and data. Any centralized entity, be it a central controller or a central data repository, introduces both a severe point of failure and a performance bottleneck. Therefore, a scalable and fault-tolerant DFS should have multiple and independent servers controlling multiple and independent storage devices.

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

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

2. NAMING AND TRANSPARENCY

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

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

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

2.1 Location Transparency and Independence

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

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

Both definitions are relative to the discussed level of naming, since files have different names at different levels (i.e., user-level textual names, and system-level numerical identifiers). A location-independent naming scheme is a dynamic mapping, since it can map the same file name to different locations at two different instances of time. Therefore, location independence is a stronger property than location transparency. Location independence is often referred to as *file migration* or *file mobility*. When referring to file migration or mobility, one implicitly assumes

that the movement of files is totally transparent to users. That is, files are migrated by the system without the users being aware of it.

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

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

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

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

2.2 Naming Schemes

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

The second approach, popularized by Sun's NFS, provides means for individual

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

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

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

2.3 Implementation Techniques

This section reviews commonly used techniques related to naming.

2.3.1 Pathname Translation

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

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

component unit	server
cu1	machine1
cu2	machine2
cu3	machine3

Location Table

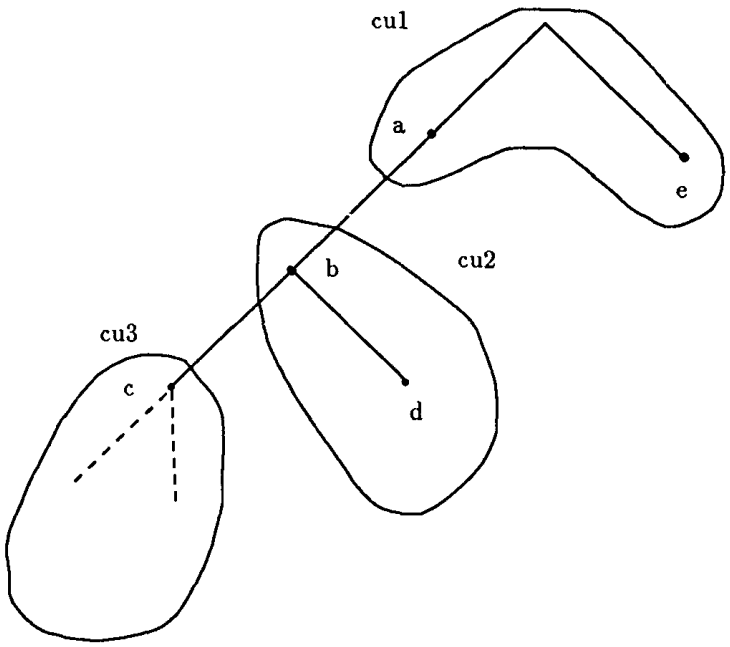

Figure 1. Lookup example.

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

2.3.2 Structured Identifiers

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

To enhance the availability of the crucial name to location mapping information, methods such as replicating it or caching parts of it locally by clients are used. As was noted, location independence means that the mapping changes in time and, hence, replicating the mapping makes updating the information consistently a complicated matter. Structured identifiers are location independent; they do not mention

servers' locations at all. Hence, these identifiers can be replicated and cached freely without being invalidated by migration of component units. A smaller, second level of mapping that maps component units to locations is the only information that does change when files migrate. The usage of the techniques of aggregation of files into component units and lower-level, location-independent file identifiers is exemplified in Andrew (Section 11) and Locus (Section 8).

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

2.3.3 Hints

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

2.3.4 Mount Mechanism

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

3. SEMANTICS OF SHARING

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

modifications of data by a client are observable, if at all, by remote clients.

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

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

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

3.1 UNIX Semantics

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

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

3.2 Session Semantics

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

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

3.3 Immutable Shared Files Semantics

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

3.4 Transaction-Like Semantics

Identifying a file session with a transaction yields the following, familiar semantics: The effects of file sessions on a file and

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

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

4. REMOTE-ACCESS METHODS

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

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

It should be realized that there is a direct analogy between disk access methods in conventional file systems and remote access methods in DFSs. A pure remote service method is analogous to performing a disk access for each and every access request. Similarly, a caching scheme in a DFS is an extension of caching or buffering techniques in conventional file systems (e.g., buffering block I/O in UNIX [McKusick et al. 1984]). In conventional file systems, the rationale behind caching is to reduce disk I/O, whereas in DFSs the goal is to reduce

network traffic. For these reasons, a pure remote service method is not practical. Implementations must incorporate some form of caching for performance enhancement. Many implementations can be thought of as a hybrid of caching and remote service. In Locus and NFS, for instance, the implementation is based on remote service but is augmented with caching for performance (see Sections 8.3, 8.4, and 9.3.3). On the other hand, Sprite's implementation is based on caching, but under certain circumstances a remote service method is adopted (see Section 10.3). Thus, when we evaluate the two methods we actually evaluate to what degree one method should be emphasized over the other.

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

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

4.1 Designing a Caching Scheme

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

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

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

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

4.1.1 Cache Unit Size

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

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

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

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

4.1.2 Cache Location

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

4.1.3 Modification Policy

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

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

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

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

Yet another variation on delayed-write, called *write-on-close*, is to write data back to the server when the file is closed. In cases of files open for very short periods or rarely modified, this policy does not signif-

icantly reduce network traffic. In addition, the write-on-close policy requires the closing process to delay while the file is written through, which reduces the performance advantages of delayed-writes. The performance advantages of this policy over delayed-write with more frequent flushing are apparent for files that are both open for long periods and modified frequently.

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

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

4.1.4 Cache Validation

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

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

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

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

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

4.2 Cache Consistency

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

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

4.3 Comparison of Caching and Remote Service

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

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

each remote access is handled across the network. The penalty in network traffic, server load, and performance is obvious.

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

5. FAULT TOLERANCE ISSUES

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

5.1 Stateful Versus Stateless Service

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

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

The advantage of stateful service is performance. File information is cached in

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

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

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

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

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

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

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

5.2 Improving Availability

Svobodova [1984] defines two file properties in the context of fault tolerance: "A file is *recoverable* if is possible to revert it to an earlier, consistent state when an operation on the file fails or is aborted by the client. A file is called *robust* if it is guaranteed to survive crashes of the storage device and decays of the storage medium." A robust

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

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

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

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

9. SUN NETWORK FILE SYSTEM

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

The system is presented in three levels of detail. First (in Section 9.1), an overview

is given. Then, two service protocols that are the building blocks for the implementation are examined (Section 9.2). Finally (in Section 9.3), a description of the SunOS implementation is given.

9.1 Overview

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

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

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

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

9.2 NFS Services

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

9.2.1 Mount Protocol

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

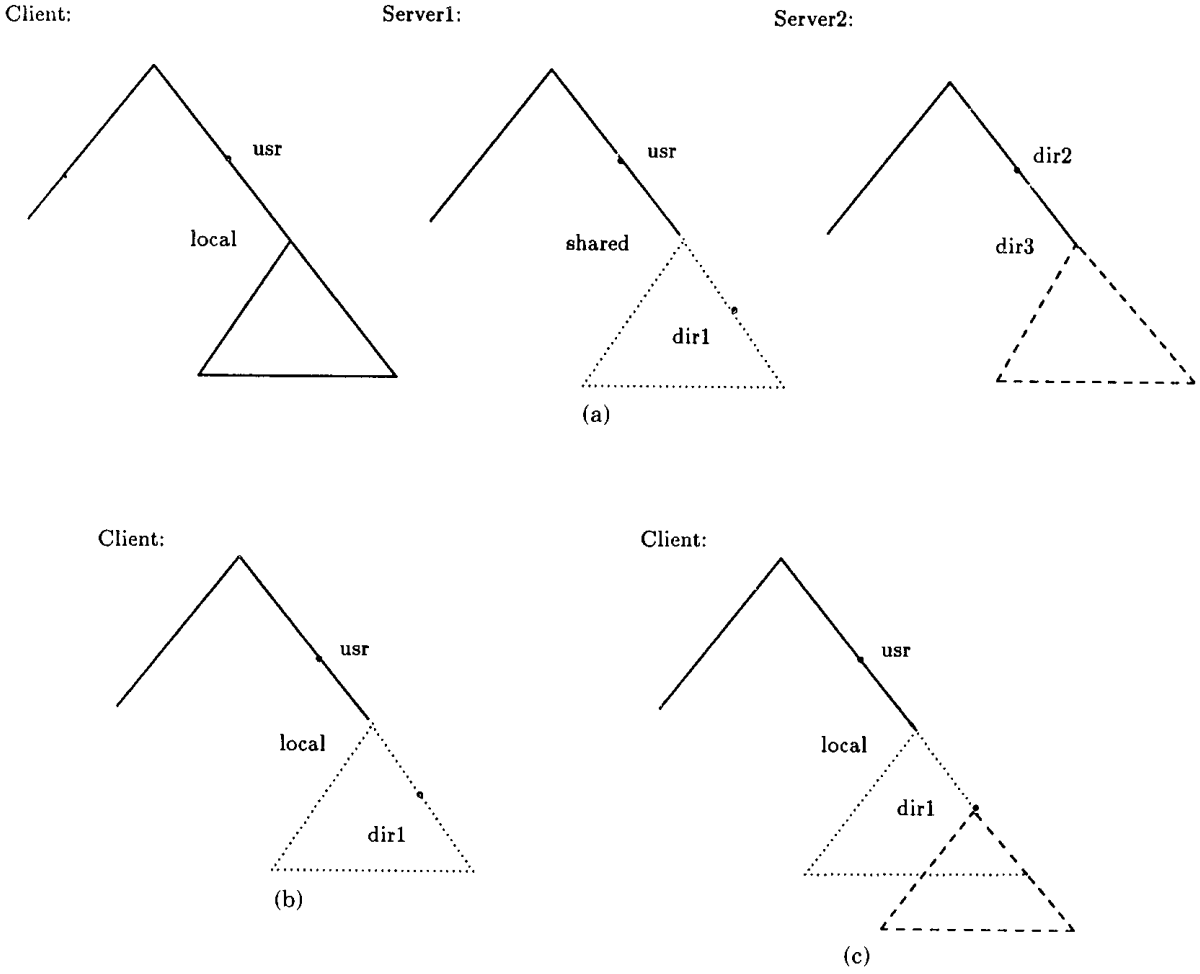

Figure 5. NFS joins independent file systems (a), by mounts (b), and cascading mounts (c).

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

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

The mount protocol is used to establish the initial connection between a server and a client. The server maintains an export list (the **/etc/exports** in UNIX) that specifies the local file systems it exports for mounting, along with names of machines permitted to mount them. Any directory within an exported file system can be remotely mounted by an accredited machine. Hence, a component unit is such a directory. When the server receives a mount

request that conforms to its export list, it returns to the client a file handle that is the key for further accesses to files within the mounted file system. The file handle contains all the information the server needs to distinguish individual files it stores. In UNIX terms, the file handle consists of a file system identifier and an i-node number to identify the exact mounted directory within the exported file system.

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

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

9.2.2 NFS Protocol

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

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

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

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

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

9.3 Implementation

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

9.3.1 Architecture

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

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

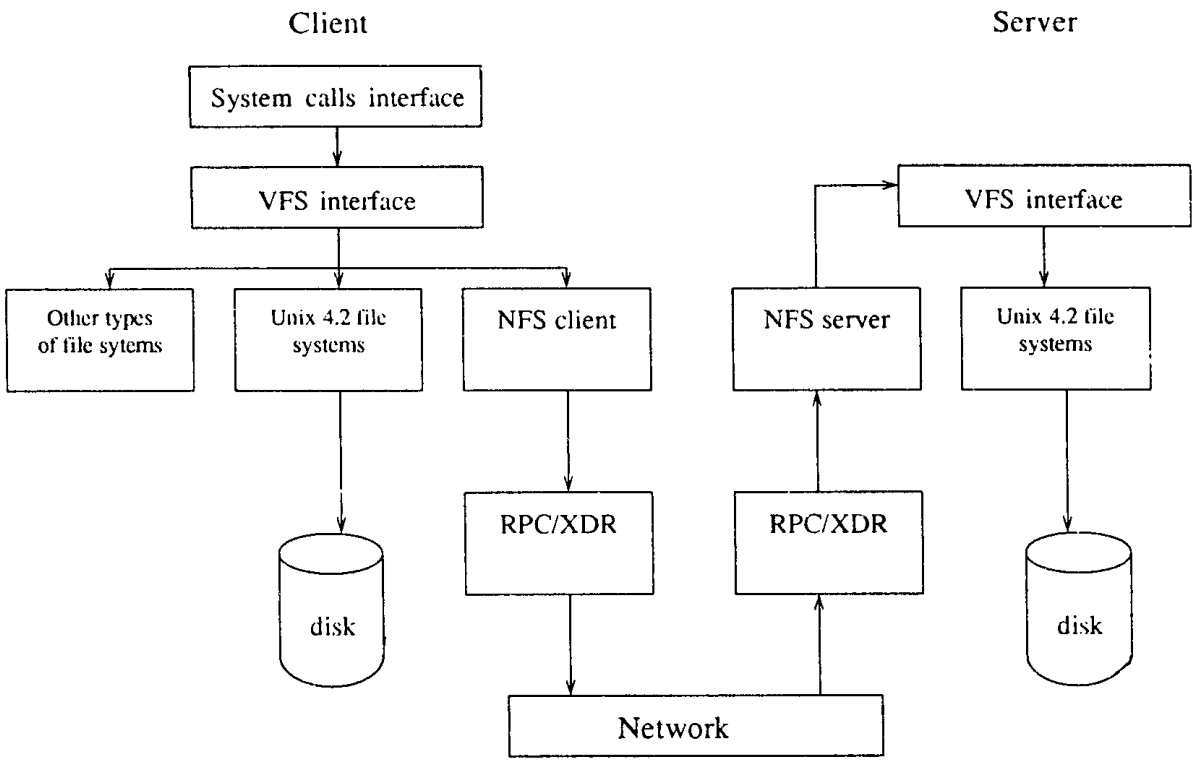

Figure 6. Schematic view of the NFS architecture.

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

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

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

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

9.3.2 *Pathname Translation*

Pathname translation is done by breaking the path into component names and doing a separate NFS lookup call for every pair of component name and directory vnode. Thus, lookups are performed remotely by the server. Once a mount point is crossed,

every component lookup causes a separate RPC to the server. This expensive pathname traversal scheme is needed, since each client has a unique layout of its logical name space, dictated by the mounts if performed. It would have been much more efficient to pass a pathname to a server and receive a target vnode once a mount point was encountered. But at any point there can be another mount point for the particular client of which the stateless server is unaware.

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

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

9.3.3 Caching and Consistency

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

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

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

9.4 Summary

- *Logical name structure.* A fundamental observation is that every machine establishes its own view of the logical name structure. There is no notion of global name hierarchy. Each machine has its own root serving as a private and absolute point of reference for its own view of the

name structure. Selective mounting of parts of file systems upon explicit request allows each machine to obtain its unique view of the global file system. As a result, users enjoy some degree of independence, flexibility, and privacy. It seems that the penalty paid for this flexibility is administrative complexity.

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

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

10. SPRITE

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

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

10.1 Overview

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

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

In spite of its functional similarity to UNIX, the Sprite kernel was developed from scratch. Oriented toward multiprocessing, the kernel is multithreaded. Synchronization between the multiple threads