Techniques for Building Highly Available Distributed File Systems

Carl D. Tait

13 March 1990

CUCS-497-89

Columbia University
Department of Computer Science
New York, NY 10027

ABSTRACT

This paper analyzes recent research in the field of distributed file systems, with a particular emphasis on the problem of high availability. Several of the techniques involved in building such a system are discussed individually: naming, replication, multiple versions, caching, stashing, and logging. These techniques range from extensions of ideas used in centralized file systems, through new notions already in use, to radical ideas that have not yet been implemented. A number of working and proposed systems are described in conjunction with the analysis of each technique. The paper concludes that a low degree of replication, a liberal use of client and server caching, and optimistic behavior in the face of network failure are all necessary to ensure high availability.

INTRODUCTION

At the heart of every operating system is its file system: the software that allows users to store and retrieve permanent data. In centralized operating systems, file systems are usually straightforward to implement. One simply keeps track of which disk blocks are free, which blocks belong to each file, which user owns each file, and who may access a specified file. Some sort of directory structure is usually supported so that files may be organized hierarchically. File systems based on this approach are used by everything from MS-DOS to UNIX with unqualified success.

In a distributed operating system, however, the design and implementation of a good file system are singularly difficult problems. Even the description "good" is not well-defined. There are numerous trade-offs to consider, and predictably, designers have come up with a number of radically different distributed file systems.

In a centralized environment, the model is simple: a single computer with a number of disk drives attached to it. But a distributed system involves multiple machines communicating via some sort of network, with disk drives connected to some or all of these machines; a number of systems support diskless workstations.

Ideally, a user should not need to know where any particular file is stored. The file system should be able to locate a given file and make it available to a user. This feature, **transparency**, is the primary difference between network and distributed file systems. In a network system, the user is aware of the multiple machines in the environment, and is responsible for knowing which files are stored on which machines. At best, subtrees of files from other machines can be mounted locally, but machine-to-machine movement of files will still be visible. Multiple machines are available for use, which may well provide more power, but much of the responsibility for managing this power falls on the user. A distributed system relieves the user of a large part of this burden: file access is machine-transparent.

One important goal of a distributed file system is high availability. Failure (or even scheduled maintenance) of a single machine or a single disk drive should not normally cause a file to become unavailable. As the world becomes increasingly dependent upon computers, high availability will become an increasingly important design objective.

In a centralized system, "mirrored disks" - drives with separate controllers that contain precisely the same data - are sometimes used to ensure availability of data. But with disk drives becoming extremely reliable, this method is too expensive for the typical office worker or scientist. One must purchase twice as many drives and controllers as are actually needed.

A similar problem dogs many distributed file systems. **Replication** - maintaining copies of each file on a number of machines - is a natural technique for increasing availability [Alsberg 76]. If one particular machine is down, a copy of the file can usually be retrieved from some other machine. Most of the time, however, the primary copy will be available, so the remaining copies will be largely unused. It is therefore unappealing for file system operations to expend a large amount of time or other resources in managing replicas of files. How, then, does one go about building a highly available distributed file system?

Every file system must have some scheme for mapping character-string file names to the files they represent, so this paper begins with a discussion of naming. This is followed by a consideration of various replication schemes. Absolute consistency of replicas is not always necessary - for example, a slightly outdated version of a text editor might well be acceptable - so systems that support multiple versions of files are examined next. Caching, stashing, and logging - three techniques that can be used to increase availability - are then discussed at length, followed by a series of conclusions.

NAMING

File name resolution is an important part of any distributed file system. The central concern of a naming scheme is to establish a unique, distribution-transparent name for each file. When a client provides such a file name, the system should be able to locate the file easily.

Name resolution needs to proceed swiftly, so the natural tendency is to make the table of name-to-file mappings widely available - at least as available as files - either through a name server or through replication. But both of these approaches have a devastating effect on scalability. A single name server is a performance bottleneck and a single failure point for the whole system, while a replicated name service introduces a new availability problem. In addition, for performance reasons, it should be possible for a file to migrate from one file server to another in a manner that is transparent to the user: server names should not be embedded in file names. Two proposed naming schemes are particularly noteworthy for how they address these issues.

Welch and Ousterhout [Welch 86] describe a name lookup mechanism known as **prefix tables**. This method is used in the Sprite system [Nelson 88], which will be discussed further in the section on caching. The distributed file system is seen as a single tree-structured hierarchy by users, but is actually divided into several **domains**. Each domain is a portion of the tree relegated to a particular server. A server may store more than one domain. Each client has a prefix table (typically incomplete) that maps file name prefixes to the servers on which the associated domains reside.

An example: suppose that a domain on server C is rooted at /chopin/etudes. A client attempting to locate the file /chopin/etudes/winter-wind would find an entry for the prefix /chopin/etudes in its prefix table along with the information that this domain is on server C. Even if a domain with root /chopin is located on server A, the client will still know that its file is on server C, since the longest applicable prefix in the table is always used.

The prefix table method does not require that server names be included in file names. Furthermore, entries in a table are regarded merely as hints. If a file is not where a table says it is, shorter prefixes are tried until the file is found. At each point, incorrect table entries are updated to reflect the new information. This gets around the update problem when a domain moves from one server to another: since table entries are just hints, they do not need to be absolutely correct at all times. File movement is therefore completely transparent to the user.

Finally, prefix tables can update themselves by exchanging information with other tables. If a client has no prefixes at all for a file (as will be the case initially), it broadcasts the file name to all servers. Relevant prefix/server mappings (with symbolic links already expanded) are sent back to the requesting client by all servers who have such mappings. In this way, prefix table information can be easily propagated around the network without the requirement that any two clients have precisely the same table. And because prefix tables contain only hints that need not be correct, this method avoids creating either an availability or a consistency problem.

A very different approach to naming is taken in the QuickSilver system [Cabrera 87], which employs the concept of **user-centered naming**. Instead of all users having a single view of a global name space, each user has a logically distinct name space in which resolution is performed. It is possible to think of a global name space in which files are identified by (user, local filename) pairs, but conceptually, each user has an individual tree of files. Internally, QuickSilver associates a unique file identifier (UFID) with each file in the system.

The primary reason for this unusual scheme is a concern with scalability. In a global naming system, the lookup required to determine that a file does not exist may take time proportional to the size of the network, which does not bode well for scalability. However, any given user will have a relatively small name space that can be searched exhaustively, if necessary, without severe detriment.

Files are added to a user’s name space through the use of file pointers called **links**, which may be either **soft**, **symbolic**, or **hard**. A soft link points to a UFID in another user’s name space. A client establishing such a link is not charged for the space to store the file. However, the file may be deleted by the user who owns it, so there is no assurance that the file will be permanently accessible.

Symbolic links are similar, except that the link is made to another user’s local file name instead of to the UFID of that file. This is useful for ensuring that one is always reading the latest version of a file such as a compiler or text editor. Finally, a hard link creates a logical copy of a file, ensuring that it will be available until explicitly deleted by the user creating the link. A user making such a link is charged for the space that the file occupies.

An example will show how name resolution proceeds. Suppose user Rachmaninoff has established a soft link to user Chopin’s file /preludes/c-minor (UFID = 1234), and a symbolic link to user Paganini’s file /caprices/number24. User Rachmaninoff’s local names for these files are /variations/chopin and /rhapsody/paganini. When the links are established, file location hints are placed in Rachmaninoff’s **user index**. The hints are created by employing a user-locating service called the **White Pages** to locate users Chopin and Paganini, and then searching their local name spaces for the desired files. Since binding of a symbolic name to a file must take place on every file access, the hint for the symbolic link is less specific than that for the soft link.

When Rachmaninoff attempts to access /variations/chopin, the hint is followed to reach the last known location of the file with UFID 1234. If this highly specific hint is wrong, QuickSilver next searches the disk index of the disk where file 1234 used to be located to see if it has moved somewhere else on the same disk. Next, other disks at that server site are examined. As a last resort, all server sites that store files belonging to Chopin - the owner of file 1234 - are searched. (The White Pages service provides a list of these sites if necessary.) In effect, QuickSilver works from specific hints to general ones, just as the prefix table method tries long prefixes before shorter ones.

One of QuickSilver’s strong points is that a user can be physically relocated without a change in user-centered view. Forwarding information can be left behind in the White Pages to make the move transparent. Since this information is just a hint, it is discarded after some period of time. After the hint is discarded, other users’ soft and symbolic links to the relocated user’s files become invalid, and attempts to access files through these links will fail. New links must be established explicitly.

User-centered naming is an interesting concept, and one that is particularly useful when scalability and user relocation are concerned. Most systems, however, continue to follow the UNIX tradition and assume a uniform, hierarchical view for all users. Future systems will undoubtedly continue to use hints, which are unquestionably helpful in any naming scheme. Incorrect hints can simply be discarded, or if desired, updated.

It is certainly possible to imagine situations where resolution of a name with N hierarchical components would require N computers to be up: every component might require a different machine for resolution. So naming is indeed tied to availability: if the name cannot be resolved due to unavailable information, the file cannot be accessed, even if it is on a machine that is up.

REPLICATION

Replication is one of the most important issues involved in the design of a distributed file system. Without replication, high availability is unattainable: if the server responsible for the single copy of a file crashes, the file becomes unavailable. Maintaining multiple copies of a file is inherently costly, however. Not only is more disk space required, but complex software is needed to keep replicas consistent. In addition, network traffic will almost certainly become heavier. Different systems have adopted widely varying solutions for balancing the trade-off between simple replica management and high availability.

Perhaps the knottiest problem that arises in a replication scheme is that of detecting and handling inconsistent copies of a file. If a file is modified while a server containing one of its replicas is unavailable, how is that server’s copy made consistent with the current version when the replica becomes accessible again? An uglier version of the problem arises when servers are separated due to network partition. In that case, different replicas may be updated differently, resulting in divergent versions that may be irreconcilable when the network is reconnected. Directories are even more problematic. Losing file updates may be tolerable under some circumstances, but losing directory updates can cause the loss of entire files.

Conceptually, the simplest form of replication is that performed on a file-by-file basis, as is done in the Roe system [Ellis 83]. Roe is designed to provide a single logical view of a heterogeneous local-area network. A **Roefile** is really a set of replicas, but the user sees only one logical entity.

In order to access a Roefile, the user issues a request to a **Transaction Coordinator**, which usually runs as a process on the user's machine. The coordinator treats the file access as an atomic transaction involving the set of replicas that the given Roefile comprises. The coordinator is also responsible for preserving enough information to recover in case of failure.

To translate the user's name for the Roefile into a set of file identifiers, the Transaction Coordinator makes use of the **Global Directory Subsystem**. This subsystem makes the required name translation, and obtains files from local file servers via each server's **Local Representative**. Because the network contains heterogeneous machines, these representatives are needed to maintain a uniform view of files for the directory subsystem across different local servers.

One of Roe's main goals is to enforce a high level of consistency among replicas. The **Weighted Voting** algorithm that the authors use meets their criteria nicely. Traditional quorum-based voting schemes using N replicas require that, if R copies are located for each read, at least N - R + 1 copies must be written on each write. This ensures that each read will see an up-to-date copy of the file. If fewer than R copies are available at read time, the file is considered inaccessible.

In weighted voting, each replica has both a timestamp and a voting strength associated with it. Quorum size is based not on the number of replicas, but on the number of votes. Highly reliable servers can be given many votes, which usually reduces the number of copies that must be read or written to form a quorum. The same principle applies: if there is a total of V votes, and a read quorum consists of R votes, then the total voting strength of copies written must be at least V - R + 1. Under this scheme, any quorum is guaranteed to contain an up-to-date replica of the given file. Furthermore, it is not necessary to worry about outdated copies, since only the replica with the most recent timestamp is read. When a file is written, all of its available replicas are given a new timestamp that is greater than the maximum of the old timestamps of those replicas.

Replicating directory information is somewhat harder. With individual files, Roe simply locks the file until the update is complete. But locking a directory for any period of time is clearly undesirable from the point of view of other users. Instead, Roe uses a callback scheme that requires users to **register** with directories that they want to access. When modifying a directory, a user attempts to update all replicas of that directory. If that is impossible, at least an appropriate write quorum must be gathered before writing, as with standard files. If some copies of the directory cannot be updated, all registered users are informed that they may no longer be using a current copy. Since a sufficient quorum was gathered before writing, however, it is easy for each user to obtain the most recent version of the directory when this occurs.

Another system that performs replication on a file-by-file basis is RNFS [Marzullo 88]. This system has high availability as a primary goal, unlike Roe, and can theoretically be implemented on top of any network file service. The authors chose Sun’s NFS (Network File System) [Sandberg 85, Kleiman 86] for several reasons, not the least of which was that it was readily available to them.

NFS has a stateless protocol: the server preserves no information between requests, so all relevant parameters must be included on each call. In addition, NFS read and write (but not control) operations are **idempotent**: calling a function arbitrarily many times with the same parameters has no more effect than calling it only once. RNFS clients can therefore recover from crashed servers simply by issuing their requests repeatedly until a response is received.

The high availability promised by RNFS is intended to be transparent to clients - the network file functions should not change visibly. To this end, RNFS interposes an **agent** process between a client and the actual file servers. Files are indeed replicated, but it is the agent’s task to hide the replication details from the client.

The scheme used to ensure consistency is an extreme one that is optimal for reads: read one, write all (a quorum-based scheme with $R = 1$). When a server becomes unavailable, the agent makes a note of that fact in the **replicated file-list**. If a subsequent write is issued to a replica on a failed server, that copy is marked as invalid in the replicated file-list. When the server comes back up, it acquires exclusive access to the file, and replaces its bad replica with a valid one. If no writes were issued to a file while it was unavailable, no replacement is necessary.

Of course agents themselves may fail, and special care must be taken in this case. To begin with, the replicated file-list must itself be replicated on all servers in a form called the stable file-list. A recovering agent uses this file-list to verify that all supposedly valid copies of a file are identical, since an agent may have crashed in the middle of a write.

In order to prevent the entire system from being inaccessible during an agent failure, agents themselves are replicated. Clients may direct their requests to any agent they choose. If an agent goes down, the client just starts using a different one. A token-passing mechanism is used to ensure that two agents never attempt to write to the same file simultaneously: an agent must hold the token in order to write to the file.

Unfortunately, there is a substantial performance penalty caused by interposing agents between clients and servers, and by writing multiple copies of a file. The system's designers believe that they can fine-tune RNFS so that it is "no more than 1.5 to 2 times slower than NFS." Preliminary tests indicate that client caching will markedly improve performance.

After observing the performance penalty that RNFS pays for availability, it is not hard to see why commercially available systems such as NFS and AT&T's RFS (Remote File Sharing) [Rifkin 86] do not have high-availability files as a design objective. Their primary goal is to make remote file access convenient and (relatively) transparent. If the server for a desired file is down, the file is simply inaccessible - replication is not supported. And if a file needs to be moved to a server that is closer to the client for performance reasons, the move will not be transparent to the user.

In the Andrew file system [Morris 86], which will be discussed in detail in a later section, replication is performed on groups of files known as **volumes** [Sidebotham 86]. A volume comprises a subtree of files - typically, all the files of a single user. Coda [Satyanarayanan 89], the newest file system for Andrew, uses **version vectors** to detect inconsistent replicas. A version vector has one component for each site where a replica is stored. When a file is written and closed, the corresponding version vectors at the sites where the file is written are incremented. Update counters within the vectors serve as a kind of timestamp that allow the system to detect which replicas are the newest, and to deal with inconsistency accordingly. This is an optimistic approach: inconsistency is not prevented, but it is always detected. By taking this position, Coda makes the implicit assumption that inconsistencies are relatively rare occurrences.

Hardware and system software support for **multicast** - the ability to send a message to a designated set of receivers - helps make replication in Coda efficient. Coda requires only about 10% longer to write three copies of a file than to write one.

Replication of volumes may be wasteful. It is likely that a user will need high availability for only a few of the files in a volume, so replicating all of them is unnecessary. Aggregating files into larger units does tend to make the overall organization simpler, however.

The LOCUS file system [Popek 85] uses a compromise between volume and single-file replication. There is a single tree-structured name space for all files in the system. **Filegroups** can be mounted onto the tree in a manner analogous to mounting file systems in UNIX. Filegroups correspond to the Andrew concept of volumes, but are replicated differently.

At every site where a given filegroup is to be replicated, a physical container called a **pack** is allocated for it. A pack can only contain files from one filegroup, but it need not contain all the files in the group. This allows individual files to have a high degree of replication without requiring that all files in the group be similarly replicated.

Furthermore, packs may vary in size, since a pack needs only to be large enough to hold the files replicated at that site. One pack is designated as the primary copy, and all members of a filegroup **must** be in that pack.

For each filegroup, one site is chosen as the **current synchronization site** (CSS). All requests to use a file in the group are directed to the CSS. Should the network become partitioned, a CSS will be created in each partition where the filegroup is used.

As its name implies, the CSS is responsible for ensuring harmonious interaction between clients attempting to use the same file. Tokens are used to synchronize reads and writes. There is also a **file offset token** that guarantees the correct offset within a file only to the client who holds the token.

A strong point of LOCUS is its ability to detect mutual inconsistency among replicas - even when some of the replicas have been renamed [Parker 83]. Using version vectors along with the concept of a unique, immutable **origin point** for each file - when and where the file was created - LOCUS applies simple graph analysis techniques to determine if inconsistent copies of a file exist after a network partition. In complex situations, timestamping approaches may detect conflicts that do not actually exist, but the LOCUS method has been proven not to suffer from this drawback.

File-by-file replication is by far the most commonly used scheme. Andrew and Coda group files into volumes for ease of replica management, but this method requires all files within a volume to be replicated at each storage site. LOCUS's pack-based replication is a happy compromise between the two extremes. Refer to the table at the end of this paper for a summary of the replication methods used by various systems.

MULTIPLE VERSIONS

The ability to maintain multiple versions of a file seems desirable, especially in a software development environment. Multiple versions can also serve as a watered-down form of replication in cases where loose consistency is acceptable. Traditionally, multiple versions of text files have been managed by programs such as RCS (Revision Control System) [Tichy 82], which store the versions of a file as sets of changes ("reverse deltas") appended to the current version. By applying a sequence of reverse deltas to the latest version, any previous version of the file can be reconstructed. Is it within the proper scope of a file system to perform version management automatically, or at least on request? Though not the primary focus of this paper, the question merits a short discussion.

One of the primary goals of the Cedar file system [Schroeder 85] is automatic support of multiple versions. This system is intended for use by programming teams sharing a collection of files. Related files can be grouped into **subsystems** specified by a user. These subsystems are accessed using a **DF file**, which contains a list of the files in a given subsystem.

Both data files and DF files are immutable. A modified file never replaces an old copy of the file. Instead, an entirely new version is created with a numerical suffix indicating the version number. Because a particular version of a file never changes, local caching is greatly simplified: a user need not worry about modifications to a remote copy of the file. In effect, the problem of maintaining cache consistency is translated into the requirement of keeping a version history. This may well be an improvement, if loose consistency is tolerable. To further simplify the scheme, Cedar caches only whole files.

Unfortunately, Cedar file names are required to include the server on which the associated file resides. A form of symbolic linking is available, however. Using a DF file, a user can form an **attachment** to a given subsystem. This is a form of lazy copying in which a file within the subsystem is not copied to the local workstation until it is actually needed. The attachment allows the user to specify much simpler names for files within a subsystem, ignoring server details after the attachment is made to a specific, immutable version of a file. This does mean, however, that a new attachment must be made if a new version of the file is created, and Cedar has tools to handle this with relative ease.

There is a problem that can arise when files are brought over to a local user. On the local machine, a file name is collapsed to a simple name: the prefix indicating the server is deleted. As the authors remark parenthetically, "Collapsing to simple names in this way can generate name conflicts, which in Cedar are avoided by careful name choice!" (exclamation point in original). Perhaps so, but this is still an undesirable feature.

To prevent versions of files from accumulating indefinitely, Cedar associates a **keep** with each local file name. The keep indicates how many versions will be retained locally. Usually, the keep for a source file is two, and that for a derived file such as object code is one, since previous versions are normally irrelevant. Unfortunately, keeps cannot be used with remote files. Clients must run utilities to get rid of unneeded versions.

The QuickSilver file system [Cabrera 87], when implemented, plans to support both multiple versions of files and update-in-place. Files that are read or written in whole-file transfers will be treated as immutable objects, as in Cedar. The QuickSilver designers propose the use of utilities, rather than keeps, to handle versions in a "coherent way." This is not an extremely attractive idea, since users are notoriously lax about cleaning up even their single-version file spaces. When update-in-place semantics are required, QuickSilver will provide only the actual file I/O. Concurrency control will be up to the application.

Maintaining multiple versions of a file certainly makes caching easier. Local cache managers need not worry about inconsistency since every version is unique. But it is not clear that this justifies saddling the file system with the job of version control. After all, an application program such as RCS can always be used to perform the same function. And RCS will, in fact, require considerably less disk space than a file system that retains explicit copies of every version.

Since an old version of a file may be accessible when the most recent version is not, availability is improved if a user is willing to accept stale data. (Of course, the user must be informed that the data **is** stale.) This will be helpful if, for example, a user happens to have an old version of a file on diskette that can be used when the file is otherwise unavailable. In addition, handling versions within the file system spares the user the burden of going through two mechanisms to access data: the actual read or write of the file and the check-out or check-in required by RCS.

CACHING AND STASHING

Caching is traditionally used to improve performance, but keeping an extra copy of a file in a cache can also increase availability. Perhaps the most significant decisions that must be made about caching are what to cache and where to cache it. This section will discuss two systems that answer these questions in completely different ways. A description of the related concept of **stashing** will conclude the section.

The primary goal of the Andrew file system [Morris 86, Satyanarayanan 85], called AFS, is high scalability, and it is apparently successful in that regard [Howard 88]. In order to meet this goal, AFS takes great pains to reduce both server utilization and network traffic. When a user needs to access a file, the entire file is transferred to the user’s local disk so that no further server interaction is needed until the file is closed.

Such whole-file caching is not an unreasonable approach. Indeed, several studies indicate that a high percentage of file accesses involve whole-file transfers [Ousterhout 85, Floyd 86], so support for caching at a finer granularity would be wasted much of the time. Further, whole-file caching is more likely to provide high availability, since entire files will be available in case of failure, instead of isolated disk blocks. Extremely large files such as databases, however, obviously cannot be manipulated in this way. The AFS designers are well aware of this restriction, but take the position that such support is not needed in their environment.

In the initial AFS implementation, a file in the local cache had to be validated before use. The client sent a message to the server requesting confirmation that the client’s copy of the file was still up-to-date. This placed an unnecessarily heavy load on the server, so validation was subsequently abandoned and replaced with a **callback** scheme. A cached file is now assumed to be valid unless the system has explicitly invalidated it by sending a message to the client. AFS keeps track of which clients are caching a given file, and sends callback messages to all of them when a new version of the file is written. This modification has improved performance significantly, while continuing to ensure cache consistency.

The Sprite network file system [Nelson 88], which is specifically designed for high performance, uses caching on both the client and server sides. Caching is block-oriented as in most centralized file systems. Furthermore, Sprite is intended to show the feasibility of diskless workstations, so all caching is done in memory instead of on disk as in AFS. If a file is concurrently write-shared, client caching is disabled so that a consistent view of the file can be maintained.

One unusual idea in Sprite cache management is dynamically varying the relative sizes of cache memory and virtual memory. The Sprite designers have no objection to a cache occupying the majority of a user’s memory, if not much space is needed for running processes. In fact, a Sprite file server uses the bulk of its memory as a file cache.

Block-oriented caching is more flexible than whole-file, but it is more expensive. It is a more complex model, and a harder one in which to maintain intra-file consistency. It is also likely to involve a heavier load on the server. On the other hand, whole-file caching seems like an immutable design decision. It is not difficult to imagine a block-oriented system being modified to support whole-file caching as well, but it is very hard to picture the reverse.

The merits of various caching media are also debatable. A memory cache will clearly be faster than one on disk, but it will certainly be smaller. In addition, if a crucial server is down for an extended period, there may be no way to save a file on a diskless Sprite workstation. (Because of Sprite's delayed writes, relatively brief server crashes may go unnoticed by the client.) Further, if a client crashes, data is more likely to be lost when using memory caching.

Either cache medium, however, will provide at least some amount of increased availability. Even if a client is completely disconnected from the rest of the system, file data in the client's cache will still be available for use. Depending on the caching method used, this data may consist of anything from a single block of a file up to many separate files. Because of its larger capacity, a disk cache appears superior for obtaining high availability.

The concept of **stashing** involves anticipatory file reads - figuring out what data the client is likely to need next so that it can be fetched in advance. This is important for availability because a failure is less likely to have an effect on a user whose heavily used files have already been fetched and stashed locally. Since Coda [Satyanarayanan 89] typically has an entire volume (all of a user’s files) in its local disk cache, the cache actually doubles as a stash.

The proposed FACE system [Alonso 89] maintains a stash that is distinct from its cache. A stash contains **quasi-copies** of a file: copies that may be somewhat out-of-date, but are never older than a certain fixed limit. The stash is continually refreshed by a **bookkeeper** that requests a new copy of a file when the current quasi-copy becomes too old. All refreshes are client-initiated so that the server does not have to keep track of when files need to be refreshed. This reduces the load on the server. A simple optimization is for the client bookkeeper to include the timestamp of the current quasi-copy in its request for an updated version. The server need not send a new copy if the client already has the latest version of the file.

FACE proposes several methods for specifying which files are to be stashed. A user may list frequently used programs such as text editors and compilers in a ".stashrc" file. The system should also be able to analyze "make" program files to determine which user files are likely to be needed. Alternatively, a user can explicitly tell the system when to start and stop monitoring file usage. Perhaps best of all, the system might determine a user’s "working set" of files by monitoring what a user does and dynamically deciding which files to stash.

Since Sun’s NFS [Sandberg 85] is so widely used, FACE is implemented as a set of enhancements to that system. Several extra fields are added to NFS data structures to support stashing, including a field that indicates whether the user wishes normal file accesses to be directed to the stash instead of to the remote file server. Because stashed copies are not guaranteed to be the latest version of a file, this strategy is recommended only for files that change very infrequently. In the normal case, the stash is used only when the file is otherwise unavailable.

FACE provides high availability at the cost of possible (but usually not severe) inconsistency. Coda’s approach of using the disk cache as a stash when needed seems preferable, however. The consistency is decidedly higher, and is not very expensive to maintain since callbacks are used.

LOGGING

Some file systems make use of a technique traditionally associated with database systems: logging. A log is a redundant collection of all updates. Unavailable or corrupted data can be reconstructed by replaying updates sequentially. By definition, one appends only to the end of a log. Therefore, writing updates to a log makes disk writes sequential rather than random. This greatly reduces seek time and improves write performance.

We will discuss two systems that use logging techniques in different ways. One is a working system that logs only a certain class of information. The other is a proposed design that makes the seemingly outrageous claim that the entire file system can be stored in a single log.

Hagmann's reimplementation of the Cedar File System [Hagmann 87] logs only "metadata" such as directories. The main goal here is to make crash recovery fast. The original version of Cedar (CFS), using a 300 megabyte disk drive, took at least an hour to recover from a crash. This was because atomic update of directory information was not supported, and the entire disk had to be analyzed in order to restore consistency.

In the reimplementation (FSD), the log of metadata obviates the scavenger hunt through the disk. By simply replaying the log, FSD can reconstruct a consistent directory. Crash recovery time is reduced to twenty-five seconds.

Although metadata must be both logged and written, normal-case performance does not suffer. Modifications are made to buffered copies of metadata pages, and then logged. When the log wraps, pages whose most recent version in the log are about to be overwritten are finally written to the appropriate directory pages on disk. But because locality is so high in name tables, almost no writes of directory pages actually take place. There is almost always a more recent log entry for a metadata page than the entry that is about to be overwritten.

FSD does not log a file’s data pages. This is based on the beliefs that hot spots are rare and that most files are written exactly once. So unlike metadata pages, logged data pages actually would end up being written twice: once in the log, and once in the file.

Ousterhout and Douglis [Ousterhout 89] make a radical proposal: restructure the entire file system so that all files exist solely as entries in a single log. Since this system relies on the idea of disk arrays, it will be enlightening to discuss that concept first.

Redundant arrays of inexpensive disks, or RAID [Patterson 88], is an approach aimed at increasing the performance of disk I/O. Although CPU speed and memory capacity have increased (and continue to increase) at a dramatic rate, performance of single large expensive disks (SLEDs) has improved only slightly. So instead of using a small number of SLEDs, a RAID employs a large number of inexpensive disks.

Since inexpensive disks are considerably slower than SLEDs, this method may seem counterproductive. But if the cheaper disks are run in parallel, and data is interleaved across disks, I/O requests can be broken down into multiple, simultaneous operations on multiple disks in a RAID. In a transaction processing environment, each disk can be used independently, allowing several transactions to perform disk I/O concurrently. In either case, the effective bandwidth is considerably higher than when using SLEDs - as much as twelve times higher.

Since disks fail independently, adding many more disks to a system greatly decreases the expected time between failures. So in order to make disk arrays feasible, some disks must contain redundant data for backup purposes. There are many ways to manage this redundant information. The designers pursued five successively better approaches, beginning with simple mirrored disks, and ending with a scheme that interleaves the data and error correction information across all disks in the RAID. No effort is made to make the system’s mean time to failure significantly longer than the product’s expected lifetime. Who cares if a RAID fails only once a century when the hardware itself will probably be used for less than twenty years?

Ousterhout and Douglis’s log-structured file system builds on the RAID idea, and attempts to reduce the time spent performing I/O even further. Recognizing that seek time is critical, the system does its best to make disk access sequential rather than random. On writes, the system succeeds admirably: new data is always appended to the end of the log. And given a large memory cache, most reads can be satisfied directly from the cache. Actual disk reads, it is claimed, can be handled with reasonable efficiency. Furthermore, the log approach is well-suited to use with large disk arrays.

There are several other alluring features of this idea. First, crash recovery will be very fast, since there is no need to analyze all directory and allocation information in order to effect repairs. Only the most recently written blocks need be examined. Second, files will exhibit **temporal locality**: files written at about the same time will be stored near each other on disk. This may well be helpful when the files are read. Finally, a versioning system would be relatively easy to add, since a new version of a file does not overwrite the old copy. (This would not be an RCS-like scheme, however, since every byte of every version would be retained.)

Writes are always extremely efficient, since data is simply added sequentially to the end of the log. Reading is much trickier, however. While most reads will be from the cache, some reads must obviously be from disk, and the system has a scheme for making these reads fairly efficient. Rather than having the directory, or "map array," stored at some fixed location on the disk, a "floating-map" technique is used in which map entries are added to the log in exactly the same fashion as data blocks. After blocks containing file data have been written, a new map entry is logged that points to all active blocks in the latest version of the file.

Now that map entries are no longer simple to locate, a map of map blocks is needed: the "super-map." The super-map is retained in memory, and is also periodically logged. After a crash, the system need only scan back to the most recent write of the super-map, and proceed from there. This is similar to checkpointing in a database system.

The system can now locate any file by scanning all map blocks pointed to by the super-map. When the proper map entry is found, the file itself can be pieced together. Since files are usually written in their entirety - and therefore stored sequentially in the log - the seek time may not be as long as it first appears. If there are many map blocks, however, much seek time may be required just to locate the correct map entry. This is an inescapable disadvantage of the floating-map technique, but since map blocks are cached, the average I/O time should not be significantly affected.

Since disks are, alas, not of infinite capacity, there must be a way to handle log wrap-around, and this system uses an incremental approach. As the wrap point advances within the log, live blocks are copied to the head of the log, overwriting dead data. This keeps live data physically contiguous, but at the expense of a great deal of recopying.

Of course, there must be some way of determining whether or not a given data block is alive. The requisite file maps will normally be in the cache, but this is not a complete solution. The system can only verify that a block is dead by scanning **all** of the map entries - and the number of entries is likely to be very large.

One could reserve space in each disk block to identify the file to which it belongs, but this is both wasteful and highly problematic. Unless the disk hardware is capable of handling these labels during

DMA operations, which is unlikely, a considerable amount of overhead will be needed. Each disk block will have to be processed individually, and each will require a separate I/O call. Alternatively, the system could maintain a bit map indicating which blocks are alive and which are dead. But this map will be extremely large, and will itself have to be periodically logged.

The idea of a log-structured file system is novel, thoroughly non-traditional, and a potential way to get around the disk I/O bottleneck, especially when coupled with the idea of "striping" the log across a disk array. Whether this approach can be made to work well remains to be seen, since neither the authors nor anyone else has ever built such a system. Ousterhout and Douglis plan to implement a prototype shortly.

CONCLUSION

Quite a number of distributed file systems have been built, and several of them are specifically designed to be highly available. Most of these systems pay a substantial performance penalty for their availability, however, because they provide consistent copies, and there is an intrinsic trade-off between consistency and availability. Are there some lessons to be learned here?

First, although replication is certainly mandatory, the degree of replication need not be very high. Even a simplistic two-copy approach - primary and backup - is likely to be effective almost all of the time. For files whose availability is critical, however, such a scheme may be insufficient. A system might allow users to assign different degrees of replication to different files, based on the need for availability of each file. A sophisticated system might even make educated guesses about the relative importance of various files. For example, a temporarily unavailable object file is no disaster if the source code is available for recompilation, but an unavailable source file may bring a programmer’s work to a halt.

Second, any file system that wants to be highly available must be optimistic in the face of network failure. A conservative scheme would be forced to deny access to a file in at least one part of the partition, since it would assume that an unavailable replica was being updated somewhere else. This is a poor scheme since conflicts are the exception, not the rule. LOCUS and Coda are certainly correct in their decision to detect conflicts after the fact rather than trying to prevent them from occurring in the first place. One major exception to this rule is database systems, in which write-sharing is commonplace and conflict prevention is worthwhile. (Coda, of course, does not support databases at all due to its use of whole-file caching.)

Finally, caching at both the client and server is of crucial importance. As discussed in the section on caching and stashing, a well-filled client cache or stash may allow a user to work even when disconnected from the rest of the system. The Coda system actually implements disconnected operation, but the performance is not yet fine-tuned. The difficult part of such a scheme is deciding how to manage the cache or stash in order to fill it with files that will be of most use to the client. Predicting the future is beyond the capabilities of most computers, so some heuristic should be used - preferably a simple one, or at least no more complex than those proposed for use in the FACE system.

The potential to increase the availability of files is one of the strong points in the idea of a distributed system. Much promising work has already been done in this area, but more research is needed before highly available file systems become a standard facility.

SYSTEM	HIGH AVAIL.	SCALABLE	REPL. UNIT	MULT. VERS.
Andrew	No	Yes	Volume	No
Coda	Yes	Yes	Volume	No
Cedar (old)	No	No	File	Yes
Cedar (new)	No	No	File	Yes
FACE	Yes	No	File	No
LOCUS	Yes	No	Pack	No
QuickSilver	No	Yes	File	Yes
RNFS	Yes	No	File	No
Roe	Yes	No	File	No
Sprite	No	Yes	File	No

~~Andrew and Coda both use whole-file caching on a client's workstation disk. Sprite~~ uses client memory caching on diskless workstations. Sprite is scalable, but does not scale as well as Andrew [Howard 88].

References

[Alonso 89] R. Alonso, D. Barbara, and L. L. Cova.
FACE: Enhancing Distributed File Systems for Autonomous Computing Environments.
Technical Report CS-TR-214-89, Princeton University, March, 1989.

[Alsberg 76] P. A. Alsberg and J. D. Day.
A Principle for Resilient Sharing of Distributed Resources.
In *Proc. Second Intl. Conf. on Software Engineering*, pages 562-570. October, 1976.

[Cabrera 87] L. F. Cabrera and J. Wyllie.
QuickSilver Distributed File Services: An Architecture for Horizontal Growth.
Technical Report RJ 5578 (56697), IBM Almaden Research Center, April, 1987.

[Ellis 83] C. S. Ellis and R. A. Floyd.
The Roe File System.
In *Proc. Third Symp. on Reliability in Distributed Software and Database Systems*, pages 175-181. IEEE, 1983.

[Floyd 86] Rick Floyd.
Short-Term File Reference Patterns in a UNIX Environment.
Technical Report TR 177, University of Rochester, March, 1986.

[Hagmann 87] R. Hagmann.
Reimplementing the Cedar File System Using Logging and Group Commit.
In *Proc. Eleventh ACM Symp. on Operating System Principles*, pages 155-162. November, 1987.

[Howard 88] J. H. Howard et al.
Scale and Performance in a Distributed File System.
ACM Trans. on Computer Systems 6(1):51-81, February, 1988.

[Kleiman 86] S. R. Kleiman.
Vnodes: An Architecture for Multiple File System Types in Sun UNIX.
In *Proc. 1986 Summer Usenix Conf.*, pages 238-247. June, 1986.

[Marzullo 88] K. Marzullo and F. Schmuck.
Supplying High Availability with a Standard Network File System.
In *Proc. Eighth Intl. Conf. on Distributed Computing Systems*, pages 447-453. May, 1988.

[Morris 86] J. H. Morris et al.
Andrew: A Distributed Personal Computing Environment.
Comm. ACM 29(3):184-201, March, 1986.

[Nelson 88] M. N. Nelson, B. B. Welch, and J. K. Ousterhout.
Caching in the Sprite Network File System.
ACM Trans. on Computer Systems 6(1):134-154, February, 1988.

[Ousterhout 85]
J. Ousterhout et al.
A Trace-Driven Analysis of the UNIX 4.2 BSD File System.
In *Proc. Tenth ACM Symp. on Operating System Principles*, pages 15-24. December, 1985.

[Ousterhout 89]
J. Ousterhout and F. Douglis.
Beating the I/O Bottleneck: A Case for Log-Structured File Systems.
ACM Operating Systems Review 23(1):11-28, January, 1989.

[Parker 83] D. Stott Parker et al.
Detection of Mutual Inconsistency in Distributed Systems.
IEEE Transactions on Software Engineering SE-9(3):240-247, May, 1983.

[Patterson 88] D. A. Patterson, G. Gibson, and R. H. Katz.
A Case for Redundant Arrays of Inexpensive Disks (RAID).
In *SIGMOD 88*, pages 109-116. ACM, 1988.

[Popek 85] G. J. Popek and B. J. Walker.
The LOCUS Distributed System Architecture.
MIT Press, 1985.

[Rifkin 86] Andrew P. Rifkin et al.
RFS Architectural Overview.
In *Proc. 1986 Summer USENIX Conf.*. June, 1986.

[Sandberg 85] R. Sandberg et al.
Design and Implementation of the Sun Network Filesystem.
In *Proc. 1985 Summer USENIX Conf.*, pages 119-130. June, 1985.

[Satyanarayanan 85]
M. Satyanarayanan et al.
The ITC Distributed File System: Principles and Design.
In *Proc. Tenth ACM Symp. on Operating System Principles*, pages 35-50. December, 1985.

[Satyanarayanan 89]
M. Satyanarayanan et al.
Coda: A Highly Available File System for a Distributed Workstation Environment.
Technical Report CMU-CS-89-165, Carnegie-Mellon University, July, 1989.

[Schroeder 85] M. D. Schroeder, D. K. Gifford, and R. M. Needham.
A Caching File System for a Programmer’s Workstation.
In *Proc. Tenth ACM Symp. on Operating System Principles*, pages 25-34. December, 1985.

[Sidebotham 86]
Bob Sidebotham.
Volumes: The Andrew File System Data Structuring Primitive.
Technical Report CMU-ITC-053, Carnegie-Mellon University, Autumn, 1986.

[Tichy 82] Walter F. Tichy.
Design, Implementation, and Evaluation of a Revision Control System.
In *Proc. Sixth Int. Conf. on Software Engineering*. IEEE, September, 1982.

[Welch 86] B. Welch and J. Ousterhout.
Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System.
In *Proc. Sixth Intl. Conf. on Distributed Computing Systems*, pages 184-189. IEEE, May, 1986.