DISTRIBUTED FILE SYSTEMS: CONCEPTS AND EXAMPLES

Eliezer Levy and Abraham Silberschatz

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

TR-89-04 March 1989

Distributed File Systems: Concepts and Examples

ELIEZER LEVY and ABRAHAM SILBERSCHATZ

Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712

Distributed File Systems are essential for sharing of data and storage space in a distributed system. A viewpoint that emphasizes the dispersed structure and decentralization of both data and control in the design of such systems is established. The concepts of location transparency, fault tolerance and scalability are defined and discussed in the context of Distributed File Systems. It is claimed that the principle of distributed operation is fundamental for a fault tolerant and scalable Distributed file System. Alternatives for the semantics of sharing and methods for providing access to remote files are also presented.

A survey of current systems, namely Unix United, Locus, Sprite, Sun's Network File System, and ITC's Andrew, illustrates the discussed concepts and demonstrates various implementations and design alternatives. Based on the assessment of these systems, a point is made that a departure from the approach of extending centralized file systems over the network is necessary to accomplish sound Distributed File System design.

INTRODUCTION

The need to share resources in a computer system arises either due to economical reasons, or due to the nature of some applications. In particular, it is necessary to facilitate sharing of storage space and data. This paper investigates Distributed File Systems (DFSs), viewed as a means for sharing space and data.

A distributed system is a collection of loosely coupled machines interconnected by communication network. We use the term *machine* to denote either a mainframe or a workstation. Unless specified otherwise, the *network* is a Local Area Network (LAN). From a point of view of a specific machine in a distributed system, the rest of the machines and their respective resources are *remote*, whereas its own resources are referred to as *local*.

In order to explain the structure of a DFS we need to define the terms service, server and client[1]. A *service* is a software entity running on one or more machines and providing a particular type of service to a priori unknown clients. A *server* is the service software running on a single machine. A *client* is a process that can invoke and control the service using a set of operations that form the *client interface*. Sometimes, a lower level interface is defined for the actual cross-machine interaction. When the need arises, we refer to this interface as the *inter-machine* interface. Clients implement interfaces more suitable for higher level applications or direct access by human users.

The task of a file system is to provide 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. A primary hardware component that a file server controls is a secondary storage device (i.e., a magnetic disk), where files are stored, and from where they are retrieved according to the clients requests. We often say that a server, or a machine store a file, and we mean that the file resides on the attached device. We refer to the file service system offered by a uni-processor, time-sharing operating system (e.g., Unix 4.2 BSD) as a *conventional* file system.

A DFS is a file service system, whose clients, servers and storage devices are dispersed among the ma-

---

[1] Part of these definitions follow [Mitchell 82].

Contents

INTRODUCTION
1. TRENDS AND TERMINOLOGY
2. NAMING AND TRANSPARENCY
   2.1 Naming Structures
   2.2 Naming Schemes
   2.3 Implementation Techniques
3. CONSISTENCY SEMANTICS
4. REMOTE ACCESS METHODS
   4.1 Designing a Caching Scheme
   4.2 Cache Consistency
   4.3 A comparison of Caching and Remote Service
5. FAULT TOLERANCE ISSUES
   5.1 Stateful versus Stateless Service
   5.2 Improving Availability
   5.3 File Replication
6. SCALABILITY ISSUES
   6.1 Guidelines by Negative Examples
7. UNIX UNITED
8. LOCUS
9. SPRITE
10. NFS
11. ANDREW
12. RELATED WORK
13. CONCLUSIONS
ACKNOWLEDGMENTS
REFERENCES

---

chines 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.

Fundamental to our view is the principle of distributed operation of the system. This principle 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 (invisible) to users. We confine ourselves to discussing DFS designs mainly 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 current systems.

The paper is logically 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-off's regarding the design of a DFS are pointed out. The second part surveys five DFSs: Unix United [Brownbridge et al. 1982; Randell 1983], Locus [Walker et al. 1983; Popek and Walker 1985], Sprite [Nelson et al. 1986; Ousterhout et al. 1988; Nelson et al. 1988], Sun's Network File System (NFS) [Sandberg et al. 1985; Sun 1988], and Andrew [Satyanarayanan et al. 1985; Morris et al. 1986; Howard et al. 1988]. These systems exemplify the concepts and observations mentioned in the first part and demonstrate various implementations. Often, an observation is made in the first part with direct mentioning of the system that demonstrates it.

Section 1 establishes useful terminology and presents the concepts of transparency, fault-tolerance, and scalability. Section 2 discusses the concept of transparency in greater detail and how it is expressed in naming

schemes. Section 3 introduces consistency notions that are important for the semantics of sharing files. The methods of Caching and Remote Service are contrasted in Section 4. Section 5 and 6 discuss issues related to fault-tolerance and scalability respectively. There is no intention to exhaust these two broad issues. The intention is to point out some observations based on the designs of the surveyed systems.

Sections 7 to 11 are descriptions of each of the five systems mentioned above. Occasionally, distinctive features of a system that are not related to the points presented in the first part are mentioned too, in that system's description. Each description is followed by a short summary that emphasizes the prominent features of the corresponding system. A compact table, Table 1, that compares the five systems concludes this survey. Many important aspects of DFSs and many systems are omitted from this paper. Section 12 reviews related work that was not emphasized in our discussion. Finally, Section 13 sums up the paper with conclusions.

1 TRENDS AND TERMINOLOGY

It has been the challenge of many designs to make the multiplicity of servers and storage devices *transparent* to clients. Ideally, a DFS should look to its clients like a conventional, centralized file system. The client interface of transparent DFS should not distinguish between local and remote files. That is, clients should be able to access remote files as if they were local, and 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*. It would be convenient to allow users to log in any machine in the system, and not to force them to use a specific machine. A transparent DFS facilitates user mobility by bringing over the user's environment (e.g., home directory) to wherever he, or she logs in.

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

The capabilities of a system to adapt to increased service load are termed *scalability*. Systems have bounded resources and can become completely saturated under increased load. Regarding a file system, saturation occurs either when a server's CPU runs at very high utilization rate, or when disks are almost full. Scalability is a relative property, but it can be accurately measured. A scalable system should react more gracefully to increased load than a non-scalable one. First, its performance should degrade more moderately than that of a non-scalable system. Second, its resources should reach a saturated state later, when compared with a non-scalable system. Even perfect design cannot accommodate 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 load from a faulty component to its backup can saturate the latter. Generally, having spare resources is essential for reliability concerns as well as for handling peak loads gracefully.

An inherent advantage that a distributed system has is a potential for fault-tolerance and scalability because of the multiplicity of resources. However, inappropriate design can obscure this potential. Fault-tolerance and scalability considerations call for a design demonstrating distribution of control and data. Any cen-

tralized entity, be it a central controller, or central data repository, introduces both a severe point of failure and a performance bottleneck. Therefore, a scalable and fault-tolerant DFS should strive for a model alluded above. A model where there are multiple and independent servers controlling multiple and independent storage devices.

The problem of dispersed storage devices is a key distinguishing feature of a DFS. The overall storage space managed by a DFS is composed out 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 storage device, independently from other units. All files belonging to the same component unit must be located on the same device. In Unix for instance, an entire removable file system [Ritchie and Thompson 1974] is a component unit since a file system must fit within a single disk partition. In all of the five covered systems, a component unit is a partial subtree of the Unix hierarchy.

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, that 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 on the disk the file is actually stored.

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 naming 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 83]. In this abstraction, both the existence of multiple copies and their location, are hidden.

2.1 Naming Structures

There are two related notions regarding name mappings in a DFS that need to be differentiated:

- *Location Transparency*. A name of a file does not reveal any hint, as of the file's physical storage location.
- *Location Independence*. A 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.

In practice, most of the current file systems (e.g. Locus, NFS, Sprite) provide a static, location transparent mapping for user-level names. These systems, however, do not support file migration, that is, automatic changing the location of a file is impossible. Hence, the notion of location independence is quite irrelevant for these systems. Files are permanently associated with a specific set of disk blocks. Disks can be manually moved between machines, but we do not consider this manual and cumbersome action as an action that can be considered as file migration. Only Andrew and some experimental file systems (e.g., Eden [Jessop et al. 1982; Almes et al. 1983] support location independence and file mobility. Andrew supports file mobility mainly for administrative purposes. A protocol provides migration of Andrew's component units upon explicit request, without changing neither the user-level names, nor the low-level names of the corresponding files (See Section 11.2 for details).

There are few aspects, that can further differentiate and contrast location independence, and static loca-

tion transparency:

- Divorcing data from location, as exhibited by location independence, provides better abstraction for files. A file name should denote the file's most significant attributes — its contents, and not its location. Location independent files can be viewed as logical data containers that are not attached to a specific storage location. Whereas, if only static location transparency is supported, the file name still denotes a specific, though hidden, set of physical disk blocks.
- Static location transparency provides users with a convenient way to share data. Users may share remote files by simply 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 the data objects. When files can be mobilized, the overall, system-wide 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.
- Location independence separates the naming hierarchy from the storage devices hierarchy and the inter-server structure. By contrast, if static location transparency is used (though 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 that is 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 possibilities 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 [Tichy and Ruan 1984] for instance, a file is uniquely identified by the name *host:local-name*, where local name is a Unix-like path. This naming scheme is neither location transparent nor location independent. Nevertheless, the same file operations can be used for both local and remote files. The structure of the DFS is a collection of isolated component units which are entire conventional file systems. In this first approach component units remained isolated, though 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 to attach remote directories to local directories, thus giving the appearance of a coherent directory tree. Only the attached, remote directories can be accessed transparently. There is some integration between the components to support transparent sharing. This integration, however, is limited and not uniform since each machine may attach different remote directories to its tree. The resulting structure is versatile. Usually it is a forest of Unix trees with shared subtrees.

Total integration between the component file systems is achieved using the third approach. A single global name structure spans all the files in the system. Ideally, the composed file system structure should be isomorphic to the structure of a conventional file system. However, in practice, there are many special files (e.g., Unix device files and binary directories) that makes the ideal goal very hard to attain. Different variations of this approach are examined in the sections on Unix United, Locus, Sprite and Andrew.

An important criterion for evaluation of the above naming structures is their administrative complexity. The most complex structure and the hardest to maintain is the NFS structure. Since any remote directory can be attached anywhere onto the local directory tree, the resulting hierarchy can be very unstructured. The effects of a failed machine, or taking a machine off-line, are that some arbitrary set of directories on different

machines becomes unavailable. In addition, a separate accreditation mechanism had to be devised for controlling which machine is allowed to attach which directory to its tree.

2.3 Implementation Techniques

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. This aggregation serves administrative purposes as well. Unix-like systems, use the hierarchical directory tree to provide name-to-location mapping, and aggregate files recursively into directories.

In order to enhance the availability of the crucial mapping information, methods such as replicating it, caching it locally, or both, are used. As was already noted, location independence means that the mapping changes in time, and hence, replicating the mapping renders a simple yet consistent update of this information impossible. A technique to overcome this obstacle is to introduce low-level, *location-independent file identifiers*. Textual file names are mapped to lower-level file identifiers that indicate which component unit the file belongs to, but are still location independent. These identifiers can be replicated and cached freely without being invalidated by migration of component units. A second level of mapping, that maps component units to locations and needs a simple yet consistent update mechanism, is the inevitable price. Implementing Unix-like directory trees using these low-level, location-independent identifiers makes the whole hierarchy invariant under component unit migration. The only thing that does change is the component unit-location mapping.

A common way to implement these low-level identifiers is with *structured names*. These are bit strings, that have usually two parts. The first part identifies the component unit that the file belongs to, and 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 either taking care not to reuse a name that is still used, or adding sufficiently more bits (this method is used in Andrew), or using a time stamp as one of the parts of the name (as done in Apollo Domain [Leach et al. 1982]).

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).

3 CONSISTENCY SEMANTICS

Consistency Semantics is an important criterion for evaluation of any file system that supports sharing of files. It is a characterization of the system that specifies the semantics of multiple clients accessing a shared file simultaneously. In particular, these semantics should specify when modifications of data by a client are observable by other clients. Borrowing from the Database jargon, consistency semantics are the effects of a concurrency control policy.

For the following discussion we need to assume that a series of file accesses (i.e., Read's and Write's) attempted by a client to the same file, are always enclosed between the Open and Close operations. After an Open operation has been performed the file is said to be open. Similarly, after a Close operation we say that a file is closed. We denote such a series of accesses as a *file session*.

A session is akin to the database transaction, but there are major differences. A transaction can involve more than one file, and more importantly transactions have the additional aspect of atomic update. This aspect is orthogonal to the current discussion, where we assume that all file accesses are completed normally.

To illustrate the concept, we sketch several prominent examples of consistency semantics that are mentioned in this paper. We outline the gist of the semantics and not the whole detail:

Unix Semantics

- Writes to an open file by a client are visible immediately by other clients that have this file open at the

same time.

- There is a mode of sharing where clients share the pointer of current location into the file. Thus, the advancing of the pointer by one client affects all sharing clients.

Here, a file has a single image that interleaves all accesses, regardless of their origin. These semantics lend themselves to an implementation where a file is associated with a single physical image that is accessed as an exclusive resource. Contention for this single image results in clients being delayed. The unique mode of sharing mentioned above is an artifact of Unix and is needed primarily for compatibly 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.

Session semantics

- Writes to an open file by a client are not visible immediately by other clients that have the same file open simultaneously.
- Once a file is closed, the changes made to it are visible only in sessions starting later. 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. Notice that almost no constraints are enforced on scheduling accesses. Therefore, it is possible to schedule accesses in such a way that their effects are not reproducible by any serial scheduling. Since accesses are not delayed and serialization is not enforced, these semantics are not a good match for a distributed database application. These semantics are used in the Andrew file system.

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 very simple since the sharing is very disciplined (read only).

Transaction-like Semantics

Identifying a file session with a Database 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. 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, their notion of a transaction is equivalent (in the concurrency control sense) to our notion of a file session.

Selecting and implementing consistency semantics are closely related to the remote access methods presented in the following Section.

4 REMOTE ACCESS METHODS

Consider a client process that requests to access (i.e., Read or Write) a remote file. Assuming that the server storing the file was located by the naming scheme, the actual data transfer to satisfy the client 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 is not already cached, then a copy of that data is brought from the server to the client. Accesses are performed on the cached copy, in the client side. The idea is to retain recently-accessed disk blocks in cache, so that repeated accesses to the same information can be handled locally, without additional network traffic. 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 to preserve the relevant consistency semantics. The problem of keeping the cached copies consistent with the master file is referred in the literature as the *cache consistency problem*.

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 just an extension of caching or buffering techniques in conventional file systems (e.g., buffering block I/O in Unix [Mckusik 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 actually 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 NFS and Locus, for instance, the implementation is based on Remote Service but is augmented with Caching for performance. On the other hand, Sprite's implementation is based on Caching, but under certain circumstances a Remote Service method is adopted. Thus, when we evaluate the two methods we actually evaluate to what degree should one method be emphasized over the other.

The Remote Service method is quite straightforward, and does not require further explanation. Thus, the bulk of the following material is concerned with the method of Caching.

4.1 Designing a Caching Scheme

The following discussion pertains to (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. Similar considerations are relevant for a 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]:

- What is the granularity of cached data?
- Where should the client's cache be kept, main memory or local disk?
- How are modifications on cached copies going to be propagated?
- How can a client determine whether its cached data is consistent or not?

The choices in answering these questions are intertwined and are related to the selected consistency 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 is cached than is needed to satisfy a single access so that many accesses can be served by the cached data. Andrew caches entire files, but it is not intended to handle very large files. The rest of the systems support caching of individual blocks driven by clients' demand. Increasing the caching unit increases the hit ratio on the one hand, but delays the actual data transfer and increases the potential for consistency problems. Selecting the unit of caching involves parameters such as the network transfer unit, and the Remote Procedure Call (RPC hereafter)

protocol service unit (in case an RPC protocol is used) [Birrel and Nelson 1984]. The network transfer unit (Ethernet packets) are about 1.5K bytes, so big units of cached data need to be disassembled for delivery and re-assembled upon reception [Welch 1986].

Block size and the total cache size are obviously of importance for block-caching schemes. In Unix-like systems, common block sizes are 4K or 8K bytes. For large caches (over 1M bytes) large block sizes (over 8K bytes) are beneficial [Ousrerhout et al. 85; Lazowska et al. 86]. 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 is kept on disk, it is still there during recovery and there is no need to fetch it again. On the other hand, main-memory caches have several advantages of their own. First, main memory caches permit workstations to be diskless. Second, data can be accesses more quickly from a cache in main memory than on a disk. Third, the current technology trend is bigger and cheaper memories. The achieved performance speedup is predicted to outweight disk caches advantages. Fourth, the server caches (the ones 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).

4.1.3 Modification Policy

The policy used to write dirty blocks back to the server's master copy has critical effect on the system's performance and reliability. The simplest policy is to write data through to disk as soon as it is placed on any cache. The advantage of *write-through* is its reliability. Little information is lost when a client crashes. However, this policy requires each write access to wait until the information is sent to the server, which results in poor Write performance. Caching with write-through is actually 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 sometime later. This policy has two advantages over write-through. First, since writes are to the cache, Write accesses complete much more quickly. Second, data may be deleted before it is written back, in which case it needs 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 long time before they written back to the server [Ousterhout et al. 85]. 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 thirty seconds interval.

Yet another variation on delayed-write is to write data back to the server when the file is closed. This policy, *write-on-close*, is used in the Andrew system. In case of files that are open for very short periods or are 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 open for long periods and are modified frequently.

As a reference, we present some data regarding the utility of Caching in Unix 4.2 BSD. Unix 4.2 BSD uses a cache of about 400K bytes holding different size blocks (the most common size is 4K bytes). A delayed-write policy with about thirty seconds intervals is used. A miss ratio (ratio of the number of real disk I/O to logical disk accesses) of 15% is reported in [McKusic et al. 84], and of 50% is reported in [Ousterhout et al. 85]. The latter paper also provides the following statistics which were obtained by simulations on Unix: A

4M bytes cache of 4K bytes disk blocks eliminates between 65% and 90% of all disk accesses for file data. A write-through policy resulted in the highest miss-ratio, and 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 the consistency semantics. Write-on-close is very suitable for Session Semantics. By contrast, using any kind of delayed-write policy when files are updated concurrently 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 its locally cached copy of the data is consistent with the master copy (and hence can be used) or not. If the client determines that its cached data is out-of-date, accesses cannot be served by that cached data any longer. An up-to-date copy of the data needs to be cached. There are basically two approaches to verify 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 is consistent with the master copy. The frequency of the validity check is the crux of this approach and determines the resulting consistency semantics. It can range from a check before every single access, to a check only on first access to a file (on file open basically). 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., i-node information in Unix). Depending on the frequency it can load both the network and the server. This phenomenon was the cause for Andrew designers to withdraw from this approach (refer to [Howard et al. 1988] for detailed performance data).
- *Server initiated approach.* The server records for each client, the (parts of) files it caches. When the server detects a potential for inconsistency, it is its responsibility to react. A potential for inconsistency occurs when a file is cached by two different clients in conflicting modes. If Session Semantics are implemented, then 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. Implementing Session Semantics, the server need not be informed about Open's of already cached files. The server's reaction is triggered only by the Close of a writing session, and hence only this kind of sessions are delayed. In Andrew, Session Semantics are implemented, and a server initiated method, called *callback*, is employed.

  On the other hand, if a more restrictive consistency semantics is implemented, like Unix Semantics, the server must be much more involved. The server must be notified whenever a file is opened and the intended mode (Read or Write mode) must be indicated for every Open. 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). Actually, disabling caching results in switching to Remote Service mode of operation.

  A problem with this 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 here trades longer accesses and greater server load using the former method, for larger server state using the latter.

4.2 Cache Consistency

Before delving into the evaluation and comparison of the two methods, we relate them to the various examples of consistency semantics introduced in the Section 3.

- Session Semantics are a perfect match for caching of 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 of parts of files defeats the purpose of simple implementation.
- A distributed implementation of Unix Semantics using caching would be complex because of the need to serialize accesses to the cached copies of the file. Some kind of distributed conflict resolution scheme is needed for this end. In addition, once a cached copy is modified, the changes need to be propagated immediately to the rest of the cached copies. Frequent Write's can generate tremendous network traffic, and cause long delays before requests are satisfied. As was already stated, these semantics lend themselves to an implementation where a file is accessed as an exclusive resource. 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 presented in [Schroeder et al. 1985] were invented for a whole file caching scheme. Observe that with this semantics the cache consistency problem vanishes totally.
- Serialization of transactions can be implemented in a straight forward manner using locking, when they are all served by the same server on the same machine as done in Remote Service.

4.3 A Comparison of Caching and Remote Service

Essentially, the choice between Caching and Remote Service trades a potential of increased performance, for simplicity. We evaluate this trade-off by listing the merits and demerits of the two methods.

- A substantial amount of the remote accesses can be handled efficiently by the local cache when Caching is used. 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. Moreover, servers are contacted only occasionally and not for each access. Consequently, server load and network traffic are reduced and the potential for scalability is enhanced. By contrast, each and every remote access is handled across the network when employing the Remote Service method. 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 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 big, contiguous segments of data rather than random disk blocks. (The last two reasons indicate strongly the merits of whole file caching as done in Andrew).
- The cache consistency problem is the major drawback of Caching. In access patterns that exhibit infrequent Write's, caching is superior. However when Write's are frequent, the mechanisms employed to overcome the consistency problem incur substantial overhead in terms of performance, network traffic, and server load.
- It is hard to emulate the consistency semantics of a centralized system, in a system employing caching as its remote access method. The reason is the cache consistency problem. Namely, the fact that accesses are directed to distributed copies, and not to a central data object. Observe that the two Caching-oriented semantics, Session Semantics Immutable Shared Files Semantics, are not restrictive and do not enforce serializability. On the other hand, using Remote Service, the server serializes all accesses, and hence, is able to implement any centralized consistency semantics.
- In order to use Caching and benefit from its merits, clients must have either local disks or large main memories. Diskless clients can use the Remote Service methods without any problems.
- Since data is transferred *en masse* between the server and client, and not in response to the specific needs of a file operation, the lower inter-machine interface is quite different from the upper client inter-

face. The remote service paradigm on the other hand, is just an extension of the local file system interface across the network. Thus, the inter-machine 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 two kinds of fault tolerance issues:

- Tolerating faults occurring when some service activity is being carried out. For instance, a server crash on active clients belongs to this category. Section 5.1 examines two service paradigms in this context.
- Tolerating faults occurring after data was deposited with the file system. For instance, a machine failure that temporarily prevents access to a file stored there. These kinds of faults can hinder the *availability* of files. In Section 5.2 we define the concept of availability and discuss how to increase the availability of files. Section 5.3 reviews file replication as another means for enhancing availability.

5.1 Stateful versus Stateless Service

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 which 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. 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 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, though 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. Clearly, it is easier to build a stateless service than a stateful service on top of a datagram communication protocol.

The distinction between stateful and stateless service becomes evident when considering the effects of a crash occurring during some service activity. A stateful server loses all its volatile state in a crash. A graceful recovery of such 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 state of crashed clients. This phenomena is 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 any difficulty. Therefore, the effects of server failures and recovery are almost not noticeable. There is no difference between a slow server and a recovering server from a client's point of view. 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 additional constraints on the design of the DFS. First, since each request identifies the target file, a uniform, system-wide, low level naming is advised. Translating remote to local names for each request would imply

even slower processing of the requests. Second, since clients re-transmit requests for files operations, these operations must be *idempotent* to prevent races. 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). However, 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, or Internetworks are used, reordering of messages is possible. A stateful, virtual-circuit oriented service, would be preferable in such a case. Also observe that if the server employs 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.

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 employs a stateless service, does not use file descriptors, and does include explicit offset in every access.

5.2 Improving Availability

Svobodova 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 a 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" [Svobodova 1984]. A robust file is not necessarily recoverable and vica versa. Different techniques must be used to implement these two distinct concepts. Recoverable files are realized by atomic update techniques. Locus for example, uses shadow paging technique to make its files recoverable. We do not give account of atomic updates techniques in this paper. Robust files are implemented by redundancy techniques such as stable storage [Lampson 1981]. We find it necessary to add an additional criterion which we call *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 very 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 files which are described bellow.

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 still if the CSS cannot be reached by a client, the file may be available, but not to that particular client. In general, having more than two machines involved in a file operation can cause bizarre situations where 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. However, locating a 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. In principle, most systems (e.g., Locus, NFS, Andrew) approach the problem by requiring that each component 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 the directories in its pathname. In a situation that can arise, 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. Andrew and NFS use this technique. Sprite employs 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 both faster and guarantees better availability. (For complete description of the prefix table mechanism refer to Section 9.2).

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, as demonstrated by mirrored disks [Lampson 1981]. Multi-machine replication can benefit performance too, since selecting a nearby replica to serve an access request, results in shorter service time.

The very 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 opaque 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. However, at some level 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, and thus, an update to any replica must be reflected on all other replicas. More precisely, the relevant consistency 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. 85] gives a survey of 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 (hopefully rare) circumstance of catastrophic failures for the sake of guaranteed progress. Among the surveyed systems, Locus employs 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).

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 au-

tomatic replication control policy. Demand replication means that reading a non-local replica causes it to be cached locally, thereby generating a new non-primary replica. Updates are performed only on the primary copy and cause all other replicas to be invalidated by sending appropriate messages. Synchronization of the invalidation is not guaranteed. Hence, it is possible that a stale replicas 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 large extent, are still visionary. Andrew is the closest system to be classified as a very large scale system with a planned configuration of couple of thousands workstations. There are no magic guidelines to ensure the scalability of a system. It is easier to point out current designs and indicate why they are not scalable. In Section 6.1 we discuss several designs that pose problems and propose possible solutions, all in the context of scalability. Section 6.2 describes an implementation technique, Light Weight Processes, that is essential for high-performance and scalable designs.

6.1 Guidelines by Negative Examples

Barak and Kornatzky [Barak and Kornatzky 1987] list several principles for designing very large scale systems. The first one 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 service mechanism whose load demand is proportional to the size of the system is destined to become clogged once the system grows beyond a certain size. Adding more resources would not alleviate such a problem. The capacity of this mechanism simply limits the growth of the system. This is why the CSS of Locus is not a scalable design. In Locus, every file group (the Locus component unit, which is equivalent to a Unix removable file system) is assigned a CSS, whose responsibility is to synchronize accesses to files in that file group. Every Open request to a file within that file group must go through this machine. Beyond a certain system size, CSS's of frequently accessed file groups are bound to become 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 simply is 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 state it maintains for them. It is interesting to contrast the ways MOS [Barak and Litman 1985; Barak and Paradise 1986] and Locus reclaim obsolete state storage on servers. The approach taken in MOS is garbage collection. It is the client's responsibility to reset an expiration date on state objects the servers maintain for it. Clients reset this date whenever they access the object 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 machine determines that a particular machine is unavailable. Among other things, this procedure releases space occupied by 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 (its own) of objects, whereas a failure detection mechanism is not scalable since it depends on the size of the system.

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, or central file server. Centralization is a form of functional asymmetry between machines comprising the system. The ideal alternative is a configuration which is functionally symmetric; that is, all the component machines have

equal role in the operation of the system, and hence each machine has some degree of autonomy. Practically, it is virtually impossible to comply with such a principle. For instance, incorporating diskless machines violates functional symmetry. However, autonomy and symmetry are important goals to aspire to.

The practical approximation to symmetric and autonomous configuration is *clustering*. The system is partitioned into a collection of semi-autonomous clusters. A cluster consists of a set of machines and a dedicated cluster server. In order to make cross-cluster file references relatively infrequent, each machine's requests should be satisfied by its own cluster server most of the time. Of course, this depends on the ability to localize file references and placing the component units appropriately. If the cluster is well balanced, that is the server in charge suffices to satisfy all 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. Andrew's clusters and cluster servers are a good example for a scalable clustering scheme.

Unix United is a system that exemplifies the concept of autonomy. There, Unix systems are joined together in a recursive manner [Randell 1983] to create a bigger and bigger global system. Each component system is a complete Unix system that can operate independently. Again, modular and autonomous components are combined to create a large scale system.

6.2 Light Weight 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 process server, is certainly not a good choice, since whenever a request necessitates disk I/O the whole service is blocked. Assigning a process for each client is a better choice, however the expensive price of frequent context switches between the processes 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, service tables etc. In Unix 4.2 BSD processes are not permitted to share address spaces, hence sharing must be obtained externally by using files and other unnatural mechanisms

It appears that one of the best solutions for the server architecture is the use of *Light Weight Processes* or *Threads*. A thread is a process that has very little non-shared 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 light weight processes is that of multiple threads of control associated with some shared resources.

There are many alternatives regarding threads; we briefly mention few of them. Threads can be supported by the kernel (as in Mach [Tevanian et al. 87]) or above the kernel, at the user level (as done in Andrew]). Usually a light weight process is not bound to a particular client. Instead, it serves single requests of different clients. Scheduling threads can be preemptive or non-preemptive. If threads are allowed to run to completion, then their shared data need not be explicitly protected. Otherwise, some explicit locking mechanism must be used.

It is quite clear that some form of light weight processes scheme is essential for servers to be scalable. Locus, Sprite, Andrew, and in the future NFS too, employ such schemes. Detailed studies of threads implementations can be found in [Tevanian et al. 87] and [Kepecs 85].

7 Unix United

The *Unix United* project from the University of Newcastle upon Tyne, England, is one of the earliest attempts aimed at scaling up 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 compo-

nent or constituent systems), so as to construct a distributed system which 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 PDP11's connected by a Cambridge Ring.

The system is presented in two levels of detail. First, an outlook of Unix United is given. Then the implementation, the *Newcastle Connection* layer, is examined, and some issues regarding networking and internetworking are discussed.

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 where each user can read or write any file, use any device, execute any command, or inspect any directory, regardless of which system it belongs to.

The component unit is a complete Unix tree belonging to a certain machine. The position of these component units in the naming hierarchy is quite arbitrary. They can appear in the naming structure in positions subservient to other component units (directly or via intermediary directories).

Roots of component units are assigned names so that they become accessible and distinguishable externally. A file system's own root is still referred to as '/' and still serves as the starting point of *all* pathnames starting with a '/'. However, a subservient file system 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 7.1, unix1, unix2, unix3, and unix4 are names of component systems. 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 that 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/unix3/f4`.

Observe that a user is aware of the upward boundaries of his current component unit since he must use the '../' syntax whenever he wishes to ascend outside of his current machine. Hence, Unix United does not provide complete location transparency.

The traditional root directories (e.g. `/dev`, `/temp`) 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 a conventional Unix.

Each component system has its own set of named users and its own administrator (super-user). The latter is responsible for the accreditation for users of his own system as well as remote users. Remote users identifiers are prefixed with the name of their original system for uniqueness. Accesses are governed by the standards Unix file protection mechanisms, even if they cross components boundaries. That is, there is no need for a user to log in separately or provide passwords, when he accesses remote files, in case he is properly accredited. However, users wishing to access files in a remote system, must arrange with the specific system administrator separately.

It is often convenient to set the naming structure as to reflect organizational hierarchy of the environment in which the system exists.

7.2 Implementation—the 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 (see Figure 7.2). It intercepts all system calls concerning files, and filters out those that have to be redirected to remote systems. Also, the Connection layer accepts system calls that have been directed to it from other systems. Remote layers communicate by the means of a RPC protocol.

Incorporating the Connection layer enables preserving both the same Unix system call interface and the Unix kernel, in spite of the extensive remote activity the system carries out. The penalty of preserving the kernel intact is the fact that the service is implemented as user-level daemon processes which slow down remote operation.

Each Connection layer stores a partial skeleton of the overall naming structure. Obviously, each system stores locally its own file system. In addition, each system stores 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). Figure 7.3(a), (b) and (c) show the partial skeletons of the hierarchy of the file systems of Figure 1 as maintained by the systems unix1, unix2, and unix3 respectively (only the relevant parts are shown).

The fragments maintained by different systems overlap and hence must remain consistent, a requirement that makes changing the overall structure a very 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 user 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 which makes it callable from any external process. This 'spawner' process performs the remote access rights checks according to a machine-user identification pair. Also, it converts this identification to a valid local name. For the sake of preserving Unix semantics, once a user process forks, its file service process forks as well. File descriptors (and not lower level means such as inodes) are used to identify files between a user and its file server. This service scheme does not excel in terms of robustness. Special recovery actions has to be taken in case of both sever and user failures. However, the Connection layer attempts to mask and isolate failures resulting from the fact that the system is a distributed one.

7.3 Networking issues

Unix United is well suited for a diverse inter-network topology, spanning LANs, as well as direct links and even WAN's. The logical name space needs to be properly mapped onto routing information in such a complex inter-network. An important design principle is that the naming hierarchy needs bear no relationship to the topology. The approach taken is that each machine routes remote requests to an appropriate one of its physically adjacent machines, which can then pass the request on further. Therefore, each system has to be aware of hardware addresses only of machines that are directly connected to it (by the same LAN or a hard-wired link). Unix United takes advantage of its own system-wide unique naming scheme to guide in addressing and routing. Within each machine routing tables translate these logical names to the next hop *en route* to the destination.

When pathname traversal is performed, the pathname is followed as an inter-machine route, so that all

authentication and protection demands are satisfied. However, it may not be necessary to repeat the same route when a file has been successfully opened, if some shorter route is available.

As a message travels along the physical links the addressing information it contains (which is described in terms of paths) will sometimes have to be adjusted to reflect the fact that movement around the physical topology causes movement around the naming structure (recall that there is no relationship between the two). For instance, in Figure 7.4, a message from `U1` to `U2` would use the path `/../U2`. Passing through `M1` this message would need this path to be changed to `/../M2/U2`.

7.4 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. The naming scheme is quite unique, since all pathnames are relative to some context and there are no absolute pathnames.
- *Recursive Structuring.* 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.
- *Inter-networking.* The intention of constructing very large distributed systems is evident also from the aspect of inter-networking capabilities Unix United has. The system is suited to operate on a variety of networks including both LANs and WANs. Attention was given to separating the network topology from the name structure.
- *The Connection Layer.* Conceptually, the connection layer implementation is very clean and simple. It is a modular subsystem interfacing two existing layers without modifying neither of them nor 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.

Finally, we note that Unix United most valuable contributions to the state of the art of DFSs should not be measured by its actual implementation and performance, but rather by the concepts it spurred and gave rise to.

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, full implementation of nested transactions [Weinstein et al. 1985], remote tasking and ability to withstand (to a certain extent) failures and network partitions. The system has been operational in UCLA for several years on a set of mainframes and workstations connected by an Ethernet. In [Sheltzer and Popek 1986], a general strategy for extending some of Locus features to an internet environment is outlined.

The heart of the Locus architecture is its DFS. We give first an overview of the features and general implementation philosophy of the file system. In the next sections we discuss in more detail the static nature of the file system (Sections 8.2) and its dynamics (Sections 8.3, and 8.4). Section 8.5 is devoted to 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 transparent; it is not possible to discern from a name of an object the 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 sites. 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 recent available version. Users may have control on both the number and location of replicated files when a file is created as well as later on. Conversely, users may prefer to be totally unaware of the replication scheme. In Locus, file replication serves mainly to increase availability for reading purposes facing failures and partitions. A primary copy approach is adopted for modifications.

Locus adheres to the same file access semantics that standard Unix presents users with. Locus strives to provide these semantics in the distributed and replicated environment in which it operates. Alternate mechanisms of advisory and enforced locking of files and parts of files are also offered. Moreover, atomic updates of files are supported by commit and abort system calls.

Operation facing failures and network partitions is emphasized in Locus design. As long as a copy of a file is available, read requests can be served, and it is still guaranteed that the version read is the most recent available one. Automatic mechanisms take care to update stale copies of files upon the merge of their storage site to a partition.

Emphasizing high performance in the design of Locus led to incorporating networking functions (such as formatting, queueing, transmitting, and retransmitting of messages) into the operating system. Specialized remote operations protocols were devised for kernel-to-kernel communication, in contrast to the prevalent approach of using RPC protocol, or some other existing protocol. Lack of multi-layering enabled achieving high performance for remote operations. On the other hand, this specialized protocols hamper Locus portability to different networks and file systems.

An efficient but limited process facility called *server processes* (or light-weight processes) is devised for serving remote requests. These are processes which have no non-privileged address space. All their code and stack are resident in the operating system nucleus, they can call internal system routines directly, and can share some data. These processes are assigned to serve network requests which 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 The Name Structure

The logical name structure disguises both location and replication details from users and applications. Virtually, logical *filegroups*[2] 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, inode number⟩, which will be referred to as a *file's designator*, serves as a globally unique low-level name for a file. Observe that the designator itself hides both location and replication details.

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. An entry records the file designator of the directory over which the filegroup is logically mounted, and indication of which site is currently responsible for access synchronization (the function of this site is explained subsequently) within the filegroup. In addition, each site that stores a copy of the directory over which a sub-tree is mounted must keep that directo-

[2] A removable Unix file system, in Locus terms, is called a filegroup. A filegroup is the component unit in Locus.

ry's inode in core with an indication that it is mounted over. This is done so that any access from any site to that directory will be caught, allowing the standard Unix mount indirection [Ritchy and Thompson 1974; Quarterman et al. 1985] to function (via the logical mount table). 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 and are assigned pack numbers which together with logical filegroup number identify an individual pack. 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 exhibit almost read-only character and are crucial for pathnames translation of most of the files.

The various copies of a file are assigned the same inode number on all the filegroup's packs. Consequently, a pack has an empty inode slot for all files that 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 its physical numbers. To facilitate automatic replication management, each inode of a file copy contains a version number, determining which copy dominates other copies.

Each site has a container table, which maps logical filegroup numbers to disk locations for the filegroups that have packs locally on this site. When requests for accesses to files stored locally get to a site, this table is consulted to map the file designator to a local disk address.

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

8.3 File Operations

Locus approach to file operations is certainly a departure from the prevalent client-server model. The fact that files are replicated, and the intention to provide synchronized accesses necessitate an additional function. Therefore, Locus distinguishes three logical roles in file accesses; each one potentially performed by a different site:

- *Using Site* (US), which issues the requests to open and access a remote file.
- *Storage Site* (SS), which is the selected site to serve the requests.
- *Current Synchronization Site* (CSS), which enforces global synchronization policy for a filegroup, and selects a SS for each and every 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., a partition). The CSS maintains the version number, and a list of physical containers for every file in the filegroup.

The following subsections describe the open, read, write, close, commit, and abort operations, as they are carried out by the above entities. Related synchronization issues are described separately in the next section.

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.

Opening a file given its designator commences as follows. The US determines the relevant CSS by looking up the filegroup in the logical mount table, and then forwards the open request to the CSS. The CSS polls potential SS's for that file to decide which one of them will act as the real SS. In its polling messages, the CSS includes the version number for the particular file, so that the potential SS's can, by comparing this number to their own, decide whether their copy is up to date, or not. The CSS selects a SS by considering the responses 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 inode structures for the opened file. The CSS needs this information to make future synchronization decisions, and the SS maintains the inode in order to serve efficiently forthcoming accesses.

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 guess as to where the in core inode is stored in the SS. Once the inode 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 stored in a kernel buffer. Once a page was 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 read, 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). Observe that there is no parallel to NFS's remote lookup operation and that 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 the CSS is even not informed of such access.

8.3.2 Write and Close

In Locus, a primary copy policy is employed for file modification. The CSS has to select the primary copy pack site as the SS for an open 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 read from the SS using the read protocol. If the change involves the entire page, 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 inode structures, and so that the CSS can alter state data which might affect its next synchronization decision.

Observe that in both read and write operations, caching of data pages is relied upon heavily. The validation of the cached data is dealt with in Section 4.

Commit and abort system calls are provided, and closing a file commits it. If a file is open for modification by more that one process, the changes are not made permanent until one of the processes issues a commit system call or until they all close the file.

8.3.3 Commit and Abort

Locus uses shadow page mechanism for implementing atomic commit. The main disadvantage of shadow paging (that is, the inability to maintain physical relationship among data pages) is not a problem in a Unix file system, or in general where record-level updates do not predominate.

When a file is modified, shadow pages are allocated at the SS. The in-core copy of the disk inode is updated to point to these new shadow pages. The disk inode is kept intact pointing to the original pages. The atomic commit operation consists merely of moving the in-core inode, to the disk inode. After that point, the file contains the new information. To abort a set of changes, one merely discards the in-core inode information, and frees up the disk space used to record the changes. The US function never deals with actual disk pag-

es, 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 SS's of the modified file as well as the CSS. At minimum these messages identify the modified file and contain the new version number (in order to prevent attempts to read the old versions). At this point, it is the responsibility of these additional SS's to bring their version up to date by propagating in 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

Locus tries to emulate conventional Unix semantics on file accesses in a distributed environment. In standard Unix, multiple processes are permitted to have the same file open concurrently. These processes issue read and write system calls and the system guarantees that each successive operation sees the effects of the ones that precede it. This is 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. Since remote tasking is supported in Locus, such situations can arise when the sharing processes do not co-reside on the same machine and hence complicate the implementation significantly.

There are two sharing modes to consider. First, in Unix, several processes descending from the same ancestor process can 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, a site can proceed with executing system calls needing the offset.

Secondly, in Unix the same in-core inode for a file can be shared by few processes. In Locus, the situation is much more complicated, since the inode of the file can be cached at several sites. Also, data pages are cached at multiple sites. A multiple data-tokens scheme is used to synchronize sharing of file's inode and data. A single exclusive writer, multiple readers policy is enforced. Only a site with the write token for a file may modify the file, and any site with a read token can read it. Both token schemed are coordinated by token managers operating at the corresponding storage sites.

The cached data pages are guaranteed to contain valid data only when the file's data token is present. When the write data token is taken from that site, the inode is copied back to the SS, as well as all modified pages. Since arbitrary changes may have occurred to the file 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 inode 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.

The above mechanism guarantees consistency; each access sees the most recent data. A different issue regarding access synchronization is serializability of accesses. To this end 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 it can 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, strict synchronization among copies of a file, so that all uses of that file within that partition see the most recent version.

The primary copy approach eliminates the possibility of conflicting updates, since the primary copy must be in the user's partition to allow an update. However, the problem of detecting updates, and propagating them to all the copies remains, especially since updates are allowed in a partitioned network. During normal operation, the commit protocol ascertains proper detection and propagation of updates as was described earlier. However, a more elaborate scheme has to be employed 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 reflected in the pack. Also, the primary copy pack keeps a complete list of all the recent commits in secondary storage. When a pack joins a partition it contacts the primary copy site, and checks whether its *lwm* is within the recent commit list bounds. If this is the case, the pack site schedules a kernel process which brings the pack to a consistent state by performing the missing updates. If the primary pack 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 that it would be informed of the most recent available (in the partition) version of each file in the filegroup. Once the new CSS accomplishes this 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 possible to Read out-of-date replicas of a file. Thus, Locus sacrifices consistency for the ability to continue and 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 in order to redo the changes. Instead, the site must inspect the entire inode space to determine which files in its pack are out of date.

When a site is lost from an operational Locus network a cleanup procedure is necessary. Essentially, once a site has decided that a particular site is unavailable, it must invoke failure handling for all resources which local processes were using at that site and for all local resources which processes at that site were using. 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 [Walker et al. 1983], conflicts among updates in different partitions may arise. However, because of the simple nature of directory entries modification, an automatic reconciliation procedure is devised. This procedure is based on comparing the inodes 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 inodes, and amounts to altering slightly the files name and notifying the files owners by electronic mail.

8.6 Summary

We summarize by outlining an overall profile and evaluation of Locus by pointing out the following issues:

- *A distributed operating system.* Because of the multiple dimensions of transparency in Locus, it comes close to [Tanenbaum and Van Renesse 1985] definition of a truly distributed operating system. At least from the file system aspect, a user is totally unaware of the multiplicity of machines, and to a large extent is presented with the illusion of a uni-processor Unix machine. Moreover, a user is partially shielded from effects of various failures of machines and the network. One of the prominent impacts of this characterization is Locus extension into the network. The common pattern in Locus is kernel to kernel communication via specialized, high-performance protocols, rather than incorporating some layered, generic network service. Essentially, kernel augmentation is the implementation strategy in Locus.
- *Replication.* A primary copy replication scheme is employed in Locus. The main merit of this kind of replication scheme is increased availability of directories which exhibit high read-write ratio. Availabil-

ity 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.* Locus does not ignore this complex issue. The sharing mode of Unix are emulated to the last detail, in spite of caching at multiple US's. Alternatively, locking facilities are provided.
- *Fault Tolerance.* Substantial effort has been devoted to designing mechanisms for fault tolerance. We name a few: An atomic update facility, merging of replicated packs after recovery and a degree of independent operation of partitions.

  The resulting robustness 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.
  - The CSS function introduces an additional point of failure. For a file to be available for opening, both the CSS for the filegroup and a 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 in-core 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 in-core space, resulting in a pure overhead of message traffic.
- *Scalability* Locus does not lend itself for very large distributed system environment, mainly because of the following reasons.
  - A CSS per an entire filegroup can easily become a bottle-neck for heavily accessed filegroups.
  - A logical mount table replicated at all sites is clearly not a scalable mechanism. Distributing this mapping table in a clever way is not an easy task.
  - 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 the 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.

Finally, we note that there is a lot to be learned from Locus, merely because of the attempt to build a novel system.

9 Sprite

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

An overview of the file system and related aspects is given in the Section 9.1. Section 9.2 elaborates on the file lookup mechanism (called prefix tables) and Section 9.3 on the caching methods employed in the file system.

9.1 Overview

Sprite designers envision the next generation of workstations as powerful machines with vast physical memory3. The configuration they target Sprite for, is that of large and fast disks concentrated on a few server machines servicing the storage needs of hundreds of diskless workstations inter-connected by several connected LANs. By caching files, the large physical memories will compensate for the lack of local disks.

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.

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

Unlike NFS, sprite enforces consistency of shared files. Each read system call is guaranteed to return the most up-to-date data for a file, even if it is being opened concurrently by several remote processes. Thus, as with Locus, Sprite emulates a single time-sharing Unix system on a distributed environment.

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 server's (physical) cache faster than from local disk, which means that a server with a large cache may provide better paging performance than local disk.

The virtual memory and file system share the same cache and negotiate on how to divide it 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.

We briefly mention some other features of Sprite. In contrast to Unix, where only code can be shared among processes, Sprite provides a mechanism for sharing an address space between user processes on a single workstation. Process migration facility which is transparent both to users as well as the migrated process, is also provided.

9.2 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* [Welch and Ousterhout 1986], whose function is to map domains to servers. The mapping is built and updated dynamically by a broadcast protocol. We first describe how the tables are used during name lookups, and later describe how the tables change dynamically.

Each entry in a prefix table corresponds to one of the domains. It contains the name of the topmost directory in the domain (called prefix for the domain), the network address of the server storing the domain and

[3] Currently, workstations have 4M to 32M bytes of main memory. Sprite designers predict that memories of 100M to 500M bytes will be commonplace in few years.

a numeric designator identifying the domain's root directory for the storing server. Typically 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 remaining 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 and then it 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 where the server does not complete the lookup:

- When the server encounters an absolute pathname in a symbolic link it returns to the client the absolute pathname without any further ado. The client looks up the new name in its prefix table and initiates another lookup with a new server.
- A pathname can ascend past the root of a domain (because of a parent, i.e., '..', component). In such a case the server returns the remainder of the pathname to the client. The latter combines the remainder with the prefix of the domain that was just exited to form a new absolute pathname.
- A pathname can descend down into a new domain too. This can happen when an entry for a domains 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. Alternatively, when a root of a domain is beneath a working directory and a file in that domain is referred to with a relative pathname, the server cannot complete the translation, too. The solution to these situations is to place a marker to indicate domain boundaries (a mount point in NFS and Locus jargon). The marker is a special kind of file called a *remote link*. Similarly 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.

Relative pathnames are treated similarly to the way they are treated in conventional Unix. When a process specifies a new working directory, the prefix mechanism is used to open the working directory and both its server address and designator are saved in the process's state. When a lookup operation detects a relative pathname, it sends the pathname directly to the server for the current working directory along with the latter's designator. Hence, from the server's point of view there is no difference between relative and absolute name lookups.

So far, the key difference from mappings based on the Unix mount mechanism was 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) which employ name lookup cache get a similar effect of avoiding the component by component lookup once the cache holds the appropriate information.

Prefix tables are 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. The 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 then can 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 gradually when 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 attempts a broadcast query. If the server has become available again, it responds to the broadcast and the

prefix table entry is re-established. This same mechanism also works if the server reboots at a different network address, or if its domains are moved to other servers.

The prefix mechanism ensures that whenever a server storing a domain is up, the domain's files can be opened and accessed from any machine regardless of the status of the servers of domains above the particular domain. Essentially, the built-in broadcast protocol enables dynamic configuration 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 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 that servers storing a replicated domain give different clients different prefix entries (standing for different replicas) for the same domain. The same technique can be used for sharing binary files by different hardware types of machines.

Since the prefix tables bypass part of the directory 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.

9.3 Caching and Consistency

An important aspect of the Sprite file system design is the extent of using caching techniques. Capitalizing on the big main memories and advocating diskless workstations, file caches are stored in-core instead of local disks (as in Andrew). Caching is used by both client and server workstations. The caches are organized on a block basis rather than a file basis (as in Andrew). Blocks are currently 4K bytes. 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 enables clients to create new blocks in the cache and locate any block without the file inode being brought from the server.

When a read kernel call is invoked to read a block of a file, the kernel first checks its cache and returns the information from the cache, if it is present. If the block is not in the cache, the kernel reads it from disk (if the file is locally stored), or requests it from the server; in either case the block is added to the cache replacing the least recently used block. If the block is requested from the server, the server checks its own cache before issuing a disk I/O and adds the block to its cache, if it was not already there. Currently, Sprite does not use read-ahead[4] to speed up sequential read (in contrast to NFS).

A delayed-write approach is used to handle file modification. When an application issues a write kernel call, the kernel simply writes the block into its cache and returns to the application. The block is not written through to the server's cache or the disk until it is ejected from the cache, or thirty seconds have elapsed since the block was last modified. Hence, a block written on a client machine will be written to the server's cache in at most thirty seconds, and will be written to the serve's disk after an additional thirty seconds. This policy results in better performance on the account of a possibility of losing recent changes in a crash.

Sprite employs a version number scheme to enforce consistency of shared files. The version number of a file is incremented whenever a file is opened in write mode. When a client opens a file, it obtains from the server the file's current version number, which the client compares to the version number associated with the

[4] Read-ahead is useful in case of sequential read. Blocks are read from the server disk and buffered on both server and client sides, before they are actually needed, to speed-up the read.

cached blocks for that file. If they 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 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. All subsequent reads and writes go through the server, which serializes the accesses. Caching is disabled on a file basis, and the disablements affects only clients with open files. Obviously, a substantial degradation of performance occurs when caching is disabled. A non-cachable 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 the file 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.

9.4 Summary

Since Sprite is currently under development its design still evolves. However, some definite characteristics of the system are evident already:

- *Extensive use of caching.* Sprite is inspired by the vision of diskless workstations with huge physical memories, and accordingly relies heavily on caching. The current design is fragile due to the amount of the state 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 on block basis in main memory—performance.
- *Consistency semantics.* Sprite sacrifices even performance in order to emulate Unix semantics. This decision eliminates the possibility of whole file caching as done in Andrew and its benefits.
- *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 which allows dynamic changes in the tables.

10 The 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 modified version of Unix 4.2 BSD, running on Sun Workstation using an unreliable datagram protocol (UDP/IP protocol [Postel 80]) and Ethernet. The specification and the implementation are intertwined in the description; whenever a level of detail is needed we refer to the SunOS implementation and whenever the description is general enough it applies to the specification also.

The system is presented in three levels of detail. First (in Section 10.1), the external capabilities of NFS are described mainly from a user's point of view. Then, two service protocols which are the building blocks for the implementation are examined (Section 10.2). In Section 10.3, a description of the SunOS implementation is given.

10.1 NFS 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 (upon explicit request) in a trans-

parent 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, and not only with dedicated server machines. Consistent with the independence of a machine, is the critical observation that sharing of a remote file system affects only the client machine and no other machine. Therefore, is no notion of 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 has to carry out a *mount* operation first[5]. The semantics of the operation is that a remote directory is mounted over a directory of a local file system. Once the mount is completed, the mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the underlying directory. The local directory becomes the name of the root of the newly mounted directory. Specifying the remote directory as an argument for the mount operation is done in a non-transparent manner; the location (i.e. hostname) of the remote directory has to be provided. However, from then on, users on the client machine can access files in the remote directory in a totally transparent manner, as if it is local. See Figure 10.1 for an example of mount mechanism. In Figure 10.1(a), the independent file systems of three machines named client, server1 and server2 are shown. In Figure 10.1(b), the effects of the mount command

```
client# /usr/etc/mount server1:/usr/shared client:/usr/local
```

performed by the super-user of client are shown. Observe that any file within the `dir1` directory for instance, can be accessed using the prefix `/usr/shared/dir1` in client after the mount is complete. The directory `/usr/local` on that machine is not visible any more.

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 (version 4.0, May 1988 described in [SUN]), diskless workstations can even mount their own roots from servers.

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. However, a machine is affected only by the mounts it has invoked *itself*. By mounting a remote file system, it does not gain access to 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 2 we illustrate cascading mounts by continuing our previous example. The figure shows the result of mounting `server2:/dir2/dir3` over `client:/usr/shared/dir1` which is already remotely mounted from server1. Files within `dir3` can be accessed in client using the prefix `/usr/shared/dir3`.

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 home environment. This property is referred to as user mobility.

One of the design goals of NFS was to operate in a heterogeneous environment of different machines, operating systems and network architectures. The NFS specification is independent of these mediums 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 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.

10.2 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 RPC's that define their functionality. These RPC's are the building blocks used to implement transparent remote file access.

---

[5] Actually, only a super-user can invoke the mount system call.

10.2.1 The Mount Protocol

The mount protocol is used to establish the initial connection between a server and a client. The semantics of the mount operation were described in Section 10.1. In Sun's implementation each machine has a server process outside the kernel, performing the protocol functions.

A mount operation includes the name of the remote directory to be mounted and the name of the server machine storing it. The mount request is mapped to the corresponding RPC and forwarded to the mount server running on the specific server machine. The server maintains an *export list* (the `etc/exports` in Unix, which can be edited only by a super-user), which specifies local file systems that it exports for mounting, along with names of machines that are permitted to mount them. Recall that 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 which 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 that the server needs to distinguish an individual file it stores. In Unix terms the file handle consists of a file system identifier, and an inode 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, for instance notifying all clients that the server is going down. Adding and deleting an entry in this list is the only way that the server state is affected by the mount protocol.

In previous NFS versions, a diskless workstation depended on the ND (Network Disk) protocol which provides raw block I/O service from remote disks. The server disk used to be partitioned, and no sharing of root file systems was allowed. In the new NFS version, the ND protocol was eliminated and root file systems can be shared too.

Usually a system has some static mounting pre-configuration which is established at boot time (`/etc/fstab` in Unix); however this layout can be modified.

Besides the actual mount procedure, the mount protocol includes several other procedures like unmount, return export list etc.

The exact details of how a mount operation changes the client's view of the file system is explained in the implementation section; however, it is crucial to note that this operation changes only the client's view and does not affect the server side.

10.2.2 The 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.
- 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*. Servers do not maintain information about their clients from one access to another access. There are no parallels to Unix's open files table or file structures on the server side. Consequently, each request has to provide a full set of arguments including a unique file identifier, and an absolute offset inside the file for the appropriate operations. The resulting design is very robust since no special measures need to be tak-

en to recover a server after a crash. File operations need to be *idempotent* for this end.

Maintaining the clients list mentioned in Section 10.2.1 seems to violate the statelessness of the server. However, it 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, it 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 a RPC is that modified data (including indirection and status blocks) must be committed to the server's disk before returning results to the client. Also, it is guaranteed that a single NFS `write` procedure call is atomic, and not intermixed with other write calls to the same file.

The NFS protocol does not provide concurrency control mechanisms at all. It is guaranteed that a single NFS `write` is atomic; however, since a write system call may be broken up to a few RPC `writes` (because each NFS `write` or `read` call can contain up to 8192 bytes of data), two clients writing to the same remote file may get intermixed. 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.

10.3 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.

10.3.1 Architecture

The NFS architecture consists of three major layers and is schematically depicted in Figure 3. The first layer is the Unix file system interface based on the `open`, `read`, `write`, `close` calls and file descriptors.

The second layer is called *Virtual File System* (VFS) layer, and it serves two important functions:

- Separating file system generic operations from their implementation by defining a clean Virtual File System interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems mounted locally.
- The VFS is based on a file representation structure called a *vnode*, that contains a numerical designator for a file that is network-wide unique. (Recall that Unix inodes are unique only within a single file system). The kernel maintains one vnode structure for each active node (file or directory).

Thus, the VFS distinguishes local files from remote ones, and local files are further distinguished according to their file system types.

Similarly to a single-site Unix, the kernel maintains a table (`/etc/mtab` in Unix) recording the details of the mounts in which it took part as a client. Further, the vnodes for each directory which is mounted over are kept in core at all times and are marked, so that requests concerning that directories will be redirected to the corresponding mounted file systems via the mount table. Essentially, the vnode structures complemented by the mount table provide for every file a pointer to its parent file system, as well as to the file system over which it is mounted.

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. The layer implementing the NFS protocol is the bottom layer of the architecture and is called the NFS service layer.

As an illustration of the architecture let us trace how an operation on an already open remote file is handled (follow the example on Figure 10.3). 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 to 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 the fact 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, that provide a temporary substitute to a light-weigh-process facility.

10.3.2 Pathname Translation

Pathnames 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. Once a mount point was crossed, *every* component lookup causes a separate RPC to the sever. This expensive pathname traversal scheme is needed since each client has a unique layout of its logical name space, dictated by the mounts it performed. It would have been much more efficient to hand a server a pathname 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 that the stateless server is unaware of.

To make lookup faster, a directory name lookup cache in the clients side 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. However, a server cannot act as an intermediary between a client and another server. Instead, a client should establish direct server-client connection with the second server by directly mounting the desired directory. When a client has a cascading mount, more than one server can be involved in a pathname traversal. However, each component lookup is performed between the original client and some server. 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. This decision is based on several factors. First, performance degradation caused by intermediate connections. Second, complication to access control. Third, preventing cyclic service arrangements rather than detecting or avoiding them.

10.3.3 Remote Operations

With the exception of opening and closing files, there is almost 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 employed 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 RPC's 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 whether to fetch or revalidate the cached attributes. 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. Cached attributes are discarded after three seconds for files or thirty seconds for directories. Both read-ahead and delayed-write techniques are used between the server and the client [Sun 88]. (Earlier versions of NFS employed write-on-close [Sandberg et al. 85]). Clients do not free delayed-write blocks until the server confirms that the data is written to disk. In contrast to Sprite, delayed-write is retained even when a file open concurrently, in conflicting modes. Hence, Unix Semantics are not preserved.

Tuning the system for performance makes it very difficult to characterize the consistency semantics of NFS. New files created on a machine may not be visible elsewhere for thirty seconds. It is indeterminate whether writes to a file at one site are visible to other sites that have this 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 neither strict emulation of Unix Semantics, nor the Session Semantics of Andrew.

10.4 Summary

- *Logical name structure.* A fundamental observation is that every machine establishes its own view of the logical name structure, since mounts affect only clients and not servers. Different machines are free to mount different file systems, and even different parts of the same exported file system. 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. This view starts from the local root and descends downward to the leaf files, crossing mount points that tailor the fragments of the file system together. 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. There are no facilities to administrate and maintain the set of machines as a coherent system. Separate attention has to be paid to each one of them. The fact that parts of file systems can be mounted complicates matters even further, since taking a server off-line has the effect of making different parts of different file systems unavailable.
- *Network service versus distributed operating system.* NFS is a network service for sharing files rather than an integral component of a distributed operating system as [Tanenbaum and Van Renesse 1985] defines one. 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 for doing so 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 is not fetched *en masse*; instead, the remote site potentially participates in each read and write operation. NFS employs 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. Should a server or the network fail, it is only necessary that clients continue to attempt to complete NFS operations. Once caching was introduced various patches had to be invented to keep the cached data consistent, without making the server stateful.
- *Consistency 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 a commercially available software with very reasonable performance and is perceived as an adequate successor of Unix.

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 over 5,000 workstations. Since 1983, Andrew has gone through design, prototype implementation, and refinement phases. We concentrate on the current design and implementation which is still a forerunner for future versions. It is interesting to examine how the design evolved from the prototype to the current version. An excellent account on 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 are Sun2's with local disks of 65M bytes, and the file servers are Sun2's or Vax-750, each with 2 or 3 disks

of 400M bytes.

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 of the Andrew file system

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 homogenous, 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 (see Figure 12.1). 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, distinct for each workstation, and contains system programs essential for autonomous operation and better performance, temporary files, and files that 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. 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* (see Figure 2). 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 offload work from the servers to the clients, in light of the common experience [Lazowska 1986] indicating that server's CPU is the system's bottle-neck. Following this heuristic, the key mechanism selected for remote file operations is *whole file caching*. Opening a file causes caching it, in its entirety, in the local disk. Reads and writes are directed to the cached copy without involving the servers at all. Under certain circumstances, the cached copy can be further used for forthcoming opens.

Entire file caching mechanism has a lot of merits which are described subsequently. However, this design cannot accommodate remote access to very large files (i.e., above few Megabytes). Thus, a separate design will have to address the issue of usage of large databases in the Andrew environment.

There are additional issues in Andrew's design that will not be discussed here and are just briefly pointed out:

- *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 his usual workstation would be some initial degraded performance due to the caching of files.
- *Security.* Special consideration was given to this issue. The Vice interface is considered as the boundary of trustworthiness since no user programs are executed on Vice machines. Authentication and secure transmission functions are provided as part of a connection-based communication package, based on the RPC paradigm. 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 The 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. Conceptually, volumes are glued together by a mechanism similar to Unix mount mechanism. However, 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 is 96 bits long and has 3 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 inodes of files in a single volume. The uniquifier allows reuse of vnode numbers, thereby keeping certain data structures compact. Fid's are location transparent; 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.

In order to balance the available disk space and utilization 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 a temporary forwarding information, so that the location database need not be updated synchronously. While the volume is being transferred the original server still may handle updates, which are shipped later to the new server. At some point the volume is briefly disabled to process the recent modifications, and 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 Consistency 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 semantics of file operations.

The operating system on each workstations 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 servers they came from when they are closed. Venus may contact Vice only when a file is opened or closed; reading and writing of individual bytes of a file are performed directly on the cached copy and bypass Venus. As a result writes at some sites are not visible immediately 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 always 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 it has been modified locally, updates it on the appropriate server. Thus, the only occasions in which Venus contacts Vice servers are on open of files that either are not in the cache or their callback has been revoked, and on closes of locally modified files.

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, it 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 link 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 which 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 Implementations

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 was described earlier. It has a mapping cache which associates volumes to server locations in order to avoid server interrogation for an already known volume location. If a volume is not present in this cache, Venus contacts any server that it has already a connection to, requests the location information, and enters it into the mapping cache. Unless Venus already has a connection to the server, it establishes a new connection. It then uses this connection to fetch the file or directory. Connection establishment is needed for authentication and security purposes. 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 inodes to avoid the expensive pathname translation routine (`namei`). Since the internal inode 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 (LRU) 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 `stat` system calls. The data cache is resident on the local disk, but 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 Light Weight Process Package (LWP) with non-preemptable scheduling to concurrently service many client requests. 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 lightweight processes to these connections. Instead, a pool of lightweight processes service client requests on all connections. The use of single, user level, server process allows to maintain in its address space caches of data structures needed for its operation. On the other hand, a single server process crash has a disastrous effect of paralyzing this particular server.

11.5 Summary

We revise 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 small and distinct local name space and can access the shared name space managed by the servers.
- *Scalability*. Undoubtedly Andrew is distinguished by this feature. 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 at all. The callback mechanism was invented to reduce the number of validity checks. Performing pathnames traversal by clients off-loads this burden from servers. The penalty for choosing this strategy and the corresponding designs includes the inability to handle large files, introducing a lot of state on the servers for the callback mechanism, and a different consistency semantics.

  The structuring of Andrew in clusters tries to take advantage of locality of reference, as clients are supposed to access their cluster server files most frequently. Also, this structuring lends itself easily to adding workstations and with some effort to adding complete clusters (the replicated location database has to be updated in the latter case).
- *Consistency semantics*. Andrew's semantics are simple and well defined (in contrast to NFS for instance, where effects of concurrent accesses are timing-dependent). However, they are not Unix semantics. Basically, Andrew's semantics ensure that file updates are visible across the network only after they are 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.

In [AN1] results of a thorough series of performance experimentation with Andrew is presented. The results confirm the current design predictions. That is, the desired effects on server CPU utilization, 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 has focused on small number of concepts and systems, and does not claim to exhaust the area of DFSs. Consequently, many aspects and many systems are omitted. In this Section we cite references that complement this paper.

A detailed survey of mainly centralized file servers is found in [Svobodova 84]. The emphasis there is on support of atomic transactions and not on location transparency and naming. A tutorial on distributed operating systems is presented in [Tanenbaum and Van Renesse 85]. There, a distributed operating system is defined and issues like communication primitives and protection are discussed. These two surveys include rich bibliography to a variety of earlier and current distributed systems.

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

- *Roe* [Ellis and Floyd 85; Floyd 89]. Roe presents a file as an abstraction hiding both replication and location details. Files are migrated to achieve balancing of system-wide disk storage allocation, and also as a remote access method. Consistency of replicated files is obtained by weighted voting algorithm [Gifford 1979].
- *Eden* [Jessop et al. 1982; Almes et al. 1983; Black 1985]. 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 [Levy 84] approaches. 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, migrated, are named in a location independent manner and can support atomic transactions. Emerald is an object-based language and system for construction of distributed programs, developed at the same university, and it pursues the same object-oriented approach of Eden [Jul et al. 1987].
- *Stork* [Paris and Tichy 1983]. Stork is an experimental file system that was 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 his files can be located. It is emphasized that file access patterns must exhibit locality in order to make file migration an attractive remote access method.
- *Ibis* [Tichy and Ruan 84]. This system is the successor of Stork. Ibis 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, structured, 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 it involves all directories containing the names of the moved file.
- *Apollo Domain* [Leach et al. 82]. 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 similarly to other objects are named by UIDs. A UID is a network-wide unique, low-level, location independent name. No global state information is kept on object locations. Instead, a simple location algorithm, based on heuristics for guessing the object's location, is employed. A unique feature of Domain is the way objects are accessed once located. Objects are mapped onto clients' address spaces and accessed via virtual memory paging. Only the needed portions of the objects are actually fetched over the network. Objects are organized in hierarchical directories that associate them with location independent textual names.

13 CONCLUSIONS

In this paper we have presented the basic concepts underlying the design of a distributed file system, and surveyed few of the most prominent systems. A comparison of the system is presented in Table 1.

A crucial observation, based on the assessment of current 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 the network is a strategy that disregards the unique characteristics of a DFS.

Supporting this claim is the observation that a loose notion of consistency semantic 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.

Another area in which departing from conventional system design is essential is the server process architecture. There is a wide consensus that some form of light weight processes are more suitable than traditional processes, for handling efficiently high loads of service requests.

It is difficult to present concrete guidelines in the context of fault tolerance and scalability, mainly because there is still 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 was 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 prove to be two sound scalability strategies. Clusters should be as autonomous as possible and should serve as a modular building block for an expandable system.

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:

- *Big main memories.* As main memories become larger and cheaper, main-memory caching (as exemplified in Sprite) would become more attractive. The rewards in terms of performance can be exceptional.
- *Optical Disks.* Write-once optical disks [Fujitani 84]] are already available. Their key features are very big density, slow access time, and non-erasable writing. For the time being, this medium is bound to become on-line tertiary storage and replace tape devices. Rewritable optical disks are under research and development.
- *Non-volatile RAMs.* Battery-backed memories can survive power outage, thereby enhancing the reliability of main-memories caches. A big 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 operating system crash [Ousterhout 89].

ACKNOWLEDGMENTS

This work was partially supported by NSF grant IRI 8805215.

REFERENCES

Almes, G. T., Black, A. P. , Lazowska, E. D., Noe, J. D. 1983. The Eden System: A Technical Review. IEEE Transactions on Software Engineering. Vol. 11, No. 1 (January) , pp. 43-59.

Barak, A., Litman, A. 1985. MOS: a Multicomputer Distributed Operating System. Software — Practice and Experience Vol. 15, No. 8 (August), pp. 725-737.

Barak, A., Malki, D., Wheeler, R. 1986. AFS, BFS, CFS ... or Distributed File Systems for UNIX. EUUG Autumn '86, (September 22-24, Manchester UK), pp. 461-472.

Barak, A., Paradise, O. G. 1986. Mos - Scaling up Unix. Proceedings of Usenix 1986 Summer Conference. pp. 414-418.

Barak, A., Kornatzky, Y. 1987. Design Principles of Operating Systems for Large Scale Multicomputers. IBM Research Division, T. J. Watson Research, RC 13220 (#59114).

Bernstein, P., A., Hadzilacos, V., Goodman, N., 1987. Concurrency Control and Recovery an Database Systems. Addison-Wes-

ley, Reading, Mass.

Black, A. P. 1985. Supporting Distributed Applications: Experience with Eden. Proceedings of the 10th Symposium on Operating Systems Principles, (Orcas Island, Washington, December 1-4). ACM, New York, pp. 181-193.

Birrel, A. D., Levin, R., Needham, R. M., Schroeder, M. D. 1982. Grapevine: An Exercise in Distributed Computing. Communications of the ACM, Vol. 25, No. 4 (April), pp. 260-274.

Birrel, A. D., Nelson, B. J. 1984. Implementing Remote Procedure calls. ACM Transactions on Computer Systems, Vol. 2, No. 1 (February).

Brownbridge, D. R., Marshall, L. F., Randell, B. 1982. The Newcastle Connection or Unixes of the World Unite! Software — Practice and Experience, Vol. 12, No. 12 (December), pp. 1147-1162.

Davidson, S. B., Garcia-Molina, H., Skeen, D. 1985. Consistency in Partitioned Networks. ACM Computing Surveys Vol. 17, No. 3 (September), pp. 341 - 370.

Dion, J. 1980. The Cambridge File Server. ACM SIGOPS Operating Systems Review Vol. 14, No. 4 (October), pp. 26 -35.

Douglis, F., Ousterhout, J. K. 1989. Beating the I/O Bottleneck. ACM SIGOPS Operating Systems Review, Vol 23, No. 1 (January), pp. 11-28.

Ellis, C. S., Floyd, R. A. 1983. The ROE File System. 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. Technical Report 272, Department of Computer Science, University of Rochester, January 1989.

Fujitani, L., Laser Optical Disks: The Coming revolution in On-line Storage. Communication of the ACM Vol. 27, No. 6 (June).

Gifford, D., Weighted Voting for Replicated Data. Proceedings of the 7th Symposium on Operating Systems Principles (December). ACM, New York, pp. 150-159.

Hill, M., et al. 1986. Design Decisions in Spur. IEEE Computer, Vol. 19, No. 11 (November), pp. 8-22.

Hoare, C. A. R. 1974. Monitors: An Operating System Structuring Concept. Communication of the ACM, Vol. 17, No. 10 (October).

Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N. 1988. Scale and Performance in a Distributed File System. ACM Transactions on Computer Systems, Vol. 6, No. 1 (February), pp. 55-81.

Jessop, W. H. et. al. 1982. The Eden Transaction Based File System. 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., Black, A. 1987. Fine Grain Mobility in the Emerald system (extended abstract). Proceedings of the 11th Symposium on Operating Systems Principles, (Austin Texas, November).

Kepecs, J. 1985. Light Weight Processes for Unix Implementation and Applications. Proceedings of Usenix 1985 Summer Conference.

Lampson, B. W., eds. 1981. Distributed Systemσ — Architecture and Implementation. Springer-Verlag.

Lazowska, E. D., Levy, H. M., Almes, G. T. 1981. The Architecture of the Eden System. Proceedings of the 8th Symposium on Operating Systems Principles (December). ACM, New York,, pp. 148-159.

Lazowska, E. D., Zahrojan, J., Cheriton, D., Zwaenepoel, W. 1986. File Access Performance of Diskless Workstations, ACM Transactions on Computer Systems, Vol. 4, No. 3 (August), pp. 238 - 268.

Leach, P. J., Stump B. L., Hamilton, J. A., Levine, P. H. 1982. UID's as internal names in a distributed file system. Proceedings of the 1st Symposium on Principles of Distributed Computing (Ottawa, Ontario, Canada, August 18 - 20). ACM, New York, pp. 34 - 41.

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

McKusic, M. K., Joy, W. N., Leffer, S. J., Fabry, R. S. 1984. A Fast File System for Unix. ACM Transactions on Computer Systems Vol. 2 No. 3 (August), pp. 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., et al. 1986. Andrew: a Distributed Personal Computing Environment. Communication of the ACM, Vol. 29, No. 3 (March), pp. 184-201.

Needham, R. M., Herbert, A. J. 1982. The Cambridge Distributed Computing System, Addison Wesley, Reading, Mass., 1982.

Nelson, M., Welch, B., Ousterhout, J. K. 1988. Caching in the Sprite Network File System. ACM Transactions on Computer Systems, Vol. 6, No. 1 (February).

Ousterhout J. K., et al. 1985. A Trace-Driven Analysis of the Unix 4.2 BSD File System. Proceedings of the 10th Symposium on Operating Systems Principles, (Orcas Island, Washington, December 1-4). ACM, New York, pp. 15-24.

Ousterhout, J. K., Cherenson, A. R., Douglis, F., Nelson, M. N., Welch, B. B. 1988. The Sprite Network Operating System. IEEE Computer, Vol. 21, No. 2 (February), pp. 23-36.

Paris, J. F., Tichy, W. F. 1983. Stork: An Experimental Migrating File System for Computer Networks, Proceedings IEEE INFCOM. IEEE, New York, pp. 168-175.

Popek, G., Walker, B. eds. 1985. The LOCUS Distributed System Architecture. MIT Press, Cambridge Mass., 1985.

Postel, J. 1980. User Datagram Protocol, RFC-768, Network Information Center, SRI, August 1980.

Quarterman, J. S., Silberschatz, A., Peterson, J. L. 1985. 4.2 and 4.3 BSD as Examples of the UNIX System. ACM Computing Surveys Vol. 17, No. 4 (December).

Randell, B. 1983. Recursively Structured Distributed Computing Systems. Proceedings of the 3rd Symposium on Reliability in Distributed Software and Database Systems, (Clearwater Beach, Florida, October 17-19) IEEE, New York, pp. 3-11.

Ritchie, D. M., Thompson, K. 1974. The UNIX Time Sharing System. Communication of the ACM, Vol. 19, No. 7 (July), pp. 365-375.

Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyone, B.. 1985. Design and Implementation of the Sun Network File System. Proceedings of Usenix 1985 Summer Conference, (June), pp. 119-130.

Satyanarayanan, M., Howard, J. H., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., Spector, A. Z., West, M. J. 1985. ITC Distributed File System: Principles and Design. Proceedings of the 10th Symposium on Operating Systems Principles, (Orcas Island, washington, December 1-4). ACM, New York, pp. 35-50.

Schroeder, M. D., Birrel, A. D., Needham, R. M. 1984. Experience with Grapevine: The Growth of a Distributed System. ACM Transactions on Computer Systems, Vol. 2, No. 1 (February), pp. 3-23.

Schroeder, M. D., Gifford, D. K., Needham, R. M. 1985. A Caching File System for a Programmer's Workstation. Symposium on Operating Systems Principles, (Orcas Island, Washington, December 1-4). ACM, New York, pp. 25-32.

Sheltzer, A. B., Popek, G. J. 1986. Internet Locus: Extending transparency to an Internet Environment, IEEE Transactions on Software Engineering. Vol. SE-12, No. 11 (November).

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 Computing Surveys Vol. 16, No. 4 (December), pp. 353 - 398.

Tanenbaum, A. S., Van Renesse R. 1985. Distributed Operating Systems. ACM Computing Surveys, Vol. 17, No. 4 (December), pp. 419-470.

Tevanian, A., Rashid, R., Golub, D., black, D., Cooper, E., Young, M. 1987. Mach Threads and the Unix Kernel: The Battle for Control. Proceedings of Usenix 1987 Summer Conference.

Tichy, W. F., Ruan, Z. 1984. Towards a Distributed File System. 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, Operating Systems Review. Vol. 17, No. 5 (October), pp. 49-70.

Weinstein, M. J., Page, T. W. Jr., B. K. Livezey, G. J. Popek. 1985. Transactions and Synchronization in a Distributed Operating System. Proceedings of the 10th Symposium on Operating Systems Principles, (Orcas Island, washington, December 1-4). ACM, New York.

Welch, B. 1986. The Sprite Remote Procedure Call system. Technical report UCB/CSD 86/302, Computer Science Division (EECS), University of California, Berkeley.

Welch, B., Ousterhout, J. K. 1986. Prefix Tables: a Simple Mechanism for Locating Files in a distributed System. Proceedings of the 6th Conference on Distributed Computing Systems (May). IEEE, New York, pp. 184-189.

Table 1

	Unix United	Locus	Sprite	NFS	Andrew
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.	Designed for an environment consisting of diskless workstations with huge main memories interconnected by a LAN.	A network service so that independent workstations would be able to share remote files transparently.	Designed as the sharing mechanism of a large-scale system for a university campus.
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.	Single Unix tree, hiding location.	Each machine has its own view of the global name space.	Private name spaces and one Unix tree for the shared name space. The shared tree descends from each local name space.
Component Unit	Entire Unix hierarchy.	A logical filegroup (Unix file system)	A domain (Unix file system).	A directory within an exported file system can be remotely mounted.	A volume (typically, all files of a single user).
User Mobility	Not supported.	Supported.	Probably supported.	Potential for support exists; demands certain configuration.	Fully supported.
Client-Server	Each machine can be both.	A triple: US,SS,CSS. Every filegroup has a CSS which selects SS and synchronizes accesses. Once a file is open, direct US-SS protocol.	Typically, clients are diskless, and servers are machines with disks.	Every machine can be both. Direct client-server relationship is enforced.	Clustering: Dedicated servers per cluster.
Remote access method	Emulation of conventional Unix across the network	Once a file is open, accesses are served by caching.	Block caching in main memory. In case of concurrent writes, switch to Remote Service.	Remote Service mixed with block caching for service.	Whole file caching in local disks.
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.	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.	Block caching similar to Unix buffering. Client checks validity of cached data on each open. Delayed-write policy	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.
Consist. Seman.		Complete Unix Semantics, including sharing of file offset.	Unix Semantics.	Not Unix Semantics, timing-dependent semantics.	Session Semantics.
Pathname traversal	The pathname translation request is forwarded from machine to machine.	US reads each directory and performs lookup itself. Given a filegroup number, the CSS is found in the replicated at all machines logical mount table. The CSS picks SS.	Prefix tables mechanism. Inside a domain, lookup is done by server.	Lookups are done remotely for each pathname component, but all of them are initiated from the client. A lookup cache for speedup.	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.
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.	Broadcast protocol supports dynamic reassignment of domains to servers.	Mount/unmount can be done dynamically by super-user for each machine, and change its directory hierarchy.	Volume migration is supported.
Availability	Not dealt with	Availability of a file means that 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.	If a server of a file is available, the file is available regardless of the state of other servers (along the pathname).	In case of cascading mount, each server along the mount chain has to be available for a file to be available.	A client has to have a connection to a server, and each pathname component must be available.
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.	No guarantees because of the delayed write policy. Stateful service.	Complete stateless service. Idempotent operations.	Not dealt with fully yet. Stateful service.
Scalability issues	Recursive structuring.	Replicated mount table on each site and CSS for a filegroup are major problems.	The fact that broadcast is relied upon and server involvement in operations might be a problem.	Not intended for very large scale systems.	Reducing server load and clustering are the main strategy. Replicated location database might be a problem.

Table 1, continued

Implementation Strategy, Architecture	Unix kernel kept intact. Connection layer is a library that 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.	New kernel based on multi-threading, intended for multi-processor workstation.	3 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.	Augmenting Unix kernel with user-level processes: Venus at each client, and a single server process on each server employing non-preemptable LWP's. Structured, low-level, location-independent file identifiers are used.
Networking	Suitable for arbitrary internetwork topology. No relationship of topology and name space. Static routing.	LAN	LAN	LAN	Cluster structure, with a router per cluster. All communication is based on high bandwidth LAN technology.
Communication Protocol	RPC	Specialized low level protocols for each operation.	RPC on top of special purpose network protocol.	RPC, and XDR on top of UDP/IP (unreliable datagram).	RPC on top of datagram protocol. Whole file transfer as a side effect.
Special features		Replication (primary copy). Atomic update by shadow paging.	Regular files used as swapping area. Interaction between file system and virtual memory system.	Stateless service.	Authentication and encryption built into the communication protocol. Access list mechanism for protection. Limited read-only replication.
Main advantage	Original Unix kernel. Internetworking capabilities.	Performance, because of kernel implementation. Fault tolerance, due to replication, atomic update and other features. Unix compatibility.	Performance, due to main memory caching.	Fault tolerance, because of stateless protocol. Implementation-independent protocols, ideal for heterogeneous environment.	Ability to scale up gracefully. Clear and simple consistency semantics.
Main disadvantage	Not fully transparent naming.	Complicated design and large kernel Unscalable features. Complex recovery due to maintained state.	Questionable scalability. Not much in terms of fault tolerance.	Unclear semantics. Performance improvements obscure clean design.	Fault tolerance issues, due to maintained state.

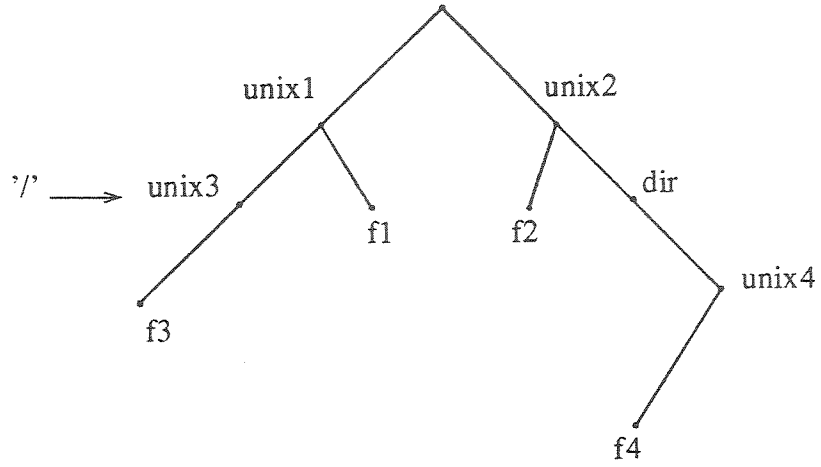


Figure 7.1

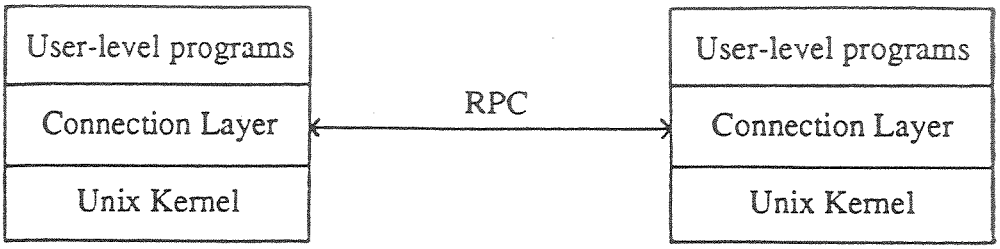


Figure 7.2 A schematic view of Unix United Architecture

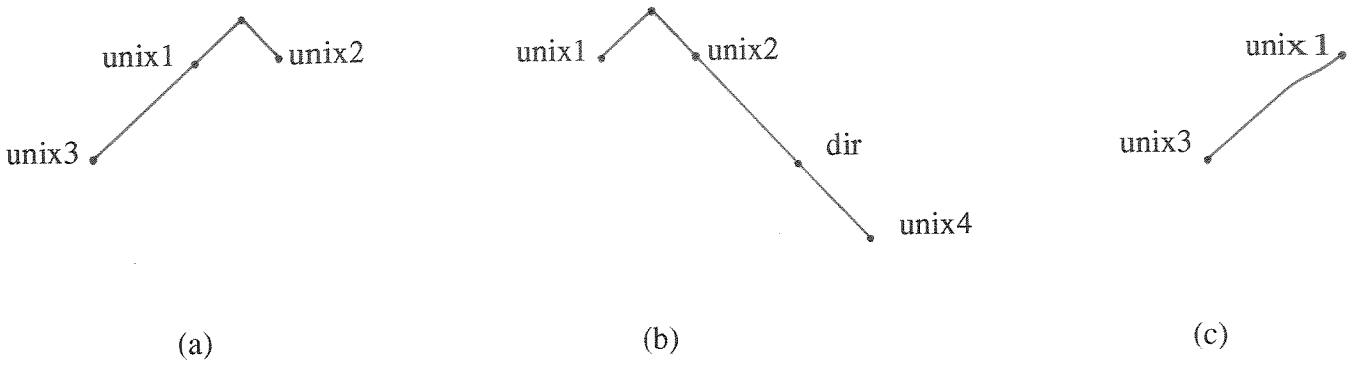
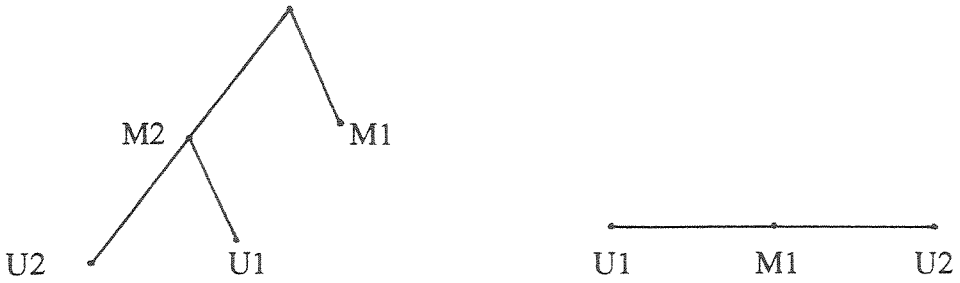


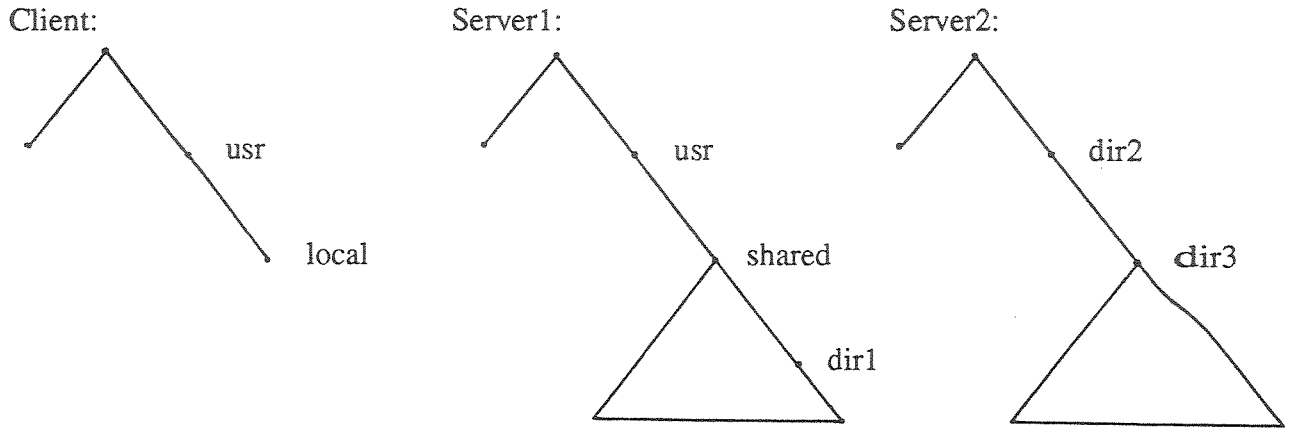
Figure 7.3



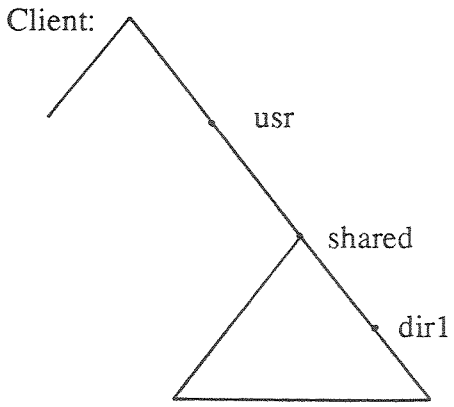
(a) Name structure

(b) Partial topology

Figure 7.4



(a)



(b)

Figure 10.1

Client:

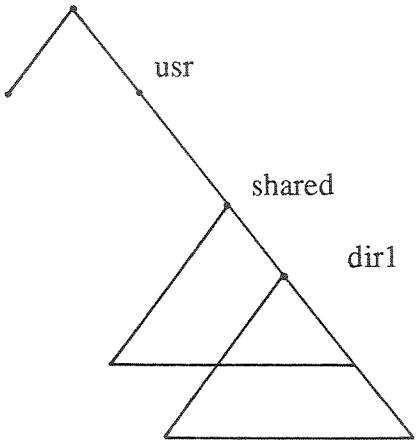


Figure 10.2

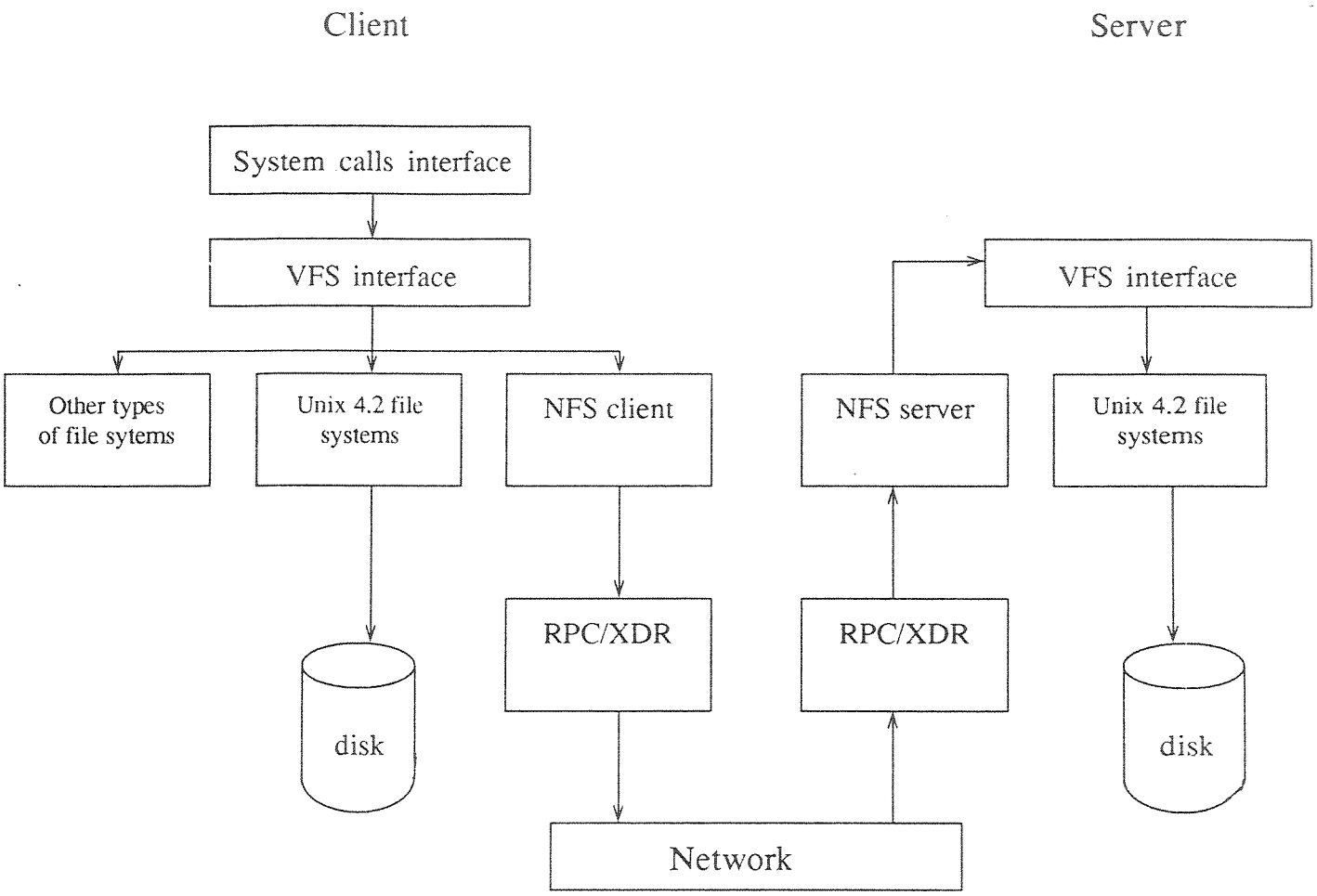


Figure 10.3 A Schematic view of the NFS architecture.