes the survey. Many important aspects of DFSs and systems are omitted from this paper; thus, Section 12 reviews related work not emphasized in our discussion. Finally, Section 13 provides conclusions and a bibliography provides related literature not directly referenced.

1. TRENDS AND TERMINOLOGY

Ideally, a DFS should look to its clients like a conventional, centralized file system. That is, the multiplicity and dispersion of servers and storage devices should be transparent to clients. As will become evident, transparency has many dimensions and degrees. A fundamental property, called *network transparency*, implies that clients should be able to access remote files using the same set of file operations applicable to local files. That is, the client interface of a DFS should not distinguish between local and remote files. It is up to the DFS to locate the files and arrange for the transport of the data.

Another aspect of transparency is *user mobility*, which implies that users can log in to any machine in the system; that is, they are not forced to use a specific machine. A transparent DFS facilitates user mobility by bringing the user's environment (e.g., home directory) to wherever he or she logs in.

The most important *performance* measurement of a DFS is the amount of time needed to satisfy service requests. In conventional systems, this time consists of disk access time and a small amount of CPU processing time. In a DFS, a remote access has the additional overhead attributed to the distributed structure. This overhead includes the time needed to deliver the request to a server, as well as the time needed to get the response across the network back to the client. For each direction, in addition to the actual transfer of the information, there is the CPU overhead of running the communication protocol software. The performance of a DFS can be viewed as another dimension of its transparency; that is, the performance of a DFS should be comparable to that of a conventional file system.

We use the term *fault tolerance* in a broad sense. Communication faults, machine failures (of type fail stop), storage device crashes, and decays of storage media are all considered to be faults that should be tolerated to some extent. A fault-tolerant system should continue functioning, perhaps in a degraded form, in the face of these failures. The degradation can be in performance, functionality, or both but should be proportional, in some sense, to the failures causing it. A system that grinds to a halt when a small number of its components fail is not fault tolerant.

The capability of a system to adapt to increased service load is called *scalability*. Systems have bounded resources and can become completely saturated under increased load. Regarding a file system, saturation occurs, for example, when a server's CPU runs at very high utilization rate or when disks are almost full. As for a DFS in particular, server saturation is even a bigger threat because of the communication overhead associated with processing remote requests. Scalability is a relative property; a scalable system should react more gracefully to increased load than a nonscalable one will. First, its performance should degrade more moderately than that of a nonscalable system. Second, its resources should reach a saturated state later, when compared with a nonscalable system.

Even a perfect design cannot accommodate an ever-growing load. Adding new resources might solve the problem, but it might generate additional indirect load on other resources (e.g., adding machines to a distributed system can clog the network and increase service loads). Even worse, expanding the system can incur expensive design modifications. A scalable system should have the potential to grow without the above problems. In a distributed system, the ability to scale up gracefully is of special importance, since expanding the network by adding new machines or interconnecting two networks together is commonplace. In short, a scalable design should withstand high-service load, accommodate growth of the user community, and enable simple integration of added resources.

Fault tolerance and scalability are mutually related to each other. A heavily loaded component can become paralyzed and behave like a faulty component. Also, shifting a load from a faulty component to its backup can saturate the latter. Generally, having spare resources is essential for reliability, as well as for handling peak loads gracefully.

An advantage of distributed systems over centralized systems is the potential for fault tolerance and scalability because of the multiplicity of resources. Inappropriate design can, however, obscure this potential and, worse, hinder the system's scalability and make it failure prone. Fault tolerance and scalability considerations call for a design demonstrating distribution of control and data. Any centralized entity, be it a central controller or a central data repository, introduces both a severe point of failure and a performance bottleneck. Therefore, a scalable and fault-tolerant DFS should have multiple and independent servers controlling multiple and independent storage devices.

The fact that a DFS manages a set of dispersed storage devices is its key distinguishing feature. The overall storage space managed by a DFS consists of different and remotely located smaller storage spaces. Usually there is correspondence between these constituent storage spaces and sets of files. We use the term *component unit* to denote the smallest set of files that can be stored on a single machine, independently from other units. All files belonging to the same component unit must reside in the same location. We illustrate the definition of a component unit by drawing an analogy from (conventional) UNIX, where multiple disk partitions play the role of distributed storage sites. There, an entire removable file system is a component unit, since a file system must fit within a single disk partition [Ritchie and Thompson 1974]. In all five systems, a component unit is a partial subtree of the UNIX hierarchy.

Before we proceed, we stress that the distributed nature of a DFS is fundamental to our view. This characteristic lays the foundation for a scalable and fault-tolerant system. Yet, for a distributed system to be conveniently used, its underlying dispersed structure and activity should be made transparent to users. We confine ourselves to discussing DFS designs in the context of transparency, fault tolerance, and scalability. The aim of this paper is to develop an understanding of these three concepts on the basis of the experience gained with contemporary systems.

2. NAMING AND TRANSPARENCY

Naming is a mapping between logical and physical objects. Users deal with logical data objects represented by file names, whereas the system manipulates physical blocks of data stored on disk tracks. Usually, a user refers to a file by a textual name. The latter is mapped to a lower-level numerical identifier, which in turn is mapped to disk blocks. This multilevel mapping provides users with an abstraction of a file that hides the details of how and where the file is actually stored on the disk.

In a transparent DFS, a new dimension is added to the abstraction, that of hiding where in the network the file is located. In a conventional file system the range of the name mapping is an address within a disk; in a DFS it is augmented to include the specific machine on whose disk the file is stored. Going further with the concept of treating files as abstractions leads to the notion of *file replication*. Given a file name, the mapping returns a set of the locations of this file's replicas [Ellis and Floyd 1983]. In this abstraction, both the existence of multiple copies and their locations are hidden.

In this section, we elaborate on transparency issues regarding naming in a DFS. After introducing the properties in this context, we sketch approaches to naming and discuss implementation techniques.

2.1 Location Transparency and Independence

This section discusses transparency in the context of file names. First, two related notions regarding name mappings in a DFS need to be differentiated:

- *Location Transparency.* The name of a file does not reveal any hint as to its physical storage location.
- *Location Independence.* The name of a file need not be changed when the file's physical storage location changes.

Both definitions are relative to the discussed level of naming, since files have different names at different levels (i.e., user-level textual names, and system-level numerical identifiers). A location-independent naming scheme is a dynamic mapping, since it can map the same file name to different locations at two different instances of time. Therefore, location independence is a stronger property than location transparency. Location independence is often referred to as *file migration* or *file mobility*. When referring to file migration or mobility, one implicitly assumes that the movement of files is totally transparent to users. That is, files are migrated by the system without the users being aware of it.

In practice, most of the current file systems (e.g., Locus, NFS, Sprite) provide a static, location-transparent mapping for user-level names. The notion of location independence is, however, irrelevant for these systems. Only Andrew and some experimental file systems support location independence and file mobility (e.g., Eden [Almes et al., 1983; Jessop et al. 1982]). Andrew supports file mobility mainly for administrative purposes. A protocol provides migration of Andrew's component units upon explicit request without changing the user-level or the low-level names of the corresponding files (see Section 11.2 for details).

There are few other aspects that can further differentiate and contrast location independence and location transparency:

- Divorcing data from location, as exhibited by location independence, provides a better abstraction for files. Location-independent files can be viewed as logical data containers not attached to a specific storage location. If only location transparency is supported, however, the file name still denotes a specific, though hidden, set of physical disk blocks.
- Location transparency provides users with a convenient way to share data. Users may share remote files by naming them in a location-transparent manner as if they were local. Nevertheless, sharing the storage space is cumbersome, since logical names are still statically attached to physical storage devices. Location independence promotes sharing the storage space itself, as well as sharing the data objects. When files can be mobilized, the overall, systemwide storage space looks like a single, virtual resource. A possible benefit of such a view is the ability to balance the utilization of disks across the system. Load balancing of the servers themselves is also made possible by this approach, since files can be migrated from heavily loaded servers to lightly loaded ones.
- Location independence separates the naming hierarchy from the storage devices hierarchy and the interserver structure. By contrast, if only location transparency is used (although names are transparent), one can easily expose the correspondence between component units and machines. The machines are configured in a pattern similar to the naming structure. This may restrict the architecture of the system unnecessarily and conflict with other considerations. A server in charge of a root directory is an example for a structure dictated by the naming hierarchy and contradicts decentralization guidelines. An excellent example of separation of the service structure from the naming hierarchy can be found in the design of the Grapevine system [Birrel et al. 1982; Schroeder et al. 1984].

The concept of file mobility deserves more attention and research. We envision future DFS that supports location independence completely and exploits the flexibility that this property entails.

2.2 Naming Schemes

There are three main approaches to naming schemes in a DFS [Barak et al. 1986]. In the simplest approach, files are named by some combination of their host name and local name, which guarantees a unique system-wide name. In Ibis for instance, a file is uniquely identified by the name *host:local-name*, where local name is a UNIX-like path [Tichy and Ruan 1984]. This naming scheme is neither location transparent nor location independent. Nevertheless, the same file operations can be used for both local and remote files; that is, at least the fundamental network transparency is provided. The structure of the DFS is a collection of isolated component units that are entire conventional file systems. In this first approach, component units remain isolated, although means are provided to refer to a remote file. We do not consider this scheme any further in this paper.

The second approach, popularized by Sun's NFS, provides means for individual machines to attach (or *mount* in UNIX jargon) remote directories to their local name spaces. Once a remote directory is attached locally, its files can be named in a location-transparent manner. The resulting name structure is versatile; usually it is a forest of UNIX trees, one for each machine, with some overlapping (i.e., shared) subtrees. A prominent property of this scheme is the fact that the shared name space may not be identical at all the machines. Usually this is perceived as a serious disadvantage; however, the scheme has the potential for creating customized name spaces for individual machines.

Total integration between the component file systems is achieved using the third approach—a single global name structure that spans all the files in the system. Consequently, the same name space is visible to all clients. Ideally, the composed file system structure should be isomorphic to the structure of a conventional file system. In practice, however, there are many special files that make the ideal goal difficult to attain. (In UNIX, for example, I/O devices are treated as ordinary files and are represented in the directory **/dev**; object code of system programs reside in the directory **/bin**. These are special files specific to a particular hardware setting.) Different variations of this approach are examined in the sections on UNIX United, Locus, Sprite, and Andrew.

All important criterion for evaluating the above naming structures is administrative complexity. The most complex structure and most difficult to maintain is the NFS structure. The effects of a failed machine, or taking a machine off-line, are that some arbitrary set of directories on different machines becomes unavailable. Likewise, migrating files from one machine to another requires changes in the name spaces of all the affected machines. In addition, a separate accreditation mechanism had to be devised for controlling which machine is allowed to attach which directory to its name space.

2.3 Implementation Techniques

This section reviews commonly used techniques related to naming.

2.3.1 Pathname Translation

The mapping of textual names to low-level identifiers is typically done by a recursive *lookup* procedure based on the one used in conventional UNIX [Ritchie and Thompson 1974]. We briefly review how this procedure works in a DFS scenario by illustrating the lookup of the textual name **/a/b/c** of Figure 1. The figure shows a partial name structure constructed from three component units using the third scheme mentioned above. For simplicity, we assume that the location table is available to all the machines. Suppose that the lookup is initiated by a client on machine1. First, the root directory '/' (whose low-level identifier and hence its location on disk is known in advance) is searched to find the entry with the low-level identifier of **a**. Once the low-level identifier of **a** is found, the directory **a** itself can be fetched from disk. Now, **b** is looked for in this directory. Since **b** is remote, an indication that **b** belongs to *cu*2 is recorded in the entry of b in the directory **a**. The component of the name looked up so far is stripped off and the remainder (**/b/c**) is passed on to machine2. On machine2, the lookup is continued and eventually machine3 is contacted and the low-level identifier of **/a/b/c** is returned to the client. All five systems mentioned in this paper use a variant of this lookup procedure. Joining component units together and recording the points where they are joined (e.g., **b** is such a point in the above example) is done by the mount mechanism discussed below.

There are few options to consider when machine boundaries are crossed in the course of a pathname traversal. We refer again to the above example. Once machine2 is contacted, it can look up **b** and respond immediately to machine1. Alternatively, machine2 can initiate the contact with machine3 on behalf of the client on machine1. This choice has ramifications on fault tolerance that are discussed in Section 5.2. Among the surveyed systems, only in UNIX United are lookups forwarded from machine to machine on behalf of the lookup initiator. If machine2 responds immediately, it can either respond with the low-level identifier of **b** or send as a reply the

component unit	*server*
cu1	machine1
cu2	machine2
cu3	machine3

Location Table

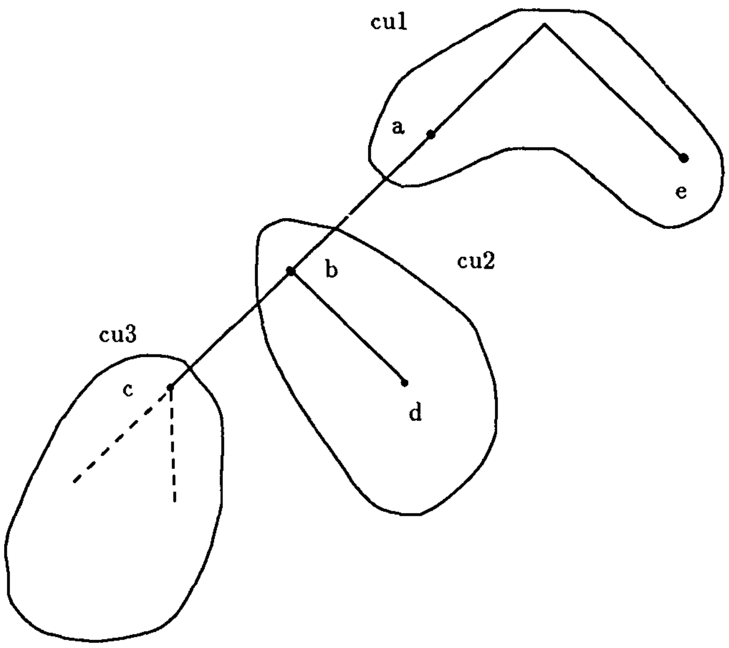

Figure 1. Lookup example.

entire parent directory of **b**. In the former it is the server (machine2 in the example) that performs the lookup, whereas in the latter it is the client that initiates the lookup that actually searches the directory. In case the server's CPU is loaded, this choice is of consequence. In Andrew and Locus, clients perform the lookups; in NFS and Sprite the servers perform it.

2.3.2 Structured Identifiers

Implementing transparent naming requires the provision of the mapping of a file name to its location. Keeping this mapping manageable calls for aggregating sets of files into component units and providing the mapping on a component unit basis rather than on a single file basis. Typically, *structured identifiers* are used for this aggregation. These are bit strings that usually have two parts. The first part identifies the component unit to which file belongs; the second identifies the particular file within the unit. Variants with more parts are possible. The invariant of structured names is, however, that individual parts of the name are unique for all times only within the context of the rest of the parts. Uniqueness at all times can be obtained by not reusing a name that is still used, or by allocating a sufficient number of bits for the names (this method is used in Andrew), or by using a time stamp as one of the parts of the name (as done in Apollo Domain [Leach et al. 1982]).

To enhance the availability of the crucial name to location mapping information, methods such as replicating it or caching parts of it locally by clients are used. As was noted, location independence means that the mapping changes in time and, hence, replicating the mapping makes updating the information consistently a complicated matter. Structured identifiers are location independent; they do not mention servers' locations at all. Hence, these identifiers can be replicated and cached freely without being invalidated by migration of component units. A smaller, second level of mapping that maps component units to locations is the only information that does change when files migrate. The usage of the techniques of aggregation of files into component units and lower-level, location-independent file identifiers is exemplified in Andrew (Section 11) and Locus (Section 8).

We illustrate the above techniques with the example in Figure 1. Suppose the pathname **/a/b/c** is translated to the structured, low-level identifier <*cu*3, 11>, where *cu*3 denotes that file's component unit and 11 identifies it in that unit. The only place where machine locations are recorded is in the location table. Hence, the correspondence between **/a/b/c** and <*cu*3, 11> is not invalidated once *cu*3 is migrated to machine2; only the location table should be updated.

2.3.3 Hints

A technique often used for location mapping in a DFS is that of *hints* [Lampson 1983; Terry 1987]. A hint is a piece of information that speeds up performance if it is correct and does not cause any semantically negative effects if it is incorrect. In essence, a hint improves performance similarly to cached information. A hint may be wrong, however; therefore, its correctness must be validated upon use. To illustrate how location information is treated as hints, assume there is a location server that always reflects the correct and complete mapping of files to locations. Also assume that clients cache parts of this mapping locally. The cached location information is treated as a hint. If a file is found using the hint, a substantial performance gain is obtained. On the other hand, if the hint was invalidated because the file had been migrated, the client's lookup would fail. Consequently, the client must resort to the more expensive procedure of querying the location server; but, still, no semantically negative effects are caused. Examples of using hints abound: Clients in Andrew cache location information from servers and treat this information as hints (see Section 11.4). Sprite uses an effective form of hints called prefix tables and resorts to broadcasting when the hint is wrong (see Section 10.2). The location mechanism of Apollo Domain is based on hints and heuristics [Leach et al. 1982]. The Grapevine mail system counts on hints to locate mailboxes of mail recipients [Birrel et al. 1982].

2.3.4 Mount Mechanism

Joining remote file systems to create a global name structure is often done by the *mount mechanism.* In conventional UNIX, the mount mechanism is used to join together several self-contained file systems to form a single hierarchical name space [Quarterman et al. 1985; Ritchie and Thompson 1974]. A mount operation binds the root of one file system to a directory of another file system. The former file system hides the subtree descending from the mounted-over directory and looks like an integral subtree of the latter file system. The directory that glues together the two file systems is called a *mount point.* All mount operations are recorded by the operating system kernel in a *mount table.* This table is used to redirect name lookups to the appropriate file systems. The same semantics and mechanisms are used to mount a remote file system over a local one. Once the mount is complete, files in the remote file system can be accessed locally as if they were ordinary descendants of the mount point directory. The mount mechanism is used with slight variations in Locus, NFS, Sprite, and Andrew. Section 9.2.1 presents a detailed example of the mount operation.

3. SEMANTICS OF SHARING

The semantics of sharing are important criteria for evaluating any file system that allows multiple clients to share files. It is a characterization of the system that specifies the effects of multiple clients accessing a shared file simultaneously. In particular, these semantics should specify when modifications of data by a client are observable, if at all, by remote clients.

For the following discussion we need to assume that a series of file accesses (i.e., Reads and Writes) attempted by a client to the same file are always enclosed between the Open and Close operations. We denote such a series of accesses as a *file session.*

It should be realized that applications that use the file system to store data and pose constraints on concurrent accesses in order to guarantee the semantic consistency of their data (i.e., database applications) should use special means (e.g., locks) for this purpose and not rely on the underlying semantics of sharing provided by the file system.

To illustrate the concept, we sketch several examples of semantics of sharing mentioned in this paper. We outline the gist of the semantics and not the whole detail.

3.1 UNIX Semantics

- Every Read of a file sees the effects of *all* previous Writes performed on that file in the DFS. In particular, Writes to an open file by a client are visible immediately by other (possibly remote) clients who have this file open at the same time.
- It is possible for clients to share the pointer of current location into the file. Thus, the advancing of the pointer by one client affects all sharing clients.

Consider a sequence interleaving all the accesses to the same file regardless of the identity of the issuing client. Enforcing the above semantics guarantees that each successive access sees the effects of the ones that precede it in that sequence. In a file system context, such an interleaving can be totally arbitrary, since, in contrast to database management systems, sequences of accesses are not defined as transactions. These semantics lend themselves to an implementation where a file is associated with a single physical image that serves all accesses in some serial order (which is the order captured in the above sequence). Contention for this single image results in clients being delayed. The sharing of the location pointer mentioned above is an artifact of UNIX and is needed primarily for compatibility of distributed UNIX systems with conventional UNIX software. Most DFSs try to emulate these semantics to some extent (e.g., Locus, Sprite) mainly because of compatibility reasons.

3.2 Session Semantics

- Writes to an open file are visible immediately to local clients but are invisible to remote clients who have the same file open simultaneously.
- Once a file is closed, the changes made to it are visible only in later starting sessions. Already open instances of the file do not reflect these changes.

According to these semantics, a file may be temporarily associated with several (possibly different) images at the same time. Consequently, multiple clients are allowed to perform both Read and Write accesses concurrently on their image of the file, without being delayed. Observe that when a file is closed, all remote active sessions are actually using a stale copy of the file. Here, it is evident that application programs that care about the serialization of accesses (e.g., a distributed database application) should coordinate their accesses explicitly and not rely on these semantics.

3.3 Immutable Shared Files Semantics

A different, quite unique approach is that of immutable shared files [Schroeder et al. 1985]. Once a file is declared as shared by its creator, it cannot be modified any more. An immutable file has two important properties: Its name may not be reused, and its contents may not be altered. Thus, the name of an immutable file signifies the fixed contents of the file, not the file as a container for variable information. The implementation of these semantics in a distributed system is simple since the sharing is in read-only mode.

3.4 Transaction-Like Semantics

Identifying a file session with a transaction yields the following, familiar semantics: The effects of file sessions on a file and their output are equivalent to the effect and output of executing the same sessions in some serial order. Locking a file for the duration of a session implements these semantics. Refer to the rich literature on database management systems to understand the concepts of transactions and locking [Bernstein et al. 1987]. In the Cambridge File Server, the beginning and end of a transaction are implicit in the Open file, Close file operations, and transactions can involve only one file [Needham and Herbert 1982]. Thus, a file session in that system is actually a transaction.

Variants of UNIX and (to a lesser degree) session semantics are the most commonly used policies. An important trade-off emerges when evaluating these two extremes of sharing semantics. Simplicity of a distributed implementation is traded for the strength of the semantics' guarantee. UNIX semantics guarantee the strong effect of making all accesses see the same version of the file, thereby ensuring that every access is affected by all previous ones. On the other hand, session semantics do not guarantee much when a file is accessed concurrently, since accesses at different machines may observe different versions of the accessed file. The ramifications on the ease of implementation are discussed in the next section.

4. REMOTE-ACCESS METHODS

Consider a client process that requests to access (i.e., Read or Write) a remote file. Assuming the server storing the file was located by the naming scheme, the actual data transfer to satisfy the client's request for the remote access should take place. There are two complementary methods for handling this type of data transfer.

- *Remote Service.* Requests for accesses are delivered to the server. The server machine performs the accesses, and their results are forwarded back to the client. There is a direct correspondence between accesses and traffic to and from the server. Access requests are translated to messages for the servers, and server replies are packed as messages sent back to the clients. Every access is handled by the server and results in network traffic. For example, a Read corresponds to a request message sent to the server and a reply to the client with the requested data. A similar notion called Remote Open is defined in Howard et al. [1988].
- *Caching.* If the data needed to satisfy the access request are not present locally, a copy of those data is brought from the server to the client. Usually the amount of data brought over is much larger than the data actually requested (e.g., whole files or pages versus a few blocks). Accesses are performed on the cached copy in the client side. The idea is to retain recently accessed disk blocks in cache so repeated accesses to the same information can be handled locally, without additional network traffic. Caching performs best when the stream of file accesses exhibits locality of reference. A replacement policy (e.g., Least Recently Used) is used to keep the cache size bounded. There is no direct correspondence between accesses and traffic to the server. Files are still identified, with one master copy residing at the server machine, but copies of (parts of) the file are scattered in different caches. When a cached copy is modified, the changes need to be reflected on the master copy and, depending on the relevant sharing semantics, on any other cached copies. Therefore, Write accesses may incur substantial overhead. The problem of keeping the cached copies consistent with the master file is referred to as the *cache consistency problem* [Smith 1982].

It should be realized that there is a direct analogy between disk access methods in conventional file systems and remote access methods in DFSs. A pure remote service method is analogous to performing a disk access for each and every access request. Similarly, a caching scheme in a DFS is an extension of caching or buffering techniques in conventional file systems (e.g., buffering block I/O in UNIX [McKusick et al. 1984]). In conventional file systems, the rationale behind caching is to reduce disk I/O, whereas in DFSs the goal is to reduce network traffic. For these reasons, a pure remote service method is not practical. Implementations must incorporate some form of caching for performance enhancement. Many implementations can be thought of as a hybrid of caching and remote service. In Locus and NFS, for instance, the implementation is based on remote service but is augmented with caching for performance (see Sections 8.3, 8.4, and 9.3.3). On the other hand, Sprite's implementation is based on caching, but under certain circumstances a remote service method is adopted (see Section 10.3). Thus, when we evaluate the two methods we actually evaluate to what degree one method should be emphasized over the other.

An interesting study of the performance aspects of the remote access problem can be found in Cheriton and Zwaenepoel [1983]. This paper evaluates to what extent remote access (using the simplest remote service paradigm) is more expensive than local access.

The remote service method is straightforward and does not require further explanation. Thus, the following material is primarily concerned with the method of caching.

4.1 Designing a Caching Scheme

The following discussion pertains to a (file data) caching scheme between a client's cache and a server. The latter is viewed as a uniform entity and its main memory and disk are not differentiated. Thus, we abstract the traditional caching scheme on the server side, between its own cache and disk.

A caching scheme in a DFS should address the following design decisions [Nelson et al. 1988]:

- The granularity of cached data.
- The location of the client's cache (main memory or local disk).
- How to propagate modifications of cached copies.
- How to determine if a client's cached data are consistent.

The choices for these decisions are intertwined and related to the selected sharing semantics.

4.1.1 Cache Unit Size

The granularity of the cached data can vary from parts of a file to an entire file. Usually, more data are cached than needed to satisfy a single access, so many accesses can be served by the cached data. An early version of Andrew caches entire files. Currently, Andrew still performs caching in big chunks (64Kb). The rest of the systems support caching individual blocks driven by clients' demand, where a block is the unit of transfer between disk and main memory buffers (see sample sizes below). Increasing the caching unit increases the likelihood that data for the next access will be found locally (i.e., the hit ratio is increased); on the other hand, the time required for the data transfer and the potential for consistency problems are increased, too. Selecting the unit of caching involves parameters such as the network transfer unit and the Remote Procedure Call (RPC) protocol service unit (in case an RPC protocol is used) [Birrel and Nelson 1984]. The network transfer unit is relatively small (e.g., Ethernet packets are about 1.5Kb), so big units of cached data need to be disassembled for delivery and reassembled upon reception [Welch 1986].

Typically, block-caching schemes use a technique called *read-ahead*. This technique is useful when sequentially reading a large file. Blocks are read from the server disk and buffered on both the server and client sides before they are actually needed in order to speed up the reading.

One advantage of a large caching unit is reduced network overhead. Recall that running communication protocols accounts for a substantial portion of this overhead. Transferring data in bulks amortizes the protocol cost over many transfer units. At the sender side, one context switch (to load the communication software) suffices to format and transmit multiple packets. At the receiver side, there is no need to acknowledge each packet individually.

Block size and the total cache size are important for block-caching schemes. In UNIX-like systems, common block sizes are 4Kb or 8Kb. For large caches (more than 1Mb), large block sizes (more than 8Kb) are beneficial since the advantages of large caching unit size are dominant [Lazowska et al. 1986; Ousterhout et al. 1985]. For smaller caches, large block sizes are less beneficial because they result in fewer blocks in the cache and most of the cache space is wasted due to internal fragmentation.

4.1.2 Cache Location

Regarding the second decision, disk caches have one clear advantage—reliability. Modifications to cached data are lost in a crash if the cache is kept in volatile memory. Moreover, if the cached data are kept on disk, the data are still there during recovery and there is no need to fetch them again. On the other hand, main-memory caches have several advantages. First, main memory caches permit workstations to be diskless. Second, data can be accessed more quickly from a cache in main memory than from one on a disk. Third, the server caches (used to speed up disk I/O) will be in main memory regardless of where client caches are located; by using main-memory caches on clients, too, it is possible to build a single caching mechanism for use by both servers and clients (as it is done in Sprite). It turns out that the two cache locations emphasize different functionality. Main-memory caches emphasize reduced access time; disk caches emphasize increased reliability and autonomy of single machines. Notice that the current technology trend is larger and cheaper memories. With large main-memory caches, and hence high hit ratios, the achieved performance speed up is predicted to outweigh the advantages of disk caches.

4.1.3 Modification Policy

In the sequel, we use the term *dirty block* to denote a block of data that has been modified by a client. In the context of caching, we use the term *to flush* to denote the action of sending dirty blocks to be written on the master copy.

The policy used to flush dirty blocks back to the server's master copy has a critical effect on the system's performance and reliability. (In this section we assume caches are held in main memories.) The simplest policy is to write data through to the server's disk as soon as it is written to any cache. The advantage of the *write-through* method is its reliability: Little information is lost when a client crashes. This policy requires, however, that each Write access waits until the information is sent to the server, which results in poor Write performance. Caching with write-through is equivalent to using remote service for Write accesses and exploiting caching only for Read accesses.

An alternate write policy is to *delay* updates to the master copy. Modifications are written to the cache and then written through to the server later. This policy has two advantages over write-through. First, since writes are to the cache, Write accesses complete more quickly. Second, data may be deleted before they are written back, in which case they need never be written at all. Unfortunately, *delayed-write* schemes introduce reliability problems, since unwritten data will be lost whenever a client crashes.

There are several variations of the delayed-write policy that differ in when to flush dirty blocks to the server. One alternative is to flush a block when it is about to be ejected from the client's cache. This option can result in good performance, but some blocks can reside in the client's cache for a long time before they are written back to the server [Ousterhout et al. 1985]. A compromise between the latter alternative and the write-through policy is to scan the cache periodically, at regular intervals, and flush blocks that have been modified since the last scan. Sprite uses this policy with a 30-second interval.

Yet another variation on delayed-write, called *write-on-close*, is to write data back to the server when the file is closed. In cases of files open for very short periods or rarely modified, this policy does not significantly reduce network traffic. In addition, the write-on-close policy requires the closing process to delay while the file is written through, which reduces the performance advantages of delayed-writes. The performance advantages of this policy over delayed-write with more frequent flushing are apparent for files that are both open for long periods and modified frequently.

As a reference, we present data regarding the utility of caching in UNIX 4.2 BSD. UNIX 4.2 BSD uses a cache of about 400Kb holding different size blocks (the most common size is 4Kb). A delayed-write policy with 30-second intervals is used. A miss ratio (ratio of the number of real disk I/O to logical disk accesses) of 15 percent is reported in McKusick et al. [1984], and of 50 percent in Ousterhout et al. [1985]. The latter paper also provides the following statistics, which were obtained by simulations on UNIX: A 4Mb cache of 4Kb blocks eliminates between 65 and 90 percent of all disk accesses for file data. A write-through policy resulted in the highest miss ratio. Delayed-write policy with flushing when the block is ejected from cache had the lowest miss ratio.

There is a tight relation between the modification policy and semantics sharing. Write-on-close is suitable for session semantics. By contrast, using any delayed-write policy, when situations of files that are updated concurrently occur frequently in conjunction with UNIX semantics, is not reasonable and will result in long delays and complex mechanisms. A write-through policy is more suitable for UNIX semantics under such circumstances.

4.1.4 Cache Validation

A client is faced with the problem of deciding whether or not its locally cached copy of the data is consistent with the master copy. If the client determines that its cached data is out of date, accesses can no longer be served by that cached data. An up-to-date copy of the data must be brought over. There are basically two approaches to verifying the validity of cached data:

- *Client-initiated approach.* The client initiates a validity check in which it contacts the server and checks whether the local data are consistent with the master copy. The frequency of the validity check is the crux of this approach and determines the resulting sharing semantics. It can range from a check before every single access to a check only on first access to a file (on file Open). Every access that is coupled with a validity check is delayed, compared with an access served immediately by the cache. Alternatively, a check can be initiated every fixed interval of time. Usually the validity check involves comparing file header information (e.g., time stamp of the last update maintained as i-node information in UNIX). Depending on its frequency, this kind of validity check can cause severe network traffic, as well as consume precious server CPU time. This phenomenon was the cause for Andrew designers to withdraw from this approach (Howard et al. [1988] provide detailed performance data on this issue).
- *Server-initiated approach.* The server records for each client the (parts of) files the client caches. Maintaining information on clients has significant fault tolerance implications (see Section 5.1). When the server detects a potential for inconsistency, it must now react. A potential for inconsistency occurs when a file is cached in conflicting modes by two different clients (i.e., at least one of the clients specified a Write mode). If session semantics are implemented, whenever a server receives a request to close a file that has been modified, it should react by notifying the clients to discard their cached data and consider it invalid. Clients having this file open at that time, discard their copy when the current session is over. Other clients discard their copy at once. Under session semantics, the server need not be informed about Opens of already cached files. The server is informed about the Close of a writing session, however. On the other hand, if a more restrictive sharing semantics is implemented, like UNIX semantics, the server must be more involved. The server must be notified whenever a file is opened, and the intended mode (Read or

Write) must be indicated. Assuming such notification, the server can act when it detects a file that is opened simultaneously in conflicting modes by disabling caching for that particular file (as done in Sprite). Disabling caching results in switching to a remote service mode of operation.

A problem with the server-initiated approach is that it violates the traditional client-server model, where clients initiate activities by requesting service. Such violation can result in irregular and complex code for both clients and servers.

In summary, the choice is longer accesses and greater server load using the former method versus the fact that the server maintains information on its clients using the latter.

4.2 Cache Consistency

Before delving into the evaluation and comparison of remote service and caching, we relate these remote access methods to the examples of sharing semantics introduced in Section 3.

- Session semantics are a perfect match for caching entire files. Read and Write accesses within a session can be handled by the cached copy, since the file can be associated with different images according to the semantics. The cache consistency problem diminishes to propagating the modifications performed in a session to the master copy at the end of a session. This model is quite attractive since it has simple implementation. Observe that coupling these semantics with caching parts of files may complicate matters, since a session is supposed to read the image of the *entire* file that corresponds to the time it was opened.
- A distributed implementation of UNIX semantics using caching has serious consequences. The implementation must guarantee that at all times only one client is allowed to write to any of the cached copies of the same file. A distributed conflict resolution scheme must be used in order to arbitrate among clients wishing to access the same file in conflicting modes. In addition, once a cached copy is modified, the changes need to be propagated immediately to the rest of the cached copies. Frequent Writes can generate tremendous network traffic and cause long delays before requests are satisfied. This is why implementations (e.g., Sprite) disable caching altogether and resort to remote service once a file is concurrently open in conflicting modes. Observe that such an approach implies some form of a server-initiated validation scheme, where the server makes a note of all Open calls. As was stated, UNIX semantics lend themselves to an implementation where a file is associated with a single physical image. A remote service approach, where all requests are directed and served by a single server, fits nicely with these semantics.
- The immutable shared files semantics were invented for a whole file caching scheme [Schroeder et al. 1985]. With these semantics, the cache consistency problem vanishes totally.
- Transactions-like semantics can be implemented in a straightforward manner using locking, when all the requests for the same file are served by the same server on the same machine as done in remote service.

4.3 Comparison of Caching and Remote Service

Essentially, the choice between caching and remote service is a choice between potential for improved performance and simplicity. We evaluate the trade-off by listing the merits and demerits of the two methods.

- When caching is used, a substantial amount of the remote accesses can be handled efficiently by the local cache. Capitalizing on locality in file access patterns makes caching even more attractive. Ramifications can be performance transparency: Most of the remote accesses will be served as fast as local ones. Consequently, server load and network traffic are reduced, and the potential for scalability is enhanced. By contrast, when using the remote service method, each remote access is handled across the network. The penalty in network traffic, server load, and performance is obvious.

- Total network overhead in transmitting big chunks of data, as done in caching, is lower than when series of short responses to specific requests are transmitted (as in the remote service method).
- Disk access routines on the server may be better optimized if it is known that requests are always for large, contiguous segments of data rather than for random disk blocks. This point and the previous one indicate the merits of transferring data in bulk, as done in Andrew.
- The cache consistency problem is the major drawback to caching. In access patterns that exhibit infrequent writes, caching is superior. When writes are frequent, however, the mechanisms used to overcome the consistency problem incur substantial overhead in terms of performance, network traffic, and server load.
- It is hard to emulate the sharing semantics of a centralized system in a system using caching as its remote access method. The problem is the cache consistency; namely, the fact that accesses are directed to distributed copies, not to a central data object. Observe that the two caching-oriented semantics, session semantics and immutable shared files semantics, are not restrictive and do not enforce serializability. On the other hand, when using remote service, the server serializes all accesses and, hence, is able to implement any centralized sharing semantics.
- To use caching and benefit from its merits, clients must have either local disks or large main memories. Clients without disks can use remote-service methods without any problems.
- Since, for caching, data are transferred en masse between the server and client, and not in response to the specific needs of a file operation, the lower intermachine interface is quite different from the upper client interface. The remote service paradigm, on the other hand, is just an extension of the local file system interface across the network. Thus, the intermachine interface mirrors the local client-file system interface.

5. FAULT TOLERANCE ISSUES

Fault tolerance is an important and broad subject in the context of DFS. In this section we focus on the following fault tolerance issues. In Section 5.1 we examine two service paradigms in the context of faults occurring while servicing a client. In Section 5.2 we define the concept of availability and discuss how to increase the availability of files. In Section 5.3 we review file replication as another means for enhancing availability.

5.1 Stateful Versus Stateless Service

When a server holds on to information on its clients between servicing their requests, we say the server is *stateful.* Conversely, when the server does not maintain any information on a client once it finished servicing its request, we say the server is *stateless.*

The typical scenario of a stateful file service is as follows. A client must perform an Open on a file before accessing it. The server fetches some information about the file from its disk, stores it in its memory, and gives the client some connection identifier that is unique to the client and the open file. (In UNIX terms, the server fetches the i-node and gives the client a file descriptor, which serves as an index to an in-core table of i-nodes.) This identifier is used by the client for subsequent accesses until the session ends. Typically, the identifier serves as an index into in-memory table that records relevant information the server needs to function properly (e.g., timestamp of last modification of the corresponding file and its access rights). A stateful service is characterized by a virtual circuit between the client and the server during a session. The connection identifier embodies this virtual circuit. Either upon closing the file or by a garbage collection mechanism, the server must reclaim the main-memory space used by clients that are no longer active.

The advantage of stateful service is performance. File information is cached in main memory and can be easily accessed using the connection identifier, thereby saving disk accesses. The key point regarding fault tolerance in a stateful service approach is the main-memory information kept by the server on its clients.

A stateless server avoids this state information by making each request self-contained. That is, each request identifies the file and position in the file (for Read and Write accesses) in full. The server need not keep a table of open files in main memory, although this is usually done for efficiency reasons. Moreover, there is no need to establish and terminate a connection by Open and Close operations. They are totally redundant, since each file operation stands on its own and is not considered as part of a session.

The distinction between stateful and stateless service becomes evident when considering the effects of a crash during a service activity. A stateful server loses all its volatile state in a crash. A graceful recovery of such a server involves restoring this state, usually by a recovery protocol based on a dialog with clients. Less graceful recovery implies abortion of the operations that were underway when the crash occurred. A different problem is caused by client failures. The server needs to become aware of such failures in order to reclaim space allocated to record the state of crashed clients. These phenomena are sometimes referred to as orphan detection and elimination.

A stateless server avoids the above problems, since a newly reincarnated server can respond to a self-contained request without difficulty. Therefore, the effects of server failures and recovery are almost not noticeable. From a client's point of view, there is no difference between a slow server and a recovering server. The client keeps retransmitting its request if it gets no response. Regarding client failures, no obsolete state needs to be cleaned up on the server side.

The penalty for using the robust stateless service is longer request messages and slower processing of requests, since there is no in-core information to speed the processing. In addition, stateless service imposes other constraints on the design of the DFS. First, since each request identifies the target file, a uniform, systemwide, low-level naming is advised. Translating remote to local names for each request would imply even slower processing of the requests. Second, since clients retransmit requests for files operations, these operations must be *idempotent*. An idempotent operation has the same effect and returns the same output if executed several times consecutively. Self-contained Read and Write accesses are idempotent, since they use an absolute byte count to indicate the position within a file and do not rely on an incremental offset (as done in UNIX Read and Write system calls). Care must be taken when implementing destructive operations (such as Delete a file) to make them idempotent too.

In some environments a stateful service is a necessity. If a Wide Area Network (WAN) or Internetworks is used, it is possible that messages are not received in the order they were sent. A stateful, virtual-circuit-oriented service would be preferable in such a case, since by the maintained state it is possible to order the messages correctly. Also observe that if the server uses the server-initiated method for cache validation, it cannot provide stateless service since it maintains a record of which files are cached by which clients. On the other hand, it is easier to build a stateless service than a stateful service on top of a datagram communication protocol [Postel 1980].

The way UNIX uses file descriptors and implicit offsets is inherently stateful. Servers must maintain tables to map the file descriptors to i-nodes and store the current offset within a file. This is why NFS, which uses a stateless service, does not use file descriptors and includes an explicit offset in every access (see Section 9.2.2).

5.2 Improving Availability

Svobodova [1984] defines two file properties in the context of fault tolerance: "A file is *recoverable* if is possible to revert it to an earlier, consistent state when an operation on the file fails or is aborted by the client. A file is called *robust* if it is guaranteed to survive crashes of the storage device and decays of the storage medium." A robust file is not necessarily recoverable and vice versa. Different techniques must be used to implement these two distinct concepts. Recoverable files are realized by atomic update techniques. (We do not give account of atomic updates techniques in this paper.) Robust files are implemented by redundancy techniques such as mirrored files and stable storage [Lampson 1981].

It is necessary to consider the additional criterion of *availability*. A file is called available if it can be accessed whenever needed, despite machine and storage device crashes and communication faults. Availability is often confused with robustness, probably because they both can be implemented by redundancy techniques. A robust file is guaranteed to survive failures, but it may not be available until the faulty component has recovered. Availability is a fragile and unstable property. First, it is temporal; availability varies as the system's state changes. Also, it is relative to a client; for one client a file may be available, whereas for another client on a different machine, the same file may be unavailable.

Replicating files enhances their availability (see Section 5.3); however, merely replicating file is not sufficient. There are some principles destined to ensure increased availability of the files described below.

The number of machines involved in a file operation should be minimal, since the probability of failure grows with the number of involved parties. Most systems adhere to the client-server pair for all file operations. (This refers to a LAN environment, where no routing is needed.) Locus makes an exception, since its service model involves a triple: a client, a server, and a Centralized Synchronization site (CSS). The CSS is involved only in Open and Close operations; but if the CSS cannot be reached by a client, the file is not available to that particular client. In general, having more than two machines involved in a file operation can cause bizarre situations in which a file is available to some but not all clients.

Once a file has been located there is no reason to involve machines other than the client and the server machines. Identifying the server that stores the file and establishing the client-server connection is more problematic. A file location mechanism is an important factor in determining the availability of files. Traditionally, locating a file is done by a pathname traversal, which in a DFS may cross machine boundaries several times and hence involve more than two machines (see Section 2.3.1). In principle, most systems (e.g., Locus, NFS, Andrew) approach the problem by requiring that each component (i.e., directory) in the pathname would be looked up directly by the client. Therefore, when machine boundaries are crossed, the server in the client-server pair changes, but the client remains the same. In UNIX United, partially because of routing concerns, this client-server model is not preserved in the pathname traversal. Instead, the pathname traversal request is forwarded from machine to machine along the pathname, without involving the client machine each time.

Observe that if a file is located by pathname traversal, the availability of a file depends on the availability of all the directories in its pathname. A situation can arise whereby a file might be available to reading and writing clients, but it cannot be located by new clients since a directory in its pathname is unavailable. Replicating top-level directories can partially rectify the problem, and is indeed used in Locus to increase the availability of files.

Caching directory information can both speed up the pathname traversal and avoid the problem of unavailable directories in the pathname (i.e., if caching occurs before the directory in the pathname becomes unavailable). Andrew and NFS use this technique. Sprite uses a better mechanism for quick and reliable pathname traversal. In Sprite, machines maintain prefix tables that map prefixes of pathnames to the servers that store the corresponding component units. Once a file in some component unit is open, all subsequent Opens of files within that same unit address the right server directly, without intermediate lookups at other servers. This mechanism is faster and guarantees better availability. (For complete description of the prefix table mechanism refer to Section 10.2.)

5.3 File Replication

Replication of files is a useful redundancy for improving availability. We focus on replication of files on different machines rather than replication on different media on the same machine (such as mirrored disks [Lampson 1981]). Multimachine replication can benefit performance too, since selecting a nearby replica to serve an access request results in shorter service time.

The basic requirement from a replication scheme is that different replicas of the same file reside on failure-independent machines. That is, the availability of one replica is not affected by the availability of the rest of the replicas. This obvious requirement implies that replication management is inherently a location-dependent activity. Provisions for placing a replica on a particular machine must be available.

It is desirable to hide the details of replication from users. It is the task of the naming scheme to map a replicated file name to a particular replica. The existence of replicas should be invisible to higher levels. At some level, however, the replicas must be distinguished from one another by having different lower level names. This can be accomplished by first mapping a file name to an entity that is able to differentiate the replicas (as done in Locus). Another transparency issue is providing replication control at higher levels. Replication control includes determining the degree of replication and placement of replicas. Under certain circumstances, it is desirable to expose these details to users. Locus, for instance, provides users and system administrators with mechanism to control the replication scheme.

The main problem associated with replicas is their update. From a user's point of view, replicas of a file denote the same logical entity; thus, an update to any replica must be reflected on all other replicas. More precisely, the relevant sharing semantics must be preserved when accesses to replicas are viewed as virtual accesses to their logical files. The analogous database term is One-Copy Serializability [Bernstein et al. 1987]. Davidson et al. [1985] survey approaches to replication for database systems, where consistency considerations are of major importance. If consistency is not of primary importance, it can be sacrificed for availability and performance. This is an incarnation of a fundamental trade-off in the area of fault tolerance. The choice is between preserving consistency at all costs, thereby creating a potential for indefinite blocking, or sacrificing consistency under some (we hope rare) circumstance of catastrophic failures for the sake of guaranteed progress. We illustrate this trade-off by considering (in a conceptual manner) the problem of updating a set of replicas of the same file. The atomicity of such an update is a desirable property; that is, a situation in which both updated and not updated replicas serve accesses should be prevented. The only way to guarantee the atomicity of such an update is by using a commit protocol (e.g., Two-phase commit), which can lead to indefinite blocking in the face of machine and network failures [Bernstein et al. 1987]. On the other hand, if only the available replicas are updated, progress is guaranteed; stale replicas, however, are present.

In most cases, the consistency of file data cannot be compromised, and hence the price paid for increased availability by replication is a complicated update propagation protocol. One case in which consistency can be traded for performance, as well as availability, is replication of the location hints discussed in Section 2.3.2. Since hints are validated upon use, their replication does not require maintaining their consistency. When a location hint is correct, it results in quick location of the corresponding file without relying on a location server. Among the surveyed systems, Locus uses replication extensively and sacrifices consistency in a partitioned environment for the sake of availability of files for both Read and Write accesses (see Section 8.5 for details).

Facing the problems associated with maintaining the consistency of replicas, a popular compromise is *read-only replication.* Files known to be frequently read and rarely modified are replicated using this restricted variant of replication. Usually, only one primary replica can be modified, and the propagation of the updates involves either taking the file off line or using some costly procedure that guarantees atomicity of the updates. Files containing the object code of system programs are good candidates for this kind of replication, as are system data files (e.g., location databases and user registries).

As an illustration of the concepts discussed above, we describe the replication scheme in Ibis, which is quite unique [Tichy and Ruan 1984]. Ibis uses a variation of the *primary copy* approach. The domain of the name mapping is a pair: primary replica identifier and local replica identifier, if there is one. (If there is no replica locally, a special value is returned.) Thus, the mapping is relative to a machine. If the local replica is the primary one, the pair contains two identical identifiers. Ibis supports *demand replication,* which is an automatic replication control policy (similar to whole-file caching). Demand replication means that reading a nonlocal replica causes it to be cached locally, thereby generating a new nonprimary replica. Updates are performed only on the primary copy and cause all other replicas to be invalidated by sending appropriate messages. Atomic and serialized invalidation of all nonprimary replicas is not guaranteed. Hence, it is possible that a stale replica is considered valid. Consistency of replicas is sacrificed for a simple update protocol. To satisfy remote Write accesses, the primary copy is migrated to the requesting machine.

6. Scalability Issues

Very large-scale DFSs, to a great extent, are still visionary. Andrew is the closest system to be classified as a very large-scale system with a planned configuration of thousands of workstations. There are no magic guidelines to ensure the scalability of a system. Examples of nonscalable designs, however, are abundant. In Section 6.1 we discuss several designs that pose problems and propose possible solutions, all in the context of scalability. In Section 6.2 we describe an implementation technique, Light Weight Processes, essential for high-performance and scalable designs.

6.1 Guidelines by Negative Examples

Barak and Kornatzky [1987] list several principles for designing very large-scale systems. The first is called Bounded Resources: "The service demand from any component of the system should be bounded by a constant. This constant is independent of the number of nodes in the system." Any server whose load is proportional to the size of the system is destined to become clogged once the system grows beyond a certain size. Adding more resources will not alleviate the problem. The capacity of this server simply limits the growth of the system. This is why the CSS of Locus is not a scalable design. In Locus, every filegroup (the Locus component unit, which is equivalent to a UNIX removable file system) is assigned a CSS, whose responsibility it is to synchronize accesses to files in that filegroup. Every Open request to a file within that filegroup must go through this machine. Beyond a certain system size, CSSs of frequently accessed filegroups are bound to become a point of congestion, since they would need to satisfy a growing number of clients.

The principle of bounded resources can be applied to channels and network traffic, too, and hence prohibits the use of broadcasting. Broadcasting is an activity that involves *every* machine in the network. A mechanism that relies on broadcasting is simply not realistic for large-scale systems.

The third example combines aspects of scalability and fault tolerance. It was already mentioned that if a stateless service is used, a server need not detect a client's crash nor take any precautions because of it. Obviously this is not the case with stateful service, since the server must detect clients' crashes and at least discard the state it maintains for them. It is interesting to contrast the ways MOS and Locus reclaim obsolete state storage on servers

[Barak and Litman 1985; Barak and Paradise 1986].

The approach taken in MOS is garbage collection. It is the client's responsibility to set, and later reset, an expiration date on state information the servers maintain for it. Clients reset this date whenever they access the server or by special, infrequent messages. If this date has expired, a periodic garbage collector reclaims that storage. This way, the server need not detect clients' crashes. By contrast, Locus invokes a clean-up procedure whenever a server machine determines that a particular client machine is unavailable. Among other things, this procedure releases space occupied by the state of clients from the crashed machine. Detecting crashes can be very expensive, since it is based on polling and time-out mechanisms that incur substantial network overhead. The scheme MOS uses requires tolerable and scalable overhead, where every client signals a bounded number of objects (the object it owns), whereas a failure detection mechanism is not scalable since it depends on the size of the system.

Network congestion and latency are major obstacles to large-scale systems. A guideline worth pursuing is to minimize cross-machine interactions by means of caching, hints, and enforcement of relaxed sharing semantics. There is, however, a trade-off between the strictness of the sharing semantics in a DFS and the network and server loads (and hence necessarily the scalability potential). The more stringent the semantics, the harder it is to scale the system up.

Central control schemes and central resources should not be used to build scalable (and fault-tolerant) systems. Examples of centralized entities are central authentication server, central naming server, and central file server. Centralization is a form of functional asymmetry among the machines comprising the system. The ideal alternative is a configuration that is functionally symmetric; that is, all the component machines have an equal role in the operation of the system, and hence each machine has some degree of autonomy. Practically, it is impossible to comply with such a principle. For instance, incorporating diskless machines violates functional symmetry. Autonomy and symmetry are, however, important goals to which to aspire.

An important aspect of decentralization is system administration. Administrative responsibilities should be delegated to encourage autonomy and symmetry, without disturbing the coherence and uniformity of the distributed system. Andrew and Apollo Domain support decentralized system management [Leach et al. 1985].

The practical approximation to symmetric and autonomous configuration is *clustering*, where a system is partitioned into a collection of semiautonomous clusters. A cluster consists of a set of machines and a dedicated cluster server. To make cross-cluster file references relatively infrequent, most of the time, each machine's requests should be satisfied by its own cluster server. Such a requirement depends on the ability to localize file references and the appropriate placement of component units. If the cluster is well balanced, that is, the server in charge suffices to satisfy a majority of the cluster demands, it can be used as a modular building block to scale up the system. Observe that clustering complies with the Bounded Resources Principle. In essence, clustering attempts to associate a server with a fixed set of clients and a set of files they access frequently, not just with an arbitrary set of files. Andrew's use of clusters, coupled with read-only replication of key files, is a good example for a scalable clustering scheme.

UNIX United emphasizes the concept of autonomy. There, UNIX systems are joined together in a recursive manner to create a larger global system [Randell 1983]. Each component system is a complex UNIX system that can operate and be administered independently. Again, modular and autonomous components are combined to create a large-scale system. The emphasis on autonomy results in some negative effects, however, since component boundaries are visible to users.

6.2 Lightweight Processes

A major problem in the design of any service is the process structure of the server. Servers are supposed to operate efficiently in peak periods when hundreds of active clients need to be served simultaneously. A single server process is certainly not a good choice, since whenever a request necessitates disk I/O the whole service is delayed until the I/O is completed. Assigning a process for each client is a better choice; however, the overhead of multiplexing the CPU among the processes (i.e., the context switches) is an expensive price that must be paid.

A related problem has to do with the fact that all the server processes need to share information, such as file headers and service tables. In UNIX 4.2 BSD processes are not permitted to share address spaces, hence sharing must be done externally by using files and other unnatural mechanisms.

It appears that one of the best solutions for the server architecture is the use of *Lightweight Processes* (LWPs) or *Threads*. A thread is a process that has very little nonshared state. A group of peer threads share code, address space, and operating system resources. An individual thread has at least its own register state. The extensive sharing makes context switches among peer threads and threads' creation inexpensive, compared with context switches among traditional, heavy-weight processes. Thus, blocking a thread and switching to another thread is a reasonable solution to the problem of a server handling many requests. The abstraction presented by a group of LWPs is that of multiple threads of control associated with some shared resources.

There are many alternatives regarding threads; we mention a few of them briefly. Threads can be supported above the kernel, at the user level (as done in Andrew) or by the kernel (as in Mach [Tevanian et al. 1987]). Usually, a lightweight process is not bound to a particular client. Instead, it serves single requests of different clients. Scheduling threads can be preemptive or nonpreemptive. If threads are allowed to run to completion, their shared data need not be explicitly protected. Otherwise, some explicit locking mechanism must be used to synchronize the accesses to the shared data.

Typically, when LWPs are used to implement a service, client requests accumulate in a common queue and threads are assigned to requests from the queue. The advantages of using an LWPs scheme to implement the service are twofold. First, an I/O request delays a single thread, not the entire service. Second, sharing common data structures (e.g., the requests queue) among the threads is easily facilitated.

It is clear that some form of LWPs scheme is essential for servers to be scalable. Locus, Sprite, Andrew, use such schemes; in the future NFS will too. Detailed studies of threads implementations can be found in Kepecs 1985 and Tevanian et al. 1987.

7. UNIX UNITED

The UNIX United project from the University of Newcastle upon Tyne, England, is one of the earliest attempts to extend the UNIX file system to a distributed one without modifying the UNIX kernel. In UNIX United, a software subsystem is added to each of a set of interconnected UNIX systems (referred to as component or constituent systems), so as to construct a distributed system that is functionally indistinguishable from a conventional centralized UNIX system.

Originally, the component systems were perceived as mainframes functioning as time-sharing UNIX systems, and indeed the original implementation was based on a set of PDP-11's connected by a Cambridge Ring.

The system is presented in two levels of detail: First, an overview of UNIX United is given. Then the implementation, the Newcastle Connection layer, is described.

7.1 Overview

Any number of inter-linked UNIX system can be joined to compose a UNIX United system. Their naming structures (for files, devices, directories, and commands) are joined together into a single naming structure, in which each component system is to all intents and purposes just a directory. Ignoring for the moment questions regarding accreditation and access control, the resulting system is one in which each user can read or write any file, use any device, execute any command, or inspect any directory, regardless of the system to which it belongs. That is, network transparency is supported.

The component unit is a complete UNIX tree belonging to a certain machine. The position of these component units in the naming hierarchy is arbitrary. They can appear in the naming structure in positions subservient to other component units (directly or via intermediary directories). It is often convenient to set the naming structure to reflect organizational hierarchy of the environment in which the system exists.

In conventional UNIX the root of a file hierarchy is its own parent and is the only directory not assigned a string name. In UNIX United, each component's root is still referred to as '/' and still serves as the starting point of all pathnames starting with a '/'. Roots of component units, however, are assigned names so that they become accessible and distinguishable externally. Also, a subservient component can access its superior system by referring to its own root parent, (i.e., '/..'). Therefore, there is only one root that is its own parent and that is not assigned a string name; namely, the root of the composite name structure, which is just a virtual node needed to make the whole structure a single tree. Under this conventions, there is no notion of absolute pathname. Each pathname is relative to some context, either the current working directory or the current component unit.

In Figure 2, the directories **unix1**, **unix2**, **unix3**, and **unix4** are component units (i.e., complete UNIX hierarchies) belonging to machines by the same names. For instance, all the files descending from **unix2**, except files that descend from **unix4**, are stored on the machine unix2. The tree rooted at **unix4** descends from the directory **dir**, which is an ordinary (local) directory of **unix2**. To illustrate the relative pathnames, note that **/../unix2/f2** is the name of the file **f2** on the system **unix2** from within the **unix1** system. From the **unix3** system, the same file is referred to as **/../..unix2/f2**. Now, suppose the current root ('/') is as shown by the arrow. Then file **f3** can be referenced as **/f3**, file **f1** is referred to as **/../f1**, file **f2** is referred to as **/../../unix2/f2**, and finally file **f4** is referred to as **/../../unix2/dir/unix4/f4**.

Observe that users are aware of the upward boundaries of the current component unit since they must use the '/..' syntax whenever they wish to ascend outside of their current machine. Hence, UNIX United fails to provide complete location transparency.

The traditional root directories (e.g., **/dev**, **/bin**) are maintained for each machine separately. Because of the relative naming scheme, they are named, from within a component system, in the exact way as in conventional UNIX (e.g., just **/dev**). Each component system has its own set of named users and its own administrator (superuser). The latter is responsible for the accreditation for users of his or her own system as well as remote users. For uniqueness, remote users' identifiers are prefixed with the name of their original system. Accesses are governed by the standard UNIX file protection mechanisms, even if they cross component boundaries. That is, there is no need for users to log in separately or provide passwords when they access remote files if they are properly accredited. Accreditation for remote users must be arranged with the system administrator separately.

UNIX United is well suited for a diverse internetwork topology, spanning LANs, as well as direct links and even WANS. The logical name space needs to be properly mapped onto routing information in such a complex internetwork. An important design principle is that the naming hierarchy needs bear no relationship to the network topology.

7.2 Implementation—Newcastle Connection

The Newcastle Connection is a (user-level) software layer incorporated in each component system. This layer separates between the UNIX kernel on one hand, and applications, command programs and the shell on the other hand. It intercepts all system calls concerning files and filters out those that have to be redirected to remote systems. Also, the Connection layer accepts

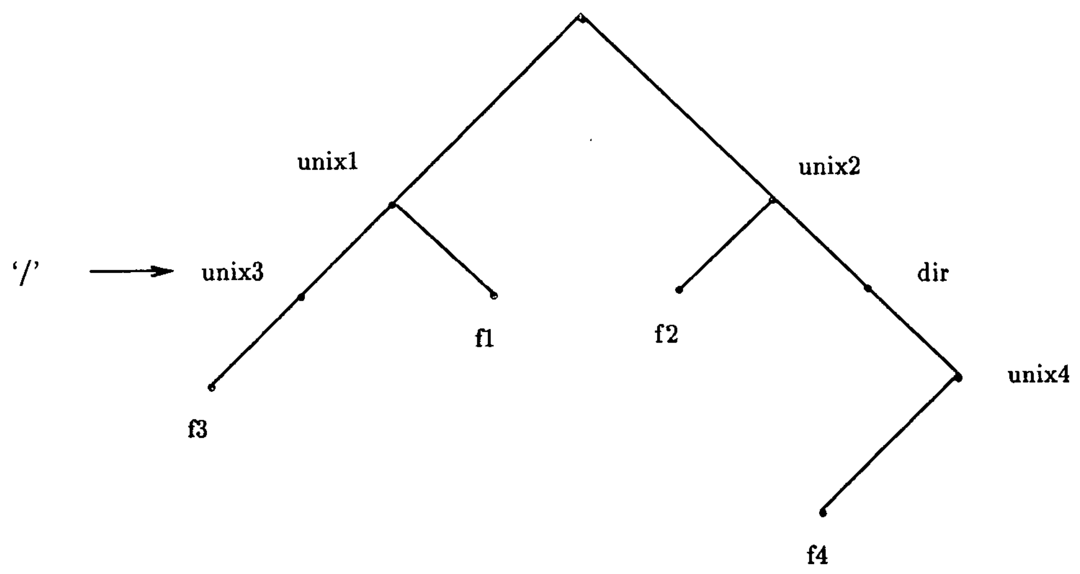

Figure 2. UNIX United hierarchy.

system calls that have been directed to it from other systems. Remote layers manage communication by the means of a RPC protocol. Figure 3 is a schematic view of the software architecture just described.

Incorporating the Connection layer preserves both the same UNIX system call interface and the UNIX kernel, in spite of the extensive remote activity carried out by the system. The penalty for preserving the kernel intact is the fact that the service is implemented as user-level daemon processes (as opposed to a kernel implementation), which slow down remote operation.

Each Connection layer stores a partial skeleton of the overall naming structure. Each system stores its own file system locally. In addition, each system maintains information on the fragments of the overall name structure that relate it to its neighboring systems in the naming structure (i.e., systems that can be reached via traversal of the naming tree without passing through another system). For instance, refer to Figure 2. System **unix2** is aware of the position of systems **unix1, unix2**, and **unix4** in the global tree. Figure 4 shows the relative positioning of the component units of the global name space that system **unix2** knows about.

The fragments maintained by different systems overlap and hence must remain consistent, a requirement that makes changing the overall structure a very expensive (and hence infrequent) event. Some leaves of the partial structure stored locally correspond to remote roots of other parts of the global file system. These leaves are specially marked and contain addresses of the appropriate storage sites of the descending file systems. Pathname traversals have to be continued remotely when encountering such marked leaves and, in fact, can span more than two systems until the target file is located. Therefore, a strict client-server pair model is not preserved. Once a name is resolved and the file is opened, it is accessed using file descriptors. The Connection layer marks descriptors that refer to remote files and keeps network addresses and routing information for them in a per-process table.

The actual remote file accesses are carried out by a set of file server processes on the target system. Each client has its own file server process with which it communicates directly. The initial connection is established with the aid of a spawner process that has a standard fixed name that makes it callable from any external process. This spawner process performs the remote access rights checks according to a machine-user identification pair. It also converts this identification to a valid local name. For the sake of preserving UNIX semantics, once a user process forks, its file service process

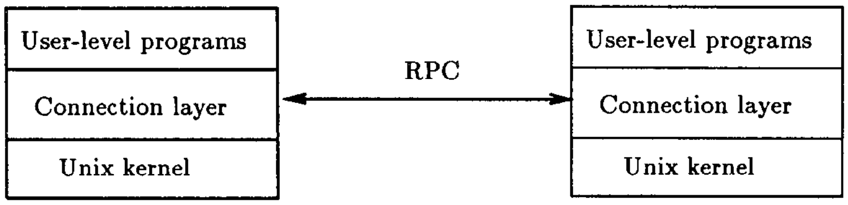

Figure 3. Schematic view of the UNIX United architecture.

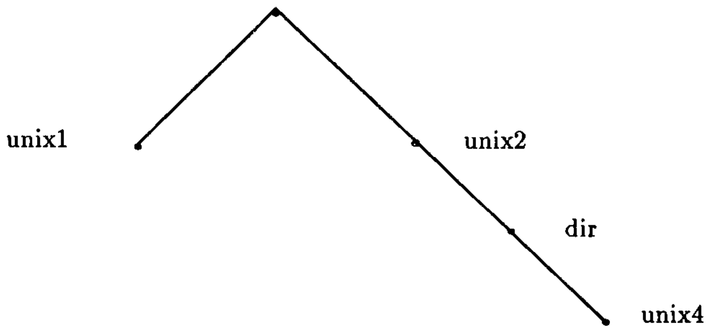

Figure 4. Partial skeleton UNIX2 has (see Figure 2).

forks as well. File descriptors (not lower level means such as i-nodes) are used to identify files between a user and its file server. This is a stateful service scheme and hence does not excel in terms of robustness.

7.3 Summary

The overall profile of the UNIX United system can be characterized by the following prominent features:

- *Logical Name Structure.* The UNIX United name structure is a hierarchy composed of component UNIX subtrees. There is an explicitly visible correspondence between a machine and a subtree in the structure; hence, machine boundaries are noticeable. Users must use the '/..' trap to get out of the current component unit. There are no absolute pathnames—all pathnames are relative to some context.
- *Recursive Structure.* Structuring a UNIX United system out of a set of component systems is a recursive process akin to a recursive definition of a tree. In theory, such a system can be indefinitely extensible. The building block of this recursive scheme is an autonomous and complete UNIX system.
- *Connection Layer.* Conceptually, the connection layer implementation is elegant and simple. It is a modular subsystem interfacing two existing layers without modifying either of them or their original semantics and still extending their capabilities by large. The implementation strategy is by relinking application programs with the Connection layer library routines. These routines intercept file system calls and forward the remote ones to user-level remote daemons at the remote sites.

Even though UNIX United is outdated, it serves our purposes well in demonstrating network transparency without location transparency, a simple implementation technique, and the issue of autonomy of component systems.

8. LOCUS

Locus is an ambitious project aimed at building a full-scale distributed operating system. The system is upward compatible with UNIX, but unlike NFS, UNIX United, and other UNIX-based distributed systems, the extensions are major ones and necessitate a new kernel rather than a modified one. Locus stands out among other systems by hosting a variety of sophisticated features such as automatic management of replicated data, atomic file update, remote tasking, ability to withstand (to a certain extent) failures and network partitions, and full implementation of nested transactions [Weinstein et al. 1985]. The system has been operational at UCLA for several years on a set of mainframes and workstations connected by an Ethernet. A general strategy for extending Locus to an internet environment is outlined in Sheltzer and Popek [1986].

The heart of the Locus architecture is its DFS. We first give an overview of the features and general implementation philosophy of the file system. Then we discuss the static nature of the file system (Sections 8.2) and its dynamics (Sections 8.3 and 8.4). We devote section 8.5 of the operation of the system in a faulty environment.

8.1 Overview

The Locus file system presents a single tree structure naming hierarchy to users and applications. This structure covers all objects (files, directories, executable files, and devices) of all the machines in the system. Locus names are fully location transparent; from a name of an object it is not possible to discern its location in the network. To a first approximation, there is almost no way to distinguish the Locus name structure from a standard UNIX tree.

A Locus file may correspond to a set of copies distributed on different sizes. An additional transparency dimension is introduced since it is the system responsibility to keep all copies up to date and assure that access requests are served by the most recently available version. As an option, users may have control over both the number and location of replicated files. In Locus, file replication serves mainly to increase availability for reading purposes. A primary copy approach is adopted for modifications.

Locus strives to provide UNIX semantics in the distributed and replicated environment in which it operates. Additionally, it supports locking of files and atomic update.

Fault tolerance issues are emphasized in Locus design. Network failure may disconnect the network into two or more partitions or disconnected subnetworks. As long as at least one copy of a file is available in a partition, read requests are served, and it is still guaranteed that the version read is the most recent one available in the partition. Automatic mechanisms take care to update stale copies of files upon the reconnection of partitions.

Emphasizing high performance in the design of Locus led to incorporating networking functions (such as formatting, queuing, transmitting, and retransmitting messages) into the operating system. Specialized remote operations protocols were devised for kernel-to-kernel communication, in contrast to the prevalent approach of using an off-the-shelf package (e.g., an RPC protocol). Lack of multilayering (as suggested in the ISO standard [Zimmermann 1980]) enabled achieving high performance for remote operations. On the other hand, this specialized protocol hampers Locus portability to different networks and file systems.

An efficient but limited kernel-supported LWP facility is devised for serving remote requests. These are processes that have no nonprivileged address space, their code and stack are resident in the operating system nucleus, and they can call internal system routines directly. These processes are assigned to serve network requests that accumulate in a system queue. The system is configured with some number of these processes, but that number is automatically and dynamically altered during system operation.

8.2 Name Structure

The logical name structure disguises both location and replication details from users and applications. A removable file system, in Locus terms, is called a *filegroup*. A filegroup is the component unit in Locus. Virtually, logical filegroups are joined together to form this unified structure. Physically, a logical filegroup is mapped to *multiple physical containers* (called also *packs*) residing at various sites and storing replicas of the files of that filegroup. The pair <logical filegroup number, i-node

number>, referred to as a *file designator* serves as low-level, location-independent file identifier. The designator itself hides both location and replication details, since it points to a file in general and not to a particular replica.

Each site has a consistent and complete view of the logical name structure. A logical mount table is globally replicated and contains an entry for each logical filegroup. The entry records the file designator of the directory over which the filegroup is logically mounted and an indication of which site is currently responsible for access synchronization (the function of this site is explained subsequently) within the filegroup. A protocol, implemented within the mount and unmount Locus system calls, performs update of the logical mount tables on all sites when necessary.

On the physical level, physical containers correspond to disk partitions. One of the packs is designated as the *Primary Copy.* A file must be stored at the site of the primary copy and, in addition, can be stored at any subset of the other sites where there exists a pack corresponding to its filegroup. Thus, the primary copy stores the filegroup completely, whereas the rest of the packs might be partial. Replication is especially useful for directories in the high levels of the name hierarchy. Such directories are rarely updated and are crucial for pathnames translation of files.

The various copies of a file are assigned to the same i-node number on all the filegroup's packs. Consequently, a pack has an empty i-node slot for all files it does not store. Data page numbers may be different on different packs, hence reference over the network to data pages use logical page numbers rather than physical ones. Each pack has a mapping of these logical numbers to physical numbers. To facilitate automatic replication management, each i-node of a file copy contains a version number, determining which copy dominates other copies.

Whereas globally unique file naming is very important most of the time, certain files and directories are hardware and site specific (e.g., /**bin** is hardware-specific, and /**dev** is site-specific). Locus provides transparent means for translating references to these traditional file names to hardware- and site-specific files.

8.3 File Operations

In contrast to the prevalent model of a server-client pair involved in a file access, Locus distinguishes three logical roles in file accesses; each one potentially performed by a different site:

- *Using Site* (US) issues the requests to open and access a remote file.
- *Storage Site* (SS) is the selected site to serve the requests.
- *Current Synchronization Site* (CSS) enforces a global synchronization policy for a filegroup and selects an SS for each Open request referring to a file in the filegroup. There is at most one CSS for each filegroup in any set of communicating sites (i.e., partition). The CSS maintains the version number and a list of physical containers for every file in the filegroup.

The following sections describe the file operations as they are carried out by the above entities. Related synchronization issues are described in Section 8.3.4.

8.3.1 Opening and Reading a File

We first describe how a file is opened and read given its designator and then describe how a designator is obtained from a string pathname.

Given a file designator, opening the file commences as follows. The US determines the relevant CSS by looking up the filegroup in the logical mount table, then forwards the Open request to the CSS. The CSS polls potential SSs for that file to decide which one will act as the real SS. In its polling messages, the CSS includes the version number for the particular file so the potential SSs can, by comparing this number to their own, decide whether or not their copy is up to date. The CSS selects an SS by considering the response it got back from the candidate sites and sends the selected SS identity to the US. Both the

CSS and the SS allocate in-core i-node structures for the opened file. The CSS needs this information to make future synchronization decisions, and the SS maintains the i-node to serve forthcoming accesses efficiently.

After a file is open, a Read request is sent directly to the SS without the CSS intervention. A Read request contains the designator of the file, the logical number of the needed page within that file, and a hint as to where the SS might store the file's i-node in main memory. Once the i-node is found, the SS translates the logical page number to physical number, and a standard low-level routine is called to allocate a buffer and get the appropriate page from disk. The buffer is queued on the network queue for transmission back to the US as a response, where it is stored in a kernel buffer. Once a page is fetched to the US, further Read calls are serviced from the kernel buffer. As in the case of local disk Reads, read-ahead is useful to speed up sequential reading, both at the US and the SS. If a process loses its connection with a file it is reading remotely, the system attempts to reopen a different copy of the same version of the file.

Translating a pathname into a file designator is carried out by seemingly conventional pathname traversal mechanism since pathnames are regular UNIX pathnames, with no exception (unlike UNIX United). Every lookup of a component of the pathname within a directory involves opening the latter and reading from it. These operations are conducted according to the above protocols (i.e., directory entries are also cached in US buffers). There is no parallel to NFS's remote lookup operation. The actual directory searching is performed by the client rather than by the server. A directory opened for pathname searching is not open for normal Read, but instead for an internal unsynchronized Read. The distinction is that no global synchronization is needed and no locking is done while the reading is performed; that is, updates to the directory can occur while the search is ongoing. When the directory is local, even the CSS is not informed of such access.

8.3.2 Modifying a File

In Locus, a primary copy policy is used for file modifications. The CSS must select the primary copy pack site as the SS if the Open is for a Write. The act of modifying data takes on two forms. If the modification does not include the entire page, the old page is first read from the SS using the Read protocol. If an entire page is modified, a buffer is set up at the US without any reads. In either case, after changes are made, possibly by delayed-write, the page is sent back to the SS. All modified pages must be flushed to the SS before a modified file can be closed.

If a file is closed by the last user process at a US, the SS and CSS must be informed so that they can deallocate in-core i-node structures and the CSS can alter state data that might affect its next synchronization decision.

Caching of data pages is relied upon heavily in both Read and Write operations. The validation of the cached data is dealt with in Section 8.3.4.

8.3.3 Commit and Abort

Locus uses the following *shadow page* mechanism for implementing atomic commit. When a file is modified, disk pages are allocated at the SS; these pages are the shadow pages. The in-core copy of the disk i-node is updated to point to the shadow pages. The disk i-node is kept intact, pointing to the original pages. To abort a set of changes, both the in-core i-node information and the shadow pages used to record the changes are discarded. The atomic Commit operation consists of moving the incore i-node to the disk i-node. After that, the file contains the new information. The US function never deals with actual disk pages, but rather with logical pages. Thus, the entire shadow page mechanism is implemented at the SS and is transparent to the US.

Locus deals with file modification by first committing the change to the primary copy. Later, messages are sent to all other SSs and to the CSS. At a minimum, these messages identify the modified file and contain the new version number (in order to prevent attempts to read the old version). It is the responsibility of these additional SSs to bring their version up to date by propagating the entire file or just the changes. A queue of propagation requests is kept within the kernel at each site, and a kernel process services the queue efficiently by issuing appropriate Read requests. This propagation procedure uses the standard commit mechanism. Thus, if contact with the file containing the newer version is lost, the local file is left with a coherent copy, albeit still out of date. Given this commit mechanism, one is always left with either the original file or a completely changed file, but never with a partially made change, even in the face of site failures.

8.4 Synchronizing Accesses to Files

The default synchronization policy in Locus is to emulate UNIX semantics on file accesses in a distributed environment. UNIX semantics can be implemented fairly easily by having the processes share the same operating system data structures and caches and by using locks on data structures to serialize requests. In Locus the sharing processes may not co-reside on the same machine, and hence the implementation is more complicated.

Recall that UNIX semantics allow several processes descending from the same ancestor process to share the same current position (offset) in a file. A single token scheme is devised to preserve this special mode of sharing. Only when the token is present, can a site proceed with executing system calls needing the offset.

In UNIX, the same in-core i-node for a file can be shared by several processes. In Locus, the situation is much more complicated since the i-node of the file, as well as the parts of the file itself, can be cached at several sites. Token schemes are used to synchronize sharing of a file's i-node and data. An exclusive-writer-multiple-readers policy is enforced. Only a site with a write token for a file may modify the file; any site with a read token can read it. The token schemes are coordinated by token managers operating at the corresponding storage sites.

The cached data pages are guaranteed to contain valid data only when the files's data token is present. When the write data token is taken from that site, the i-node, as well as all modified pages, is copied back to the SS. Since arbitrary changes (initiated by remote clients) may have occurred when the token was not present, all cached buffers are invalidated when the token is released. When a data token is granted to a site, both the i-node and data pages need to be fetched from the SS. There are some exceptions to enforcing this policy. Some attribute reading and writing calls (e.g., **stat**) as well as directory reading and modifying (e.g., **lookup**) calls are not subject to the synchronization constraints. These calls are sent directly to the SS, where the changes are made, committed, and propagated to all storage and using sites.

Alternatively to the default UNIX semantics, Locus offers facilities for locking entire files or parts of them. Locking can be advisory (only checked as a result of a locking attempt) or enforced (checked on all reads and writes). A process can choose to either fail if it cannot immediately get a lock or wait for it to be released.

8.5 Operation in a Faulty Environment

The basic approach in Locus is to maintain, within a single partition, consistency among copies of a file. The policy is to allow updates only in a partition that has the primary copy. It is guaranteed that the most recent version of a file in a partition is read. The latter guarantee applies to all partitions.

A central point addressed in this section is the reconciliation of replicated filegroups residing at partitioned sites. During normal operation, the commit protocol ascertains proper propagation of updates as described earlier. A more elaborate scheme has to be used by recovering sites wishing to bring their packs up to date. To this end, the system maintains a *commit count* for each filegroup, enumerating each commit of every file in the filegroup. Each pack has a *lower-water-mark* (lwm) that is a commit count value up to which the system guarantees that all prior commits are reflects in the pack. Also, the primary copy pack (usually stored at the CSS) keeps a list enumerating the files in the filegroup and the corresponding commit counts of all the recent commits in secondary storage. When a pack joins a partition it attempts to contact the CSS and checks whether its lwm is within the recent commit list range. If this is the case, the pack site schedules a kernel process that brings the pack to a consistent state by copying only the files that reflect commits later than that of the site's lwm. If the CSS is not available, writing is disallowed in this partition, but reading is possible after a new CSS is chosen. The new CSS communicates with the partition members so it will be informed of the most recent available (in the partition) version of each file in the filegroup. Once the new CSS accomplishes the objective, other pack sites can reconcile themselves with it. As a result, all communicating sites see the same view of the filegroup, and this view is as complete as possible, given a particular partition. Note that since updates are allowed within the partition with the primary copy and Reads are allowed in the rest of the partitions, it is possible to Read out-of-date replicas of a file. Thus, Locus sacrifices consistency for the ability to continue to both update and read files in a partitioned environment.

When a pack is too far out of date (i.e., its lwm indicates a prior value to the earliest commit count value in the primary copy commit list), the system invokes an application-level process to bring the filegroup up to date. At this point, the system lacks sufficient knowledge of the most recent commits to identify the missing updates. Instead, the site must inspect the entire i-node space to determine which files in its pack are out of date.

When a site is lost from an operational Locus network, a clean-up procedure is necessary. Essentially, once site **a** has decided that site **b** is unavailable, site **a** must invoke failure handling for remote resources that processes local to **a** were using at site **b**, and for all local resources being used by processes local to site **b**. This substantial cleaning procedure is the penalty of the state information kept by all three sites participating in file access.

Since directory updates are not restricted to be applied to the primary copy, conflicts among updates in different partitions may arise [Walker et al. 1983]. Because of the simple nature of directory modification, however, an automatic reconciliation procedure is devised. This procedure is based on comparing the i-nodes and string name pairs of replicas of the same directory. The most extreme action taken is when the same name string corresponds to two different i-nodes (i.e., the same name is used for creating two different files) and amounts to altering the file names slightly and notifying the files owners by electronic mail.

8.6 Summary

An overall profile and evaluation of Locus is summarized by pointing out the following issues:

- *Distributed operating system.* Because of the multiple dimensions of transparency in Locus, it comes close to the definition of a truly distributed operating system in contrast to a collection of network services [Tanenbaum and Van Renesse 1985].
- *Implementation strategy.* Essentially, kernel augmentation is the implementation strategy in Locus. The common pattern in Locus is kernel-to-kernel communication via specialized, high-performance protocols. This strategy is needed to support the philosophy of a distributed operating system.
- *Replication.* A primary copy replication scheme is used in Locus. The main merit of this kind of replication scheme is increased availability of directories that exhibit high read-write ratio. Availability for modifying files is not increased by the primary copy approach. Handling replication transparently is one of the reasons for introducing the CSS entity, which is

a third entity taking part in a remote access. In this context, the CSS functions as the mapping from an abstract file to a physical replica.

- *Access synchronization.* UNIX semantics are emulated to the last detail, in spite of caching at multiple USs. Alternatively, locking facilities are provided.
- *Fault tolerance.* Substantial effort has been devoted to designing mechanisms for fault tolerance. A few are an atomic update facility, merging replicated packs after recovery, and a degree of independent operation of partitions. The effects can be characterized as follows:
  - Within a partition, the most recent, available version of a file is read. The primary copy must be available for write operations.
  - The primary copy of a file is always up to date with the most recent committed version. Other copies may have either the same version or an older version, but never a partially modified one.
  - A CSS function introduces an additional point of failure. For a file to be available for opening, both the CSS for the filegroup and an SS must be available.
  - Every pathname component must be available for the corresponding file to be available for opening.

A basic questionable decision regarding fault tolerance is the extensive use of incore information by the CSS and SS functions. Supporting the synchronization policy is a partial cause for maintaining this information; however, the price paid during recovery is enormous. Besides, explicit deallocation is needed to reclaim this incore space, resulting in a pure overhead of message traffic.

- *Scalability.* Locus does not lend itself to very large distributed system environment, mainly because of the following reasons:
  - One CSS per file group can easily become a bottleneck for heavily accessed filegroups.
  - A logical mount table replicated at all sites is clearly not a scalable mechanism.
  - Extensive message traffic and server load caused by the complex synchronization of accesses needed to provide UNIX semantics.
- *UNIX compatibility.* The way Locus handles remote operation is geared to emulation of standard UNIX. The implementation is merely an extension of UNIX implementation across a network. Whenever buffering is used in UNIX, it is used in Locus as well. UNIX compatibility is indeed retained; however, this approach has some inherent flaws. First, it is not clear whether UNIX semantics are appropriate. For instance, the mechanism for supporting shared file offset by remote processes is complex and expensive. It is unclear whether this peculiar mode of sharing justifies this price. Second, using caching and buffering as done in UNIX in a distributed system has some ramifications on the robustness and recoverability of the system. Compatibility with UNIX is indeed an important design goal, but sometimes it obscures the development of an advanced distributed and robust system.

9. SUN NETWORK FILE SYSTEM

The Network File System (NFS) is a name for both an implementation and a specification of a software system for accessing remote files across LANs. The implementation is part of the SunOS operating system, which is a flavor of UNIX running on Sun workstations using an unreliable datagram protocol (UDP/IP protocol [Postel 1980]) and Ethernet. The specification and implementation are intertwined in the following description; whenever a level of detail is needed we refer to the SunOS implementation, and whenever the description is general enough it also applies to the specification.

The system is presented in three levels of detail. First (in Section 9.1), an overview is given. Then, two service protocols that are the building blocks for the implementation are examined (Section 9.2). Finally (in Section 9.3), a description of the SunOS implementation is given.

9.1 Overview

NFS views a set of interconnected workstations as a set of independent machines with independent file systems. The goal is to allow some degree of sharing among these file systems in a transparent manner. Sharing is based on server-client relationship. A machine may be, and often is, both a client and a server. Sharing is allowed between any pair of machines, not only with dedicated server machines. Consistent with the independence of a machine is the critical observation that NFS sharing of a remote file system affects only the client machine and no other machine. Therefore, there is no notion of a globally shared file system as in Locus, Sprite, UNIX United, and Andrew.

To make a remote directory accessible in a transparent manner from a client machine, a user of that machine first has to carry out a mount operation. Actually, only a superuser can invoke the mount operation. Specifying the remote directory as an argument for the mount operation is done in a nontransparent manner; the location (i.e., hostname) of the remote directory has to be provided. From then on, users on the client machine can access files in the remote directory in a totally transparent manner, as if the directory were local. Since each machine is free to configure its own name space, it is not guaranteed that all machines have a common view of the shared space. The convention is to configure the system to have a uniform name space. By mounting a shared file system over user home directories on all the machines, a user can log in to any workstation and get his or her home environment. Thus, user mobility can be provided, although again by convention.

Subject to access rights accreditation, potentially any file system or a directory within a file system can be remotely mounted on top of any local directory. In the latest NFS version, diskless workstations can even mount their own roots from servers (Version 4.0, May 1988 described in Sun Microsystems Inc. [1988]). In previous NFS versions, a diskless workstation depends on the Network Disk (ND) protocol that provides raw block I/O service from remote disks; the server disk was partitioned and no sharing of root file systems was allowed.

One of the design goals of NFS is to provide file services in a heterogeneous environment of different machines, operating systems, and network architecture. The NFS specification is independent of these media and thus encourages other implementations. This independence is achieved through the use of RPC primitives built on top of an External Date Representation (XDR) protocol—two implementation-independent interfaces [Sun Microsystems Inc. 1988]. Hence, if the system consists of heterogeneous machines and file systems that are properly interfaced to NFS, file systems of different types can be mounted both locally and remotely.

9.2 NFS Services

The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote file access services. Accordingly, two separate protocols are specified for these services—a mount protocol and a protocol for remote file accesses called the NFS protocol. The protocols are specified as sets of RPCs that define the protocols' functionality. These RPCs are the building blocks used to implement transparent remote file access.

9.2.1 Mount Protocol

We first illustrate the semantics of mounting by a series of examples. In Figure 5a, the independent file systems belonging to the machines named client, server1, and server2 are shown. At this stage, at each machine only the local files can be accessed. The triangles in the figure represent subtrees of directories of interest in this

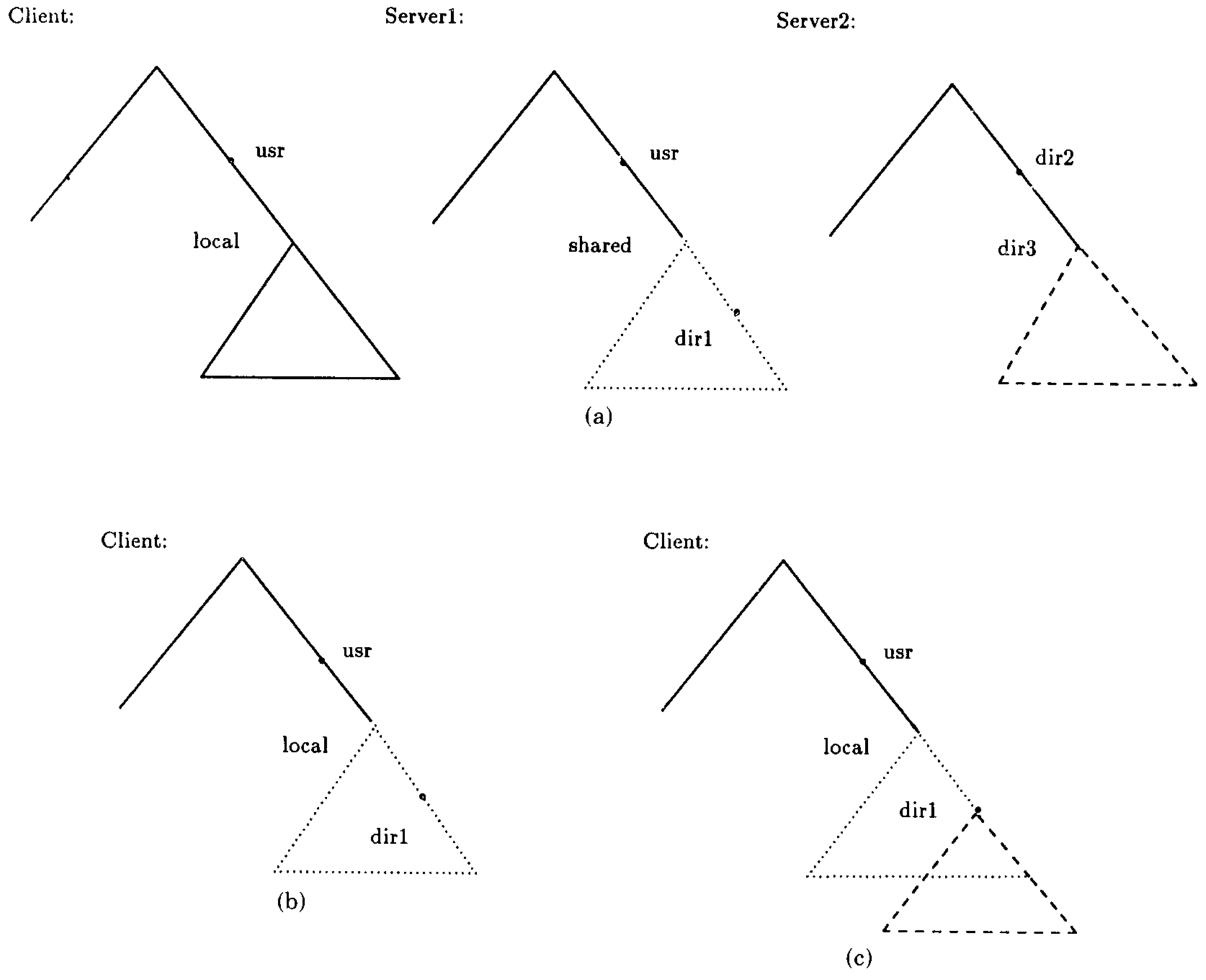

Figure 5. NFS joins independent file systems (a), by mounts (b), and cascading mounts (c).

example. In Figure 5b, the effects of the mounting **server1:/usr/shared** over **client:/usr/local** are shown. This figure depicts the view users on client have of their file system. Observe that any file within the **dir1** directory, for instance, can be accessed using the prefix **/usr/local/dir1** in client after the mount is complete. The original directory **/usr/local** on that machine is not visible any more.

Cascading mounts are also permitted. That is, a file system can be mounted over another file system that is not a local one, but rather a remotely mounted one. A machine's name space, however, is affected only by those mounts the machine's own superuser has invoked. By mounting a remote file system, access is not gained for other file systems that were, by chance, mounted over the former file system. Thus, the mount mechanism does not exhibit a transitivity property. In Figure 5c we illustrate cascading mounts by continuing our example. The figure shows the result of mounting **server2:/dir2/dir** over **client:/usr/local/dir1**, which is already remotely mounted from **server1**. Files within **dir3** can be accessed in client using the prefix **/usr/local/dir1**.

The mount protocol is used to establish the initial connection between a server and a client. The server maintains an export list (the **/etc/exports** in UNIX) that specifies the local file systems it exports for mounting, along with names of machines permitted to mount them. Any directory within an exported file system can be remotely mounted by an accredited machine. Hence, a component unit is such a directory. When the server receives a mount request that conforms to its export list, it returns to the client a file handle that is the key for further accesses to files within the mounted file system. The file handle contains all the information the server needs to distinguish individual files it stores. In UNIX terms, the file handle consists of a file system identifier and an i-node number to identify the exact mounted directory within the exported file system.

The server also maintains a list of the client machines and the corresponding currently mounted directories. This list is mainly for administrative purposes, such as for notifying all clients that the server is going down. Adding and deleting an entry in this list is the only way the server state is affected by the mount protocol.

Usually a system has some static mounting preconfiguration that is established at boot time; however, this layout can be modified (**/etc/fstab** in UNIX).

9.2.2 NFS Protocol

The NFS protocol provides a set of remote procedure calls for remote file operations. The procedures support the following operations:

- Searching for a file within a directory (i.e., lookup).
- Reading a set of directory entries.
- Manipulating links and directories.
- Accessing file attributes.
- Reading and writing files.

These procedures can be invoked only after having a file handle for the remotely mounted directory. Recall that the mount operation supplies this file handle.

The omission of Open and Close operations is intentional. A prominent feature of NFS servers is that they are stateless. There are no parallels to UNIX's open files table or file structures on the server side. Maintaining the clients list mentioned in Section 9.2.1 seems to violate the statelessness of the server. The client list, however, is not essential in any manner for the correct operation of the client or the server and hence need not be restored after a server crash. Consequently, this list might include inconsistent data and should be treated only as a hint.

A further implication of the stateless server philosophy and a result of the synchrony of an RPC is that modified data (including indirection and status blocks) must be committed to the server's disk before the call returns results to the client. The NFS protocol does not provide concurrency control mechanisms. The claim is that since locks management is inherently stateful, a service outside the NFS should provide locking. It is advised that users would coordinate access to shared files using mechanisms outside the scope of NFS (e.g., by means provided in a database management system).

9.3 Implementation

In general, Sun's implementation of NFS is integrated with the SunOS kernel for reasons of efficiency (although such integration is not strictly necessary). In this section we outline this implementation.

9.3.1 Architecture

The NFS architecture is schematically depicted in Figure 6. The user interface is the UNIX system calls interface based on the Open, Read, Write, Close calls, and file descriptors. This interface is on top of a middle layer called the Virtual File System (VFS) layer. The bottom layer is the one that implements the NFS protocol and is called the NFS layer. These layers comprise the NFS software architecture. The figure also shows the RPC/XDR software layer, local file systems, and the network and thus can serve to illustrate the integration of a DFS with all these components. The VFS serves two important functions:

- It separates file system generic operations from their implementation by defining a clean interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to a variety of types of file systems mounted locally (e.g., 4.2 BSD or MS-DOS).

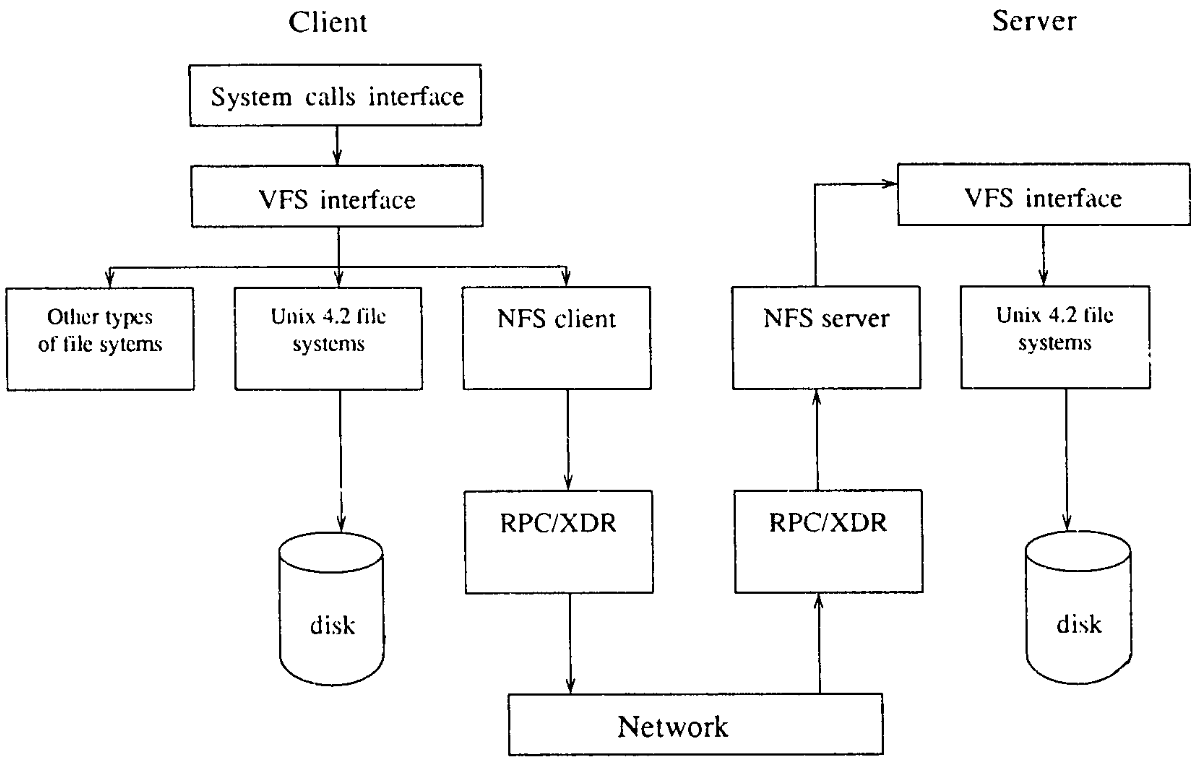

Figure 6. Schematic view of the NFS architecture.

- The VFS is based on a file representation structure called a *vnode*, which contains a numerical designator for a file that is networkwide unique. (Recall that UNIX-i-nodes are unique only within a single file system.) The kernel maintains one vnode structure for each active node (file or directory). Essentially, for every file the vnode structures complemented by the mount table provide a pointer to its parent file system, as well as to the file system over which it is mounted.

Thus, the VFS distinguishes local files from remote ones, and local files are further distinguished according to their file system types. The VFS activates file system specific operations to handle local requests according to their file system types and calls the NFS protocol procedures for remote requests. File handles are constructed from the relevant vnodes and passed as arguments to these procedures.

As an illustration of the architecture, let us trace how an operation on an already open remote file is handled (follow the example in Figure 6). The client initiates the operation by a regular system call. The operating system layer maps this call to a VFS operation on the appropriate vnode. The VFS layer identifies the file as a remote one and invokes the appropriate NFS procedure. An RPC call is made to the NFS service layer at the remote server. This call is reinjected into the VFS layer, which finds that it is local and invokes the appropriate file system operation. This path is retraced to return the result. An advantage of this architecture is that the client and the server are identical; thus, it is possible for a machine to be a client, or a server, or both.

The actual service on each server is performed by several kernel processes, which provide a temporary substitute to a LWP facility.

9.3.2 Pathname Translation

Pathname translation is done by breaking the path into component names and doing a separate NFS lookup call for every pair of component name and directory vnode. Thus, lookups are performed remotely by the server. Once a mount point is crossed, every component lookup causes a separate RPC to the server. This expensive pathname traversal scheme is needed, since each client has a unique layout of its logical name space, dictated by the mounts if performed. It would have been much more efficient to pass a pathname to a server and receive a target vnode once a mount point was encountered. But at any point there can be another mount point for the particular client of which the stateless server is unaware.

To make lookup faster, a directory name lookup cache at the client holds the vnodes for remote directory names. This cache speeds up references to files with the same initial pathname. The directory cache is discarded when attributes returned from the server do not match the attributes of the cached vnode.

Recall that mounting a remote file system on top of another already mounted remote file system (cascading mount) is allowed in NFS. A server cannot, however, act as an intermediary between a client and another server. Instead, a client must establish a direct server-client connection with the second server by mounting the desired server directory. Therefore, when a client does a lookup on a directory on which the server has mounted a file system, the client sees the underlying directory instead of the mounted directory. When a client has a cascading mount, more than one server can be involved in a pathname traversal. Each component lookup is, however, performed between the original client and some server.

9.3.3 Caching and Consistency

With the exception of opening and closing files, there is almost a one-to-one correspondence between the regular UNIX system calls for file operations and the NFS protocol RPCs. Thus, a remote file operation can be translated directly to the corresponding RPC. Conceptually, NFS adheres to the remote service paradigm, but in practice buffering and caching techniques are used for the sake of performance. There is no direct correspondence between a remote operation and an RPC. Instead, file blocks and file attributes are fetched by the RPCs and cached locally. Future remote operations use the cached data subject to some consistency constraints.

There are two caches: file blocks cache and file attribute (i-node information) cache. On a file open, the kernel checks with the remote server about whether to fetch or revalidate the cached attributes by comparing time stamps of the last modification. The cached file blocks are used only if the corresponding cached attributes are up to date. The attribute cache is updated whenever new attributes arrive from the server after a cache miss. Cached attributes are discarded typically after 3 s for files or 30 s for directories. Both read-ahead and delayed-write techniques are used between the server and the client [Sun Microsystems Inc. 88]. (Earlier version of NFS used write-on-close [Sandberg et al. 1985]). The caching unit is fairly large (8Kb) for performance reasons. Clients do not free delayed-write blocks until the server confirms the data are written to disk. In contrast to Sprite, delayed-write is retained even when a file is open concurrently in conflicting modes. Hence, UNIX semantics are not preserved.

Tuning the system for performance makes it difficult to characterize the sharing semantics of NFS. New files created on a machine may not be visible elsewhere for 30 s. It is indeterminate whether writes to a file at one site are visible to other sites that have the file open for reading. New opens of that file observe only the changes that have already been flushed to the server. Thus, NFS fails to provide either strict emulation of UNIX semantics or any other clear semantics.

9.4 Summary

- *Logical name structure.* A fundamental observation is that every machine establishes its own view of the logical name structure. There is no notion of global name hierarchy. Each machine has its own root serving as a private and absolute point of reference for its own view of the name structure. Selective mounting of parts of file systems upon explicit request allows each machine to obtain its unique view of the global file system. As a result, users enjoy some degree of independence, flexibility, and privacy. It seems that the penalty paid for this flexibility is administrative complexity.

- *Network service versus distributed operating system.* NFS is a network service for sharing files rather than an integral component of a distributed operating system [Tanenbaum and Van Renesse 1985]. This characterization does not contradict the SunOS kernel implementation of NFS, since the kernel integration is only for performance reasons. Being a network service has two main implications. First, remote-file sharing is not the default; the service initiating remote sharing (i.e., mounting) has to be explicitly invoked. Moreover, the first step in accessing a remote file, the mount call, is a location dependent one. Second, perceiving NFS as a service and not as part of the operating system allows its design specification to be implementation independent.
- *Remote service.* Once a file can be accessed transparently I/O operations are performed according to the remote service method: The data in the file are not fetched en masse; instead, the remote site potentially participates in each Read and Write operation. NFS uses caching to improve performance, but the remote site is conceptually involved in every I/O operation.
- *Fault tolerance.* A novel feature of NFS is the stateless approach taken in the design of the servers. The result is resiliency to client, server, or network failures. Should a client fail, it is not necessary for the server to take any action. Once caching was introduced, various patches had to be invented to keep the cached data consistent without making the server stateful.
- *Sharing semantics.* NFS does not provide UNIX semantics for concurrently open files. In fact, the current semantics cannot be characterized clearly, since they are timing dependent.

Finally, it should be realized that NFS is commercially available, has very reasonable performance, and is perceived as a de facto standard in the user community.

10. SPRITE

Sprite is an experimental, distributed operating system under development at the University of California at Berkeley. It is part of the Spur project, whose goal is the design and construction of high-performance multiprocessor workstation [Hill et al. 1986]. A preliminary version of Sprite is currently operational on interconnected Sun workstations.

Section 10.1 gives an overview of the file system and related aspects. Section 10.2 elaborates on the file lookup mechanism (called prefix tables) and Section 10.3 on the caching methods used in the file system.

10.1 Overview

Sprite designers envision the next generation of workstations as powerful machines with vast main memory. Currently, workstations have 4 to 32Mb of main memory. Sprite designers predict that memories of 100 to 500Mb will be commonplace in a few years. Their claim is that by caching files from dedicated servers, the large physical memories can compensate for lack of local disks in clients' workstations.

The interface that Sprite provides in general and to the file system in particular is much like the one provided by UNIX. The file system appears as a single UNIX tree encompassing all files and devices in the network, making them equally and transparently accessible from every workstation. As with Locus, the location transparency is complete; there is no way to discern a file's network location from its name. Sprite enforces UNIX semantics for share files.

In spite of its functional similarity to UNIX, the Sprite kernel was developed from scratch. Oriented toward multiprocessing, the kernel is multithreaded. Synchronization between the multiple threads is based on monitorlike structures with many small locks protecting the shared data [Hoare 1974]. Network integration is based on a simple kernel-to-kernel RPC facility implemented on top of a special-purpose network protocol. The technique used in the protocol is implicit acknowledgment, originally discussed in Birrel and Nelson [1984].

A unique feature of the Sprite file system is its interplay with the virtual memory system. Most versions of UNIX use a special disk partition as a swapping area for virtual memory purposes. In contrast, Sprite uses ordinary files (called *backing files*) to store data and stacks of running processes. The motivation for this design is that it simplifies process migration and enables flexibility and sharing of the space allocated for swapping. Backing files are cached in the main memories of servers, just like any other file. It is claimed that clients would be able to read random pages from a server's (physical) cache faster than from a local disk, which means that a server with a large cache may provide better paging performance than from a local disk. The virtual memory and file system share the same cache, which is dynamically partitioned according to their conflicting needs. Sprite allows the file cache on each machine to grow and shrink in response to changing demands of the machine's virtual memory and file system. Among other features of Sprite are support for user LWPs and a process migration facility, which is transparent both to users and the migrated process.

10.2 Looking Up Files with Prefix Tables

Sprite presents its user with a single file system hierarchy. The hierarchy is composed of several subtrees called *domains* (the Sprite term for component unit), with each server providing storage for one or more domains. Each machine maintains a server map called a *prefix table*, whose function is to map domains to servers [Welch and Ousterhout 1986]. The mapping is built and updated dynamically by a broadcast protocol. We first describe how the tables are used during name lookups, then describe how the tables change dynamically.

Each entry in a prefix table corresponds to one of the domains. It contains the pathname of the topmost directory in the domain (that pathname is called the *prefix* for the domain), the network address of the server storing the domain, and a numeric designator identifying the domain's root directory for the storing server. This designator is an index into the server table of open files; it saves repeating expensive name translation.

Every lookup operation for an absolute pathname starts with the client searching its prefix table for the longest prefix matching the given file name. The client strips the matching prefix from the file name and sends the remainder of the name to the selected server along with the designator from the prefix table entry. The server uses this designator to locate the root directory of the domain, then proceeds by usual UNIX pathname translation for the remainder of the file name. If the server succeeds in completing the translation, it replies with a designator for the open file.

There are several cases in which the server does not complete the lookup. For instance, a pathname can descend down into a new domain. This can happen when an entry for a domain is absent from the table and, as a result, the prefix of the domain above the missing domain is the longest matching prefix. The selected server cannot complete the pathname traversal since it descends outside its domain. The solution to this problem is to place a marker to indicate domain boundaries (a mount point). The marker is a special kind of file called a *remote link*. Similar to a symbolic link, its content is a file name—its own name in this case. When a server encounters a remote link, it returns the file name to the client.

So far, the key difference from mappings based on the UNIX mount mechanism is the initial step of matching the file name against the prefix table instead of looking it up component by component. Systems (such as NFS and conventional UNIX) that use a name lookup cache get a similar effect of avoiding the component-by-component lookup once the cache holds the appropriate information. Prefix tables are, however, a unique mechanism mainly because of the way they evolve and change. When a remote link is encountered by the server, it indicates that the client lacks an entry for a domain—the domain whose remote link was encountered. To obtain the missing prefix information, a client broadcasts a file name. A server storing that file responds with the prefix table entry for this file, including the string to use as a prefix, the server's address, and the descriptor corresponding to the domain's root. The client can then fill in the details in its prefix table.

Initially, each client starts with an empty prefix table. The broadcast protocol is invoked to find the entry for the root domain. More entries are added as needed; a domain that has never been accessed will not appear in the table.

The server locations kept in the prefix table are hints that are corrected when found to be wrong. Hence, if a client tries to open a file and gets no response from the server, it invalidates the prefix table entry and issues a broadcast query. If the server has become available again, it responds to the broadcast and the prefix table entry is reestablished. This same mechanism also works if the server reboots at a different network address or if its domains are migrated to other servers.

The prefix mechanism ensures that whenever a server storing a domain is up, the domain's files can be accessed regardless of the status of servers storing domains that appear in the pathname of the accessed files. In essence, the built-in broadcast protocol enables dynamic reconfiguration and a certain degree of robustness. Also, when a prefix for a domain exists in a client's table, a direct client-server connection is established as soon as the client attempts to open a file in that domain (in contrast to pathname traversal schemes).

A machine with a local disk wishing to keep some local files private can accomplish this by placing an entry for the private domain in its prefix table and refusing to respond to broadcast queries about it. One of the uses of this provision can be for the directory **/usr/tmp**, which holds temporary files generated by many UNIX programs. Every workstation needs access to **/usr/tmp**. But workstations with local disks would probably prefer to use their own disk for the temporary space. They can set up their **/usr/tmp** domains for private use, with a network file server providing a public version of the domain for diskless clients. All broadcast queries for **/usr/tmp** would be handled by the public server.

A primitive form of read-only replication can also be provided. It can be arranged so that servers storing a replicated domain give different clients different prefix entries (standing for different replicas) for the same domain. As a result, the service load is divided among the servers as each replica serves a different set of clients. The same technique can be used for sharing binary files by different hardware types of machines.

Since the prefix tables bypass part of the director lookup mechanism, the permission checking done during lookup is bypassed too. The effect is that all programs implicitly have search permission along all the paths denoting prefixes of domains. If access to a domain is to be restricted, it must be restricted at the root of the domain or below it.

10.3 Caching and Consistency

An important aspect of the Sprite file system design is the extent to which it uses using caching techniques. Capitalizing on the large main memories and advocating diskless workstations, file caches are stored incore. The same caching scheme is used to avoid local disk accesses as well as to speed up remote accesses. The caches are organized on a block basis. Blocks are currently 4Kb. Each block in the cache is virtually addressed by the file designator and a block location within the file. Using virtual addresses instead of physical disk addresses enable clients to create new blocks in the cache and locate any block without the file i-node being brought from the server. Currently, Sprite does not use read-ahead to speed up sequential read (in contrast to NFS).

A delayed-write approach is used to handle file modification. A dirty block is not written through to the servers cache or the disk until it is ejected from the cache or 30 s have elapsed since the block was last modified. Hence, a block written on a client machine will be written to the servers cache in at most 30 s and will be written to the server's disk after an additional 30 s.

Exact emulation of UNIX semantics is one of Sprite's goals. A hybrid cache validation method is used for this end. Files are associated with a version number. The version number of a file is incremented whenever a file is opened in Write mode. When a client opens a file, it obtains the file's current version number from the server and compares this number to the version number associated with the cached blocks for that file. If the version numbers are different, the client discards all cached blocks for the file and reloads its cache from the server when the blocks are needed. Because of the delayed-write policy, the server does not always have the current file data. Servers handle this situation by keeping track of the last writer for each file. When a client other than the last writer opens the file, the server forces the last writer to write all its dirty blocks back to the server's cache. When a server detects (during an Open operation) that a file is open on two or more workstations and at least one of them is writing the file, it disables client caching for that file (thereby resorting to a remote service mode). All subsequent Reads and Writes go through the server, which serializes the accesses. Caching is disabled on a file basis, and the disablement affects only clients with open files. A substantial degradation of performance occurs when caching is disabled. A noncachable file becomes cachable again when it has been closed on all clients. A file may be cached simultaneously by several active readers.

This approach depends on the fact that the server is notified whenever a file is opened or closed. This prohibits performance optimizations such as name caching in which clients open files without contacting their servers. Essentially, the servers are used as centralized control points for cache consistency. In order to fulfill this function, they must maintain state information about open files.

10.4 Summary

Since Sprite is currently under development its design is evolving. Some definite characteristics of the system are, however, already evident;

- *Extensive use of caching.* Sprite is inspired by the vision of diskless workstations with huge main memories and accordingly relies heavily on caching. The current design is fragile due to the amount of the state data kept in-core by the servers. A server crash results in aborting all processes using files on the server. On the other hand, Sprite demonstrates the big merit of caching in main memory—performance.
- *Sharing semantics.* Sprite sacrifices even performance in order to emulate UNIX semantics. This decision eliminates the possibility and benefits of caching in big chunks.
- *Prefix tables.* There is nothing out of the ordinary in prefix tables. Nevertheless, for LAN-based file systems, prefix tables are a most efficient, dynamic, versatile, and robust mechanism for file lookup. The key advantages are the built-in facility for processing whole prefixes of pathnames (instead of processing component by component) and the supporting broadcast protocol that allows dynamic changes in the tables.

11. ANDREW

Andrew is a distributed computing environment that has been under development since 1983 at Carnegie-Mellon University. The Andrew file system constitutes the underlying information-sharing mechanism among users of the environment. One of the most formidable requirements of Andrew is its scale—the system is targeted to span more than 5000 workstations. Since 1983, Andrew has gone through design,

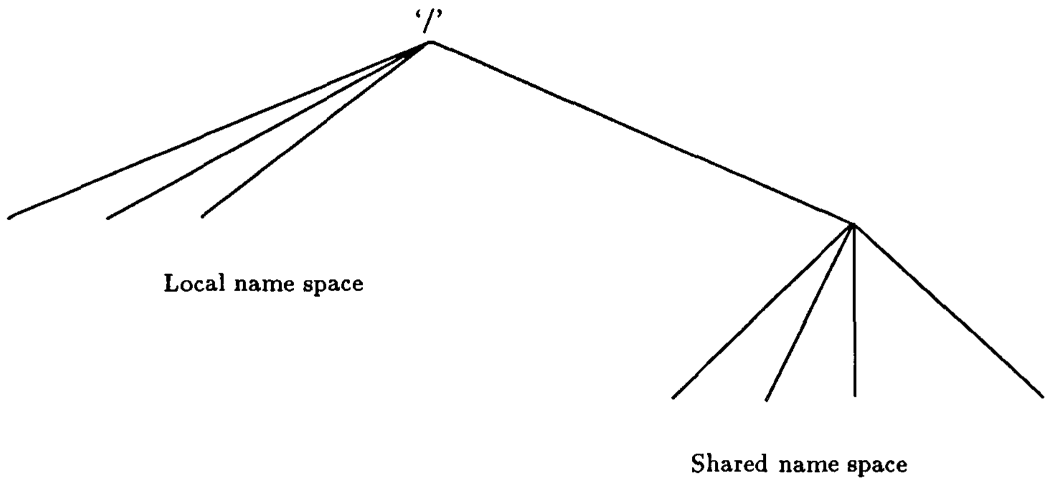

Figure 7. Andrew's name spaces.

prototype implementation, and refinement phases. Our description concentrates on a recent version reported mainly in Howard et al. [1988]. It is interesting to examine how the design evolved from the prototype to the current version. An excellent account of this evolution along with a concise description of the first prototype can be found in Howard et al. [1988].

In early 1987 Andrew encompassed about 400 workstations and 16 servers. Typically, the workstations were Sun's and IBM RTs, with local disks; the file servers were Sun's or Vax's, with much larger disks. Section 11.1 gives a brief overview of the file system and introduces its primary architectural components. Sections 11.2, 11.3, and 11.4 discuss the shared name space structure, the strategy for implementing file operations, and various implementation details, respectively.

11.1 Overview

Andrew distinguishes between client machines (sometimes referred to just as workstations) and dedicated server machines. Servers and clients alike run the UNIX 4.2BSD operating system and are interconnected by an internet of LANs.

Clients are presented with a partitioned space of file names: a *local name space* and a *shared name space*. A collection of dedicated servers, collectively called *Vice*, presents the shared name space to the clients as an identical and location-transparent file hierarchy. The local name space is the root file system of a workstation from which the shared name space descends (Figure 7). Workstations are required to have local disks where they store their local name space, whereas servers collectively are responsible for the storage and management of the shared name space. The local name space is small and distinct from each workstation and contains system programs essential for autonomous operation and better performance, temporary files, and files the workstation owner explicitly wants, for privacy reasons, to store locally. Viewed at a finer granularity, clients and servers are structured in clusters interconnected by a backbone LAN (Figure 8). Each cluster consists of a collection of workstations, a representative of Vice called a *cluster server*, and is connected to the backbone by a router. The decomposition into clusters is primarily to address the problem of scale. For optimal performance, workstations should use the server on their own cluster most of the time, thereby making cross-cluster file references relatively infrequent.

The file system architecture was motivated by consideration of scale, too. The basic heuristic was to off-load work from the servers to the clients, in light of the common experience indicating that server's CPU is the system's bottleneck [Lazowska et al. 1986]. Following this heuristic, the

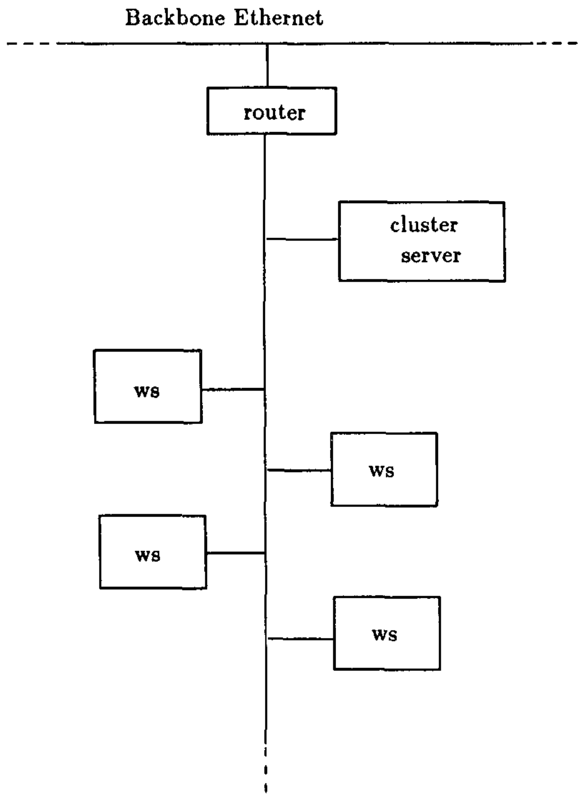

Figure 8. Typical cluster in Andrew.

key mechanism selected for remote file operations is whole file caching. Opening a file causes it to be cached, in its entirety, in the local disk. Reads and writes are directed to the cached copy without involving the servers. Under certain circumstances, the cached copy can be retained for later use.

Entire file caching has many merits, which are described subsequently. This design cannot, however, efficiently accommodate remote access to very large files (i.e., above a few megabytes). Thus, a separate design will have to address the issue of usage of large databases in the Andrew environment. Additional issues in Andrew's design are briefly noted:

- *User mobility.* Users are able to access any file in the shared name space from any workstation. The only noticeable effect of a user accessing files not from the usual workstation would be some initial degraded performance due to the caching of files.
- *Security.* Special consideration was given to security. The Vice interface is considered the boundary of trustworthiness since no user programs are executed on Vice machines. Authentication and secure transmission functions based on the RPC paradigm, are provided as part of communication package. After mutual authentication, a Vice server and a client communicate via encrypted messages. Encryption is performed by hardware devices. Information about users and groups is stored in a protection database that is replicated at each server.
- *Protection.* Andrew provides access lists for protecting directories and the regular UNIX bits for file protection. The access lists mechanism is based on recursive groups structure, similar to the registration database of Grapevine [Birrel et al. 1982].
- *Heterogeneity.* Defining a clear interface to Vice is a key for integration of diverse workstation hardware and operating system. To facilitate heterogeneity, some files in the local **/bin** directory are symbolic links pointing to machine-specific executable files residing in Vice.

11.2 Shared Name Space

Andrew's shared name space is constituted of component units called *volumes*. Andrew's volumes are unusually small component unit. Typically, they are associated with the files of a single user. Few volumes reside within a single disk partition and may grow (up to a quota) and shrink in size. Volumes are joined together by a mechanism similar to the mount mechanism. The granularity difference is significant, since in UNIX only an entire disk partition (containing a file system) can be mounted. Volumes are a key administrative unit and play a vital role in identifying and locating an individual file.

A Vice file or directory is identified by a low-level identifier called *fid.* Each Andrew directory entry maps a pathname component to a fid. A fid has three equal length components: a volume number, a vnode number, and a uniquifier. The vnode number is used as an index into an array containing the i-node of files in a single volume. The uniquifier allows reuse of vnode numbers, thereby keeping certain data structures compact. Fid's are location independent; therefore, file movements from server to server do not invalidate cached directory contents.

Location information is kept on a volume basis in a *volume location database* replicated on each server. A client can identify the location of every volume in the system, querying this database. It is the aggregation of files into volumes that makes it possible to keep the location database at a manageable size.

To balance the available disk space and use of servers, volumes need to be migrated among disk partitions and servers. When a volume is shipped to its new location, its original server is left with temporary forwarding information so the location database need not be updated synchronously. While the volume is being transferred, the original server still may handle updates, which are later shipped to the new server. At some point the volume is briefly disabled to process the recent modifications, then the new volume becomes available again at the new site. The volume movement operation is atomic; if either server crashes the operation is aborted.

Read-only replication at the granularity of an entire volume is supported for system-executable files and seldom-updated files in the upper levels of the Vice name space. The volume location database specifies the server containing the only read-write copy of a volume and a list of read-only replication sites.

11.3 File Operations and Sharing Semantics

The fundamental architectural principle in Andrew is the caching of entire files from servers. Accordingly, a client workstation interacts with Vice servers only during opening and closing of files, and even this is not always necessary. No remote interaction is caused by reading or writing files (in contrast to the remote service method). This key distinction has far-reaching ramifications on performance as well as on semantics of file operations. The operating system on each workstation intercepts file system calls and forwards them to a user-level process on that workstation. This process, called *Venus*, caches files from Vice when they are opened and stores modified copies of files back on the servers from which they came when they are closed. Venus may contact Vice only when a file is opened or closed; reading and writing individual bytes of a file are performed directly on the cached copy and bypass Venus. As a result, writes at some sites are not immediately visible at other sites.

Caching is further exploited for future opens of the cached file. Venus assumes that cached entries (files or directories) are valid unless notified otherwise. Therefore, Venus need not contact Vice on a file open in order to validate the cached copy. The mechanism to support this policy is called *Callback*, and it dramatically reduces the number of cache validation requests received by servers. It works as follows: When a client caches a file or a directory, the server updates its state information recording this caching. We say that the client has a callback on that file. The server notifies the client before allowing a modification to the file by another client. In such a case, we say that the server removes the callback on the file for the former client. A client can use a cached file for open purposes only when the file has a callback. Therefore, if a client closed a file after modifying it, all other clients caching this file lose their callbacks. When these clients open the file later, they have to get the new version from the server.

Reading and writing bytes of a file are done directly by the kernel without Venus intervention on the cached copy. Venus regains control when the file is closed, and if the file has been modified locally, Venus updates the file on the appropriate server. Thus, the only occasions in which Venus contacts Vice servers are on opening files that either are not in the cache or have had their callbacks revoked, and on Close-of-writing sessions.

Basically, Andrew implements session semantics. The only exceptions are file operations other than the primitive Read and Write (such as protection changes at the directory level), which are visible everywhere on the network immediately after the operation completes.

In spite of the callback mechanism, a small amount of cached validation traffic is still present, usually to replace callbacks lost because of machine or network failures. When a workstation is rebooted, Venus considers all cached files and directories suspect and generates a cache validation request for the first use of each such entry.

The callback mechanism forces each server to maintain callback information and each client to maintain validity information. If the amount of callback information maintained by a server is excessive, the server can break callbacks and reclaim some storage by unilaterally notifying clients and revoking the validity of their cached files. There is a potential for inconsistency if the callback state maintained by Venus gets out of sync with the corresponding state maintained by the servers.

Venus also caches contents of directories and symbolic links for pathname translation. Each component in the pathname is fetched, and a callback is established for it if it is not already cached or if the client does not have a callback on it. Lookups are done locally by Venus on the fetched directories using fid's. There is no forwarding of requests from one server to another. At the end of a pathname traversal all the intermediate directories and the target file are in the cache with callbacks on them. Future open calls to this file will involve no network communication at all, unless a callback is broken on a component of the pathname.

The only exception to the caching policy are modifications to directories that are made directly on the server responsible for that directory for reasons of integrity. There are well-defined operations in the Vice interface for such purposes. Venus reflects the changes in its cached copy to avoid refetching the directory.

11.4 Implementation

User processes are interfaced to a UNIX kernel with the usual set of system calls. The kernel is modified slightly to detect references to Vice files in the relevant operations and to forward the requests to the user-level Venus process at the workstation.

Venus carries out pathname translation component by component as described earlier. It has a mapping cache that associates volumes of server locations to avoid server interrogation for an already known volume location. The information in this cache is treated as a hint. If a volume is not present in this cache or if the location information turned out to be wrong, Venus contacts a server, requests the location information, and enters this information into the mapping cache. When a target file is found and cached, a copy is created on the local disk. Venus then returns to the kernel, which opens the cached copy and returns its handle to the user process.

The UNIX file system is used as a low-level storage system for both servers and clients. The client cache is a local directory on the workstation's disk. Within this directory are files whose names are place holders for cache entries. Both Venus and server processes access UNIX files directly by their i-nodes to avoid the expensive pathname translation routine (**namei**). Since the internal i-node interface is not visible to user-level processes (both Venus and server processes are user-level processes), an appropriate set of additional system calls was added.

Venus manages two separate caches—one for status and the other for data. Venus uses a simple least-recently used algorithm to keep each of them bounded in size. When a file is flushed from the cache, Venus notifies the appropriate server to remove the callback for this file. The status cache is kept in virtual memory to allow rapid servicing of system calls that ask for status information (e.g., the UNIX **stat** call). The data cache is resident on the local disk, but the UNIX I/O buffering mechanism does some caching of disk blocks in memory that is transparent to Venus.

A single user-level process on each file server services all file requests from clients. This process uses a LWP package with nonpreemptable scheduling to service many client requests concurrently. The RPC package is integrated with the LWP, thereby allowing the file server to be concurrently making or servicing one RPC per lightweight process. RPC is built on top of a low-level datagram abstraction.

Whole file transfer is implemented as a side effect of RPC call. There is an RPC connection per client, but there is no a priori binding of LWPs to these connections. Instead, a pool of LWPs service client requests on all connections. The use of a single, user-level, server process allows Venus to maintain in its address space caches of data structures needed for its operation. On the other hand, a single server process crash has the disastrous effect of paralyzing this particular server.

A more recent version of Andrew differs slightly from the version described here. Instead of whole-file caching, caching in chunks of 64Kb, is used. Also, the sharing semantics were modified slightly. Updates are still immediately invisible. The updating client can, however, explicitly request that the changes become visible even to remote clients having the file already open.

11.5 Summary

We review the highlights of the Andrew file system:

- *Name space and service model.* Andrew explicitly distinguishes among local and shared name spaces, as well as among clients and dedicated servers. Clients have a small and distinct local name space and can access the shared name space managed by the servers.
- *Scalability.* Andrew is distinguished by its scalability. The strategy adopted to address scale is whole file caching (to local disks) in order to reduce servers load. Servers are not involved in reading and writing operations. The callback mechanism was invented to reduce the number of validity checks. Performing pathname traversals by clients off-loads this burden from servers. The penalty for choosing this strategy and the corresponding design includes maintaining a lot of state data on the servers to support the callback mechanism and specialized sharing semantics.
- *Sharing semantics.* Andrew's semantics are simple and well defined (in contrast to NFS, for instance, where effects of concurrent accesses are time dependent). They are not, however, UNIX semantics. Basically, Andrew's semantics ensure that a file's updates are visible across the network only after the file has been closed.
- *Component units and location mapping.* Andrew's component unit—the volume—is of relatively fine granularity and exhibits some primitive mobility capabilities. Volume location mapping is implemented as a complete and replicated mapping at each server.

Results of a thorough series of performance experimentation with Andrew are presented in Howard et al [1988]. The results confirm the current design predictions. That is, the desired effects on server CPU use, network traffic, and overall time needed to perform remote file operations were obtained, in particular under severe server load. The performance experiments include a benchmark comparison with NFS in which Andrew demonstrated its superiority regarding the recently mentioned criteria, again especially for severe server load.

12. OVERVIEW OF RELATED WORK

This paper focused on several concepts and systems without exhausting the area of DFSs. Consequently, many aspects and systems were omitted. In this section we therefore cite references that complement this paper.

Many studies of typical properties of file and characteristics of file accesses have been done over the years [Ousterhout et al. 1985; Satyanarayanan 1981; Smith 1981]. These empirical results have vast impact on the design of a DFS. Material on another subject that was not covered in this survey, namely security and authentication, can be found in Needham and Schroeder [1978] and Satyanarayanan [1989].

A detailed survey of mainly centralized file servers is found in Svobodova [1984]. The emphasis is on support of atomic transactions, not on location transparency and naming. A tutorial on distributed operating systems is presented in Tanenbaum and Van Renesse [1985]. There, a distributed operating system is defined and issues like communication primitives and protection are discussed. These two surveys include an extensive bibliography to a variety of distributed systems.

Next, we give a concise overview of a few noteworthy DFSs that were not surveyed in this paper.

- *Roe* [Ellis and Floyd 1983; Floyd 1989]. Roe presents a file as an abstraction hiding both replication and location details. Files are migrated to achieve balancing of systemwide disk storage allocation and also as a remote access method. Consistency of replicated files is obtained by a weighted voting algorithm [Gifford 1979].
- *Eden* [Almes et al, 1983; Black 1985; Jessop et al. 1982]. A radically different approach is adopted for the experimental Eden file system from the University of Washington. The system is based on the object-oriented and capability-based approaches [Levy 1984]. A file is a dynamic object that can be viewed as an instance of an abstract data type. It includes processes that satisfy requests oriented to the file (i.e., there is no separation of passive data files and active server processes). A kernel-supported storage system provides primitives for checkpointing the representation of an object to secondary storage, copying it, or moving it from machine to machine. Eden files can be replicated, can be migrated, are named in a location-independent manner, and can support atomic transactions. More material on migratory objects can be found in the context of the Emerald project, conducted in the same university [Jul et al. 1987].
- *Stork* [Paris and Tichy 1983]. Stork is an experimental file system designed to evaluate the feasibility of file migration as a remote access method. Locating a migratory file is based on a primitive mechanism of associating the file's owner with a list of possible machines where the files can be located. It emphasizes that file access patterns must exhibit locality to make file migration an attractive remote access method.
- *Ibis* [Tichy and Ruan 1984]. Ibis is the successor of Stork. It is a user-level extension of UNIX. Remote file names are prefixed with their host name and can appear in system calls as well as in shell commands. The replication scheme was described in Section 5.3. Low-level, structure, but location-dependent names are used. One of the parts of the structured name designates the machine that currently stores the file. These names render file migration a very expensive operation, since all directories containing the name of the migrated file must be updated.
- *Apollo Domain* [Leach et al., 1982, 1985]. The Domain system is a commercial product featuring a collection of powerful workstations connected by a high-speed LAN. An object-oriented approach is taken. Files are objects, and as such they may be of different types. Accordingly, it is possible to construct file operations that are customized for a particular file type. All the objects in the system are named by a networkwide, unique, low-level, location-independent name, called a UID. Objects are organized in hierarchical, UNIX-like, directories that associate textual names with UIDs. No global state information is kept on object locations. Instead, an interesting location algorithm, based on heuristics (hints) for guessing the object's location, is used. For instance, one helpful heuristic is to assume that objects created at the same machine are likely to be located together. A unique feature of Domain is the way objects are accessed once located. Objects are mapped directly onto clients' address spaces and accessed via virtual memory paging. In terms of remote access methods, this amounts to caching in the granularity of pages. Write-through policy

is used for modification, and client-initiated approach is used for validation of cached data.

13. CONCLUSIONS

In this paper we presented the basic concepts underlying the design of a distributed file system and surveyed five of the most prominent systems. A comparison of the systems is presented in Table 1. A crucial observation, based on the assessment of contemporary DFSs, is that the design of a DFS must depart from approaches developed for conventional file systems. Basing a DFS on emulation of a conventional file system might be a transparency goal, but it certainly should not be an implementation strategy. Extending mechanisms developed for conventional file systems over a network is a strategy that disregards the unique characteristics of a DFS.

Supporting this claim is the observation that a loose notion of sharing semantics is more appropriate for a DFS than conventional UNIX semantics. Restrictive semantics incur a complex design and intolerable overhead. A provision to facilitate restrictive semantics for database applications may be offered as an option. Consequently, UNIX compatibility should be sacrificed for the sake of a good DFS design. In this respect, the approach used in Andrew to the semantics of sharing prove superior to those used in Locus and NFS.

Another area in which a fresh approach is essential is the server process architecture. There is a wide consensus that some form of LWPs is more suitable than traditional processes for efficiently handling high loads of service requests.

It is difficult to present concrete guidelines in the context of fault tolerance and scalability, mainly because there is not enough experience in these areas. It is clear, however, that distribution of control and data as presented in this paper is a key concept. User convenience calls for hiding the distributed nature of such a system. As we pointed out in Section 2, the additional flexibility gained by mobile files is the next step in the spirit of distribution and transparency. Based on the Andrew experience, off-loading work from servers to clients and structuring a system as a collection of clusters are two sound scalability strategies. Clusters should be as autonomous as possible and should serve as a modular building block for an expandable system. A challenging aspect of scale that might be of interest for future designs is the extension of the DFS paradigm over WANs. Such an extended DFS would be characterized by larger latencies and higher failure probabilities.

A factor that is certain to be prominent in the design of future DFSs is the available technology. It is important to follow technological trends and exploit their potential. Some imminent possibilities are as follows:

- *Large main memories.* As main memories become larger and less expensive, main-memory caching (as exemplified in Sprite) becomes more attractive. The rewards in terms of performance can be exceptional.
- *Optical disks.* Optical storage technology has an impact on file systems in general and hence on DFSs in particular, too. Write-once optical disks are already available [Fujitani 1984]. Their key features are very large density, slow access time, high reliability, and nonerasable writing. This medium is bound to become on-line tertiary storage and replace tape devices. Rewritable optical disks are becoming available and might replace magnetic disks altogether.
- *Optical fiber networks.* A change in the entire approach to the remote access problem can be justified by the existence of these remarkably fast communication networks. The concept of local disk is faster may be rendered obsolete.
- *Nonvolatile RAMs.* Battery-backed memories can survive power outage, thereby enhancing the reliability of main-memories caches. A large and reliable memory can cause a revolution in storage techniques. Still, it is questionable whether this technology is sufficient to make main memories as reliable as disks because of the unpredictable consequences of an operating system crash

Table 1. Comparison of Surveyed Systems

	UNIX United	Locus
Background	Interconnecting a set of loosely coupled UNIX systems without modifying the kernel.	A highly reliable distributed operating system providing multiple dimensions of transparency and is UNIX compatible.
Naming scheme	Single pseudo-UNIX tree. Noticeable machine boundaries. All pathnames are relative (by the '. .' syntax). Independence of component systems. Recursive structuring.	Single UNIX tree, hiding both replication and location.
Component unit	Entire UNIX hierarchy.	A logical filegroup (UNIX file system).
User mobility	Not supported.	Supported.
Client-server	Each machine can be both.	A triple: US, SS, CSS. Every file group has a CSS that selects SS and synchronizes accesses. Once a file is open, direct US-SS protocol.
Remote-access method	Emulation of conventional UNIX across the network.	Once a file is open, accesses are served by caching.
Caching	Emulation of UNIX buffering.	Block caching similar to UNIX buffering. A token scheme for cache consistency. Closing a file commits it on the server.
Sharing semantics		Complete UNIX semantics, including sharing of file offset.
Pathname traversal	The pathname translation request is forwarded from machine to machine.	US reads each directory and performs lookup itself. Given a file group number, the CSS is found replicated on all machines in the logical mount table. The CSS picks SS.
Reconfiguration, file mobility	Impossible to move a file without changing its name. No dynamic reconfiguration.	Because of replication, servers can be taken off-line or fail without disturbance. Directory hierarchy can be changed by mounting/unmounting.
Availability		Availability of a file means the CSS and SS are available. Each component in the file's pathname must be available for the file to be opened. The primary copy must be available for a Write.

Table 1—*Continued*

NFS	Sprite	Andrew
A network service so that independent workstations would be able to share remote files transparently.	Designed for an environment consisting of diskless workstations with huge main memories interconnected by a LAN.	Designed as the sharing mechanism of a large-scale system for a university campus.
Each machine has its own view of the global name space.	Single UNIX tree; hiding location.	Private name spaces and one UNIX tree for the shared name space. The shared tree descends from each local name space.
A directory within an exported file system can be remotely mounted.	A domain (UNIX file system).	A volume (typically, all files of a single user).
Potential for support exists; demands certain configuration.	Supported.	Fully supported.
Every machine can be both. Direct client-server relationship is enforced.	Typically, clients are diskless and servers are machines with disks.	Clustering: Dedicated servers per cluster.
Remote service mixed with block caching for service.	Block caching in main memory. In case of concurrent writes, switch to Remote Service.	Whole file caching in local disks.
Block caching similar to UNIX buffering. Client checks validity of cached data on each open. Delayed-write policy.	Block caching similar to UNIX buffering. Delayed-write policy. Client checks validity of cached data on each open. Server disables caching when a file is opened in conflicting modes.	Read and Write are served directly by the cache without server involvement. Write-on-close policy. Server-initiated approach for cache validation (callback), hence no need to check on each open.
Not UNIX semantics. Timing-dependent semantics.	UNIX semantics.	Session semantics.
Lookups are done remotely for each pathname component, but all are initiated from the client. A lookup cache for speedup.	Prefix tables mechanism. Inside a domain, lookup is done by server.	Client caches each directory and performs lookup itself. Given a volume number, the server is found in a volume location database replicated on each server. Parts of this database are cached on each machine.
Mount/unmount can be done dynamically by superuser for each machine.	Broadcast protocol supports dynamic reassignment of domains to servers.	Volume migration is supported.
In case of cascading mount, each server along the mount chain has to be available for a file to be available.	If a server of a file is available, the file is available regardless of the state of other servers (along the pathname).	A client has to have a connection to a server, and each pathname component must be available.

Table 1—*Continued*

	UNIX United	Locus
Other fault tolerance issues		A file is committed on close. The primary copy is always up to date. Other replicas may have older (but not partially modified) versions.
Scalability issues	Recursive structuring.	Replicated mount table on each site and CSS for a file group are major problems.
Implementation strategy, architecture	UNIX kernel kept intact. Connection layer intercepts remote calls. User-level daemons forward and service remote operations. A spawner process creates a server process per user that accesses files using file descriptors.	Extensive UNIX kernel modification. Kernel is pushed into the network. Some kernel LWP for remote services. Structured, low-level, location-independent file identifiers are used.
Networking	Suitable for arbitrary internetwork topology.	LAN
Communication protocol	RPC	Specialized low-level protocols for each operation.
Special features		Replication (primary copy). Atomic update by shadow paging.
Main advantage	Original UNIX kernel. Internetworking capabilities.	Performance, because of kernel implementation. Fault tolerance, due to replication, atomic update, and other features. UNIX compatibility.
Main disadvantage	Not fully transparent naming.	Complicated design and large kernel. Unscalable features. Complex recovery due to maintained state.

[Ousterhout 1989]. Other problems of relatively slow access time and limited size still plague this technology.

ACKNOWLEDGMENTS

This work was partially supported by NSF grant IRI 8805215 and by the Texas Advanced Research Program under grant No. 4355. We would like to thank the anonymous reviewers and the editor-in-chief, Salvatore March, for their constructive comments that led us to improve the clarity of this paper.

REFERENCES

ALMES, G. T., BLACK, A. P., LAZOWSKA, E. D., AND NOE, J. D. 1983. The Eden system: A technical review. *IEEE Trans. Softw. Eng. 11*, 1 (Jan.), 43–59.

BARAK, A., AND KORNATZKY, Y. 1987. *Design Principles of Operating Systems for Large Scale Multicomputers*. IBM Research Division, T. J. Watson Research Center, Yorktown Heights, New York. RC 13220 (#59114).

BARAK, A., AND LITMAN, A. 1985. MOS: A multicomputer distributed operating system. *Softw. Prac. Exper. 15*, 8 (Aug.), 725–737.

Table 1—*Continued*

NFS	Sprite	Andrew
Complete stateless service. Idempotent operations.	No guarantees because of the delayed write policy. Stateful service.	Not dealt with fully yet. Stateful service.
Not intended for very large-scale systems.	Because broadcast is relied upon and of server involvement in operations there might be a problem.	Reducing server load and clustering are the main strategy. Replicated location database might be a problem.
Three layers: UNIX system call interface, VFS interface to separate file system implementation from operations, and NFS layer. Independent specifications for mount and NFS protocols. The Current implementation is kernel based.	New kernel based on multithreading, intended for multiprocessor workstation.	Augmenting UNIX kernel with user-level processes: Venus at each client, and a single server process on each server using nonpreemptable LWPs. Structured, low-level, location-independent file identifiers are used.
LAN	LAN	Cluster structure, with a router per cluster. All communication is based on high bandwidth LAN technology.
RPC and XDR on top of UDP/IP (unreliable datagram).	RPC on top of special-purpose network protocol.	RPC on top of datagram protocol. Whole file transfer as a side effect.
Stateless service.	Regular files used as swapping area. Interaction between file system and virtual memory system.	Authentication and encryption built into the communication protocol. Access list mechanism for protection. Limited read-only replication.
Fault tolerance, because of stateless protocol. Implementation-independent protocols, ideal for heterogeneous environment.	Performance due to main memory caching.	Ability to scale up gracefully. Clear and simple consistency semantics.
Unclear semantics. Performance improvements obscure clean design.	Questionable scalability. Not much in terms of fault tolerance.	Fault tolerance issues due to maintained state.

BARAK, A., MALKI, D., AND WHEELER, R. 1986. AFS, BFS, CFS ... or Distributed File Systems for UNIX. In *European UNIX Users Group Conference Proceedings* (Sept. 22–24, Manchester, U.K.). EUUG, pp. 461–472.

BARAK, A., AND PARADISE, O. G. 1986. MOS: Scaling up UNIX. In *Proceedings of USENIX 1986 Summer Conference.* USENIX Association, Berkeley, California, pp. 414–418.

BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency Control and Recovery an Database Systems.* Addison-Wesley, Reading, Mass.

BIRREL, A. D., LEVIN, R., NEEDHAM, R. M., AND SCHROEDER, M. D. 1982. Grapevine: An exercise in distributed computing. *Commun. ACM 25,* 4 (Apr.), 260–274.

BIRREL, A. D., AND NELSON, B. J. 1984. Implementing remote procedure calls. *ACM Trans. Comput Syst. 2,* 1 (Feb.), 39–59.

BLACK, A. P. 1985. Supporting distributed applications: Experience with Eden. In *Proceedings of the 10th Symposium on Operating Systems Principles* (Orcas, Island, Wash., Dec. 1–4). ACM, New York, pp. 181–193.

BROWNBRIDGE, D. R., MARSHALL, L. F., AND RANDELL, B. 1982. The Newcastle connection or UNIXes of the world unite! *Softw. Prac. Exper. 12,* 12 (Dec.), 1147–1162.

CHERITON, D. R., AND ZWAENEPOEL, W. 1983. The distributed V kernel and its performance for diskless workstations. *In Proceedings of the 9th Symposium on Operating Systems Principles* (Bretton Woods, N.H., Oct.). ACM, New York, pp. 128–140.

DAVIDSON, S. B., GARCIA-MOLINA, H., AND SKEEN, D. 1985. Consistency in partitioned networks. *ACM Comput. Surv. 17,* 3 (Sept.), 341–370.

DION, J. 1980. The Cambridge file server. *ACM SIGOPS, Oper. Syst. Rev. 14,* 4 (Oct.), 26–35.

DOUGLIS, F., AND OUSTERHOUT, J. K. 1989. Beating the I/O bottleneck. *ACM SIGOPS, Oper. Syst. Rev. 23,* 1 (Jan.), 11–28.

ELLIS, C. S., AND FLOYD, R. A. 1983. The ROE file system. In *Proceedings of the 3rd Symposium on Reliability in Distributed Software and Database Systems* (Clearwater Beach, Florida, October 17-19). IEEE, New York.

FLOYD, R. 1989. Transparency in distributed file systems. Tech. Rep. 272, Department of Computer Science, University of Rochester.

FUJITANI, L. 1984. Laser optical disks: The coming revolution in on-line storage. *Commun. ACM 27,* 6 (June).

GIFFORD, D. 1979. Weighted voting for replicated data. In *Proceedings of the 7th Symposium on Operating Systems Principles* (Pacific Grove, Calif. Dec. 10-12). ACM, New York, pp. 150-159.

HILL M., EGGERS, S., LARUS, J., TAYLOR, G., ADAMS, G., BOSE, B. K., GIBSON, G., HANSEN, P., KELLER, J., KONG, S., LEE, C., LEE, D., PENDLETON, J., RITCHIE, S., WOOD, D., ZORN, B., HILFINGER, P., HODGES, D., KATZ, R., OUSTERHOUT, J., AND PATTERSON, D. 1986. Design decisions in SPUR. *IEEE Comput. 19,* 11 (Nov.), 8–22.

HOARE, C. A. R. 1974. Monitors: An operating system structuring concept. *Commun. ACM 17,* 10 (Oct.), 549–557.

HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., AND SIDEBOTHAM, R. N. 1988. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst. 6,* 1 (Feb.), 55–81.

JESSOP, W. H., JACOBSON, D. M., NOE, J. D., BAER, J. L., AND PU, C. 1982. The Eden transaction based file system. In *Proceedings of the 2nd Symposium on Reliability in Distributed Software and Databases Systems* (July). IEEE, New York, pp. 163–169.

JUL, E., LEVY, H. M., HUCHINSON, N., AND BLACK, A. 1987. Fine grain mobility in the Emerald system (extended abstract). In *Proceedings of the 11th Symposium on Operating Systems Principles,* (Austin, Texas, November). ACM, New York.

KEPECS, J. 1985. Light weight processes for UNIX implementation and applications. In *Proceedings of Usenix 1985 Summer Conference.*

LAMPSON, B. W. 1981. Atomic transactions. In *Distributed Systems–Architecture and Implementation: An Advanced Course,* G. Goos and J. Hartmanis, Eds., Springer-Verlag, Berlin, Chap. 11, pp. 246-265.

LAMPSON, B. W. 1983. Hints for computer system designers. In *Proceedings of the 9th Symposium on Operating Systems Principles* (Bretton Woods, N.H., Oct.). ACM, New York, pp. 33–48.

LAZOWSKA, E. D., LEVY, H. M., AND ALMES, G. T. 1981. The architecture of the Eden system. In *Proceedings of the 8th Symposium on Operating Systems Principles* (Asilomar, Calif., Dec.). ACM, New York, pp. 148–159.

LAZOWSKA, E. D., ZAHORJAN, J., CHERITON, D., AND ZWAENEPOEL, W. 1986. File access performance of diskless workstations. *ACM Trans. Comput. Syst. 4,* 3 (Aug.), 238–268.

LEACH, P. J., STUMP, B. L., HAMILTON, J. A., AND LEVINE, P. H. 1982. UIDs as internal names in a distributed file system. In *Proceedings of the 1st Symposium on Principles of Distributed Computing* (Ottawa, Ontario, Canada, Aug. 18–20). ACM, New York, pp. 34–41.

LEACH, P. J., LEVINE, P. H., HAMILTON, J. A., STUMP, B. L. 1985. The file system of an integrated local network. In *Proceedings of the ACM Computer Science Conference* (New Orleans, Mar.). ACM, New York.

LEVY, H. M. 1984. *Capability Based Computer Systems.* Digital Press, Bedford, Mass.

MCKUSICK, M. K., JOY, W. N., LEFFER, S. J., FABRY, R. S. 1984. A fast file system for UNIX. *ACM Trans. Comput. Syst. 2,* 3 (Aug.), 181–197.

MITCHELL, J. G. 1982. File servers for local area networks. Lecture Notes, Course on Local Area Networks, University of Kent, Canterbury, England, pp. 83–114.

MORRIS, J. H., SATYANARAYANAN, M., CONNER, M. H., HOWARD, J. H., ROSENTHAL, D. S. H., AND SMITH, F. D. 1986. Andrew: A distributed personal computing environment. *Commun. ACM 29,* 3 (Mar.), 184–201.

NEEDHAM, R. M. HERBERT, A. J. 1982. *The Cambridge Distributed Computing System,* Addison Wesley, Reading, Mass.

NEEDHAM, R. M., AND SCHROEDER, M. D. 1978. Using encryption for authentication in large networks of computers. *Commun. ACM 21,* 12 (Dec. 1978).

NELSON, M., WELCH, B., AND OUSTERHOUT, J. K. 1988. Caching in the Sprite network file system. *ACM Trans. Comput. Syst. 6,* 1 (Feb.).

OUSTERHOUT, J. K., DA COSTA, H., HARISSON, D., KUNZE, J. A., KUPFLER, M., AND THOMPSON, J. G. 1985. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the 10th Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1–4). ACM, New York, pp. 15–24.

OUSTERHOUT, J. K., CHERENSON, A. R., DOUGLIS, F., NELSON, M. N., AND WELCH, B. B. 1988.

The Sprite network operating system. *IEEE Comput. 21,* 2 (Feb.), 23–36.

PARIS, J. F., AND TICHY, W. F. 1983. Stork: An experimental migrating file system for computer networks. In *Proceedings IEEE INFCOM.* IEEE, New York, pp. 168–175.

POPEK, G., AND WALKER, B. Eds. 1985. *The LOCUS Distributed System Architecture.* MIT Press, Cambridge Mass.

POSTEL, J. 1980. User datagram protocol. RFC-768. Network Information Center, SRI.

QUARTERMAN, J. S., SILBERSCHATZ, A., AND PETERSON, J. L. 1985. 4.2 and 4.3 BSD as examples of the UNIX system. *ACM Comput. Surv. 17,* 4 (Dec.).

RANDELL, B. 1983. Recursively structured distributed computing systems. In *Proceedings of the 3rd Symposium on Reliability in Distributed Software and Database Systems* (Clearwater Beach, Fla., Oct. 17–19). IEEE, New York, pp. 3–11.

RITCHIE, D. M., AND THOMPSON, K. 1974. The UNIX time sharing system. *Commun. ACM 19,* 7 (Jul.), 365–375.

SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYONE, B. 1985. Design and implementation of the Sun network file system. In *Proceedings of Usenix 1985 Summer Conference* (Jun.), pp. 119–130.

SATYANARAYANAN, M. 1981. A Study of file sizes and functional lifetimes. In *Proceedings of the 8th Symposium on Operating Systems Principles* (Asilomar, Calif., Dec.). ACM, New York.

SATYANARAYANAN, M. 1989. Integrating security in a large distributed system. *ACM Trans. Comput. Syst. 7,* 3 (Aug.), 247–280.

SATYANARAYANAN, M., HOWARD, J. H., NICHOLS, D. A., SIDEBOTHAM, R. N., SPECTOR, A. Z., AND WEST, M. J. 1985. ITC distributed file system: Principles and design. In *Proceedings of the 10th Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1–4). ACM, New York, pp. 35–50.

SCHROEDER, M. D., BIRREL, A. D., AND NEEDHAM, R. M. 1984. Experience with grapevine: The growth of a distributed system. *ACM Trans. Comput. Syst. 2,* 1 (Feb.), 3–23.

SCHROEDER, M. D., GIFFORD, D. K., AND NEEDHAM, R. M. 1985. A caching file system for a programmer's workstation. *Proceedings of the 10th Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1–4), ACM, New York, pp. 25–32.

SHELTZER, A. B., AND POPEK, G. J. 1986. Internet Locus: Extending transparency to an Internet environment. *IEEE Trans. Softw. Eng. SE-12,* 11 (Nov.).

SMITH, A. J. 1981. Analysis of long term file reference patterns for application to file migration algorithms. *IEEE Trans. Softw. Eng. 7,* 4 (Jul.).

SMITH, A. J. 1982. Cache memories. *ACM Comput. Surv. 14,* 3 (Sept.), 473–530.

SUN MICROSYSTEMS, INC. 1988. Network Programming, Sun Microsystems, Part Number: 800-1779-10, Revision A, of 9 May 1988.

SVOBODOVA, L. 1984. File servers for network based distributed systems. *ACM Comput. Surv. 16,* 4 (Dec.), 353–398.

TANENBAUM, A. S., AND VAN RENESSE, R. 1985. Distributed operating systems. *ACM Comput. Surv. 17,* 4 (Dec.) 419–470.

TERRY, D. B. 1987. Caching hints in distributed systems. *IEEE Trans. Softw. Eng. SE-13,* 1 (Jan.), 48–54.

TEVANIAN, A., RASHID, R., GOLUB, D., BLACK, D., COOPER, E., AND YOUNG, M. 1987. Mach threads and the UNIX kernel: The battle for control. In *Proceedings of USENIX 1987 Summer Conference.* USENIX Association, Berkeley, California.

TICHY, W. F., AND RUAN, Z. 1984. Towards a distributed file system. In *Proceedings of Usenix 1984 Summer Conference* (Salt Lake City, Utah), pp. 87–97.

WALKER, B., POPEK, J., ENGLISH, R., KLINE, C., THIEL, G. 1983. The LOCUS distributed operating system. *ACM SIGOPS, Oper. Syst. Rev. 17,* 5 (Oct.), 49–70.

WEINSTEIN, M. J., PAGE, T. W. JR., B. K. LIVEZEY, AND G. J. POPEK. 1985. Transactions and synchronization in a distributed operating system. In *Proceedings of the 10th Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1–4). ACM, New York.

WELCH, B. 1986. The Sprite remote procedure call system. Tech. Rep. UCB/CSD 86/302, Computer Science Division (EECS), University of California, Berkeley.

WELCH, B., AND OUSTERHOUT, J. K. 1986. Prefix tables: A Simple mechanism for locating files in a distributed system. In *Proceedings of the 6th Conference on Distributed Computing Systems* (Cambridge, Mass., May), IEEE, New York, pp. 184–189.

ZIMMERMANN, H. 1980. OSI reference model: The ISO model of architecture for open system interconnection. *IEEE Trans. Commun. COM-28* (Apr.), 425–432.

BIBLIOGRAPHY

BACH, M. J., LUPPI, M. W., MELAMED, A. S., AND YUEH, K. 1987. A remore file cache for RFS. *Summer Usenix Conference Proceedings.* Phoenix, Ariz.

BIRREL, A. D., AND NEEDHAM, R. M. 1980. A universal file server. *IEEE Trans. Softw. Eng. SE-6,* 5 (Sept.), 450–453.

BROWN, M. R., KOLLING, K. N., AND TAFT, E. A. 1985. The Alpine file system. *ACM Trans. Comput. Syst. 3,* 4 (Nov.).

CABRERA, L. F., AND WYLLIE, J. 1988. QuickSilver distributed file services: An architecture for horizontal growth. In *Proceedings of the 2nd IEEE Conference on Computer Workstations* (Santa Clara, Calif.). IEEE, New York.

CALLAGHAN, B., AND LYON, T. 1989. The Automounter. In *Winter USENIX Conference Proceedings* (San Diego). USENIX Association, Berkeley, California.

CHARLOCK, H. 1987. RFS in SunOS. In *Summer USENIX Conference Proceedings* (Phoenix, Ariz.). USENIX Association, Berkeley, California.

DAVCEV, C., AND BURKHARD, W. A. 1985. Consistency and recovery control for replicated files. In *Proceedings of the 10th Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1-4). ACM, New York.

FINLAGSON, R. S., AND CHERITON, D. R. 1987. Log files: An extended file service exploiting write-once storage. In *Proceedings of the 11th Symposium on Operating Systems Principles* (Austin, Tex., Nov.). ACM, New York.

FRIDIRICH, M., AND OLDER, W. 1981. The Felix file server. In *Proceedings of the 8th Symposium on Operating Systems Principles* (Asilomar, Calif., Dec.). ACM, New York.

GAIT, J. 1988. The optical file cabinet: A random-access file system for write-once optical disks. *IEEE Comput. 21*, 6 (June).

KLEIMAN, S. R. 1986. Vnodes: An architecture for multiple file system types in Sun *UNIX*. In *Summer USENIX Conference Proceedings* (Atlanta, Ga.). USENIX Association, Berkeley, California.

MITCHELL, J. G., AND DION, J. A. 1982. Comparison of two network-based file servers. *Commun. ACM 25,* 4 (Apr.).

MULLENDER, S. J., AND TANENBAUM, A. S. 1985. A distributed file service based on optimistic concurrency control. In *Proceedings of the 10th Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1-4). ACM, New York.

NEEDHAM, R. M., HERBERT, A. J., AND MITCHELL, J. G. 1983. How to connect stable memory to a computer. *Operat. Syst. Rev. 17,* 1 (Jan.).

PINKERTON, C. B., LAZOWSKA, E. D., NOTKIN, D., AND ZAHORJAN, J. A. 1988. Heterogeneous remote file system. Tech. Rep. 88-08-08, Dept. of Computer Science, Univ. of Washington.

RIFKIN, FORBES, A. P., HAMILTON, R. L., SABREO, M., SHAH, S., AND YUEH, K. 1986. RFS architectural overview. In *Summer USENIX Conference Proceedings* (Atlanta, Ga.). USENIX Association, Berkeley, California.

ROSEN, M. B., WILDE, M. J., FRASER-CAMPBELL, B. 1986. NFS portability. In *Summer Usenix Conference Proceedings* (Atlanta, Ga.).

SATYANARAYANAN, M. 1988. On the influence of scale in a distributed system. In *Proceedings of the 10th International Conference on Software Engineering*, (Singapore, April).

SIDEBOTHAM, R. N. 1986. Volumes: The Andrew file system data structuring primitive. In *European Unix Users Group Conference Proceedings* (Aug.). EUUG.

SPECTOR, A. Z., AND KAZAR, M. L. Wide area file services and the AFS experimental system. *UNIX Rev. 7,* 3 (Mar.).

STURGIS, H. E., MITCHELL, J. G., AND ISRAEL, J. 1980. Issues in the design and use of a distributed file system. *Operat. Syst. Rev. 14,* 3 (Jul.), 55–69.

SVOBODOVA, L. 1986. A reliable object-oriented data repository for a distributed computer system. In *Proceedings of the 8th Symposium on Operating Systems Principles* (Asilomar, Calif., Dec.). ACM, New York.

WALSH, D., LYON, B., SAGER, G., CHANG, J. M., GOLDBERG, D., KLEIMAN, S., LYON, T., SANDBERG, R., AND WEISS, P. 1985. Overview of the Sun network filesystem. In *Winter USENIX Conference Proceedings* (Dallas, Tex.). USENIX Association, Berkeley, California.

WUPIT, A. 1983. *Comparison of Unix Network Systems.* ACM, New York.

Received March 1989; final revision accepted December 1989.