111

The Echo Distributed File System

Andrew D. Birrell, Andy Hisgen, Chuck Jerian,
Timothy Mann, and Garret Swart

Friday, September 10, 1993

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

The Echo Distributed File System

Andrew D. Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garret Swart

Echo is an ambitious distributed file system. It was designed around a truly global name space. It uses a coherent caching algorithm. It is fault tolerant. And it is real—it was the primary file system for a large group of researchers. Its novel aspects include an extensible "junction" mechanism for global naming; extensive write-behind with ordering semantics that allow applications to maintain invariants without resorting to synchronous writes; and fault tolerance mechanisms that are highly configurable and that tolerate network partitions. It was designed with the intention that its performance could be as good as a local file system, while supporting large numbers of clients per server. Its reliability was designed to be higher than other distributed file systems, and higher than centralized systems. It was designed to work well in arbitrarily large networks.

CONTENTS

WHAT AND WHY?

One of the holy grails of operating systems research is building a system that combines the virtues of centralized time sharing systems with the virtues of distributed and personal computer systems. This grail is all the harder to grasp because researchers have different views of what the virtues of the respective systems are and the environment in which such a "best of both" system is to operate.

In 1988 we started the Echo project to capture the file system portion of this grail. We thought of the Echo file system as a crucial first piece of a truly distributed system. When we started Echo we had a particular view of the virtues of centralized and distributed systems we were trying to capture. From the world of centralized systems, we wanted to have:

- Easy sharing of data;
- Centralized administration;
- A simple failure model—either the system works or it doesn't.

From the worlds of distributed and personal systems we wanted:

- Fault tolerance—if one service is down, the user can access another, either manually or automatically ;
- Scalability ;
- Proximity of computing power to the user.

In this section we outline what properties a file system has to have to achieve these virtues. The remainder of this paper describes how the Echo design and implementation set out to realize these properties, and our experience with the resulting system.

A Distributed System Is Not Just a Network of Computers

Computer networks are a low-level technology—they allow programs to transfer data and control amongst the connected computers. A collection of computers inter-connected by a network does not of itself constitute a distributed system. The collection becomes a *system* only when the component parts co-operate sufficiently well that the collection behaves as a coherent entity. It should appear coherent to the user, to the system managers, and to the programmers.

We believe that to achieve this coherence, and hence to qualify as part of a distributed system, a network-based file system must have the following properties.

- *Global naming:* It must be possible for a user or program on one computer to utter a name for a file, with the assurance that that name will have the same meaning to another user or program elsewhere in the same distributed system.

The alternative to global naming is context-dependent naming. Most often, this means that file names are interpreted relative to some per-host root (as in NFS[1]). Sometimes too, the actual naming hierarchy is host-dependent (as in NFS, if remote volumes are mounted in a host-dependent way). Without global naming, users and programs must always be aware of the context in which a name is meaningful. This notion of "context" removes the coherence from the system. A program executing a distributed algorithm cannot use file names to communicate amongst its parts; a program cannot use file names to communicate with itself across time; users cannot effectively pass file names amongst themselves.

- *Global access:* The same files and directories must be accessible everywhere throughout the system—independently of location—with the same operations available on them. The performance, availability, and reliability of these operations might vary depending on the location, but the semantics should not. One way in which a file system can fail to provide global access is if the naming hierarchy differs on a per-host basis (as in NFS), so that not all files are accessible from all hosts. This restricts the location transparency of the distributed system (e.g., when you are in Paris you might not be able to access your files in New York). A more insidious failure to provide global access occurs if the file system uses a non-coherent caching strategy (as in NFS), whereby updates made to a file on one host are not necessarily visible to a program on another host. This restricts the ease with which you can build distributed algorithms—the application must take explicit steps to achieve the coherence that the file system failed to provide.

- *Global security:* The security of the file system should be no worse than that of a well-built time-sharing system. When your system uses a network, especially a network that goes off-site, your system becomes vulnerable to all manner of attacks from strangers. And even if the network is only inside your own building, your system becomes more vulnerable when you allow the owner of each workstation to control the installation and configuration of that workstation's system—taking the place of the trusted manager of your centralized time-sharing system. The techniques for countering these threats are now understood [1,5,12], but currently available distributed file systems (such as NFS) do not apply these techniques.

Reaping the Benefits of Distributed Systems Technology

Networked computer systems have several natural advantages over centralized ones. These include ease of growth (you can start small and grow without wasting your original investment), autonomy of growth (small groups can make their own purchasing decisions, instead of relying on centralized resources), and lifetime (you can upgrade

---

1 We need to use examples to make this discussion more concrete, and we're mostly using NFS as the "bad guy". This choice is primarily because NFS is so well known, not because NFS is worse than its competitors. Indeed, in many ways it's better. But NFS provides no global naming, poor global access and very little security. Despite its various desirable properties (notably its simplicity and its pervasiveness), NFS falls far short of being an ideal distributed file system.

incrementally to new versions of sub-systems or to replacements for existing sub-systems). These benefits come naturally from adopting a network computer inter-connect, although the file system designer still needs to take care to avoid losing them. But there are two further benefits achievable in distributed systems, if the designer addresses them explicitly:

- *Fault tolerance:* in a well-designed distributed system, you can provide the same service from multiple computers, and you can do so in such a way that the service remains available even if one (or more) of the computers fails, or even if part of the network fails. This is not just an opportunity for doing better than centralized systems—it is essential if you are to do as well. Using multiple computers increases the probability of one of them failing and preventing you from getting your work done. Thus a good distributed file system must offer fault tolerance. It should be configurable to provide any desired level of reliability (by replicating data) and of availability (by offering access to the data through more than one computer).

- *Scale:* a well-designed network can grow very large (e.g. the Internet, with about half a million registered names). With care, the distributed file system also can be effectively unlimited in scale. But done badly, the file system will hit its scale limits long before the network does. For example, you could design your file system to use a proprietary service for its global naming. But this would prevent you inter-connecting with the existing name services that are literally global (the Internet's Domain Name Service and ISO's global X.500 name space). Or you could rely on a security system that does not allow for differing levels of trust across the naming hierarchy, or that requires too much manual intervention to build a truly large system (e.g. Kerberos version 5 uses *inter-realm* links to achieve security across the untrusted Internet, but all these links must be installed pairwise and manually). All aspects of the file system design are affected by considerations of scale. There is no particular scale mechanism described in this paper; rather, the need to scale well explains several of the design decisions described in the following sections.

Summary: The Challenges Addressed by the Echo Project

The Echo project is an attempt to learn how to build a distributed file system meeting the requirements we have outlined, and to actually build such a system—well enough to be the dominant file system in daily use by a large and active group of researchers. In fact, the Echo project has constructed a distributed file system with the following properties:

- *Global Naming*—while retaining global access, security, fault tolerance and scalability.

- *Global Access*—despite the use of local caching, and in the face of fault tolerance mechanisms.

- *Global Security*—as good as a time-sharing system, but with large scale and geographic dispersion.

- *Fault Tolerance*—while using local caches for performance, and accepting the possibility of widely dispersed clients.
- *Global Scale*—but without global trust, and without compromising performance or availability in the dominant local-area cases.

The remainder of this paper describes how Echo achieves this. More details about many of these topics are presented in three other Echo papers [10, 14, 23].

Assumptions

We made a few assumptions throughout this project. If you don't agree with these assumptions, you probably won't like what we've produced.

- Good RPC: we have an RPC system that provides very good performance (about 2 milliseconds round-trip for simple calls, using 3 MIP processors) [22,24], and has powerful features (most of the Modula-2 type system, plus additional support for remote context handles, marshallable bindings, distributed garbage collection, and authentication). All the communication with Echo servers uses RPC exclusively.
- Fail-stop servers: we assume that our servers either give the correct answer, or give no answer (or an exception). They never give a wrong answer. Of course, this is an over-simplification. But any behavior that violates this assumption is by definition a bug and gets fixed. A correct server can measure intervals of time with no more than a known error bound; however a correct server can have arbitrary performance characteristics.
- Fail-stop media: our storage (disks) either return the data that we stored there, or give an error (although the error might not be reported until you next try to read the data). This assumption is very close to being true, we believe.
- Byzantine clients: the correctness of the service (i.e. the answers we give to clients) is unaffected by incorrect clients (or incorrect operating systems on client computers). An error on a client computer might at worst cause all clients on that computer to get incorrect results, but it cannot cause incorrect results to be given by Echo to clients on other computers. In other words, we intend that however much a client or client computer malfunctions, it seems to clients on other computers just as if that one computer was making strange but valid operations. However, a malfunctioning client or client computer might cause denial of service to others (e.g., by overloading the server, or by malicious behavior in the cache token algorithm).
- Liveness: the service is correct independently of the liveness of the client computers, the servers, or the network. But the liveness or performance of the system can be affected by the liveness of all of these components.

GLOBAL NAMING

Echo provides a global hierarchic name space: a tree of labelled arcs, with a single common root. File names are paths through this naming tree. Each file name path consists of a series of arcs. The arcs are looked up sequentially in directories, to yield further directories, until the final arc yields the named object—either a directory or a file. In addition, there are symbolic links: resolving an arc might lead to a specialized node that contains another name, to be pre-pended to the remainder of the original path. All of this is just the same as in a conventional centralized Unix system.

Unlike a centralized system, Echo's name space is implemented by multiple computers dispersed across the network. Unlike NFS, the Echo name space has a single global root, world-wide. Unlike NFS, the meaning of an arc in the Echo name-space is independent of the node on which the application using the name is running (mostly; see later for the exceptions).

The Echo name space consists of *volumes* glued together by *junctions*. An Echo volume is just a sub-tree of the name space. Each volume resides on a single server (or a set of replicated servers), although one server might implement many volumes. A junction is a leaf in one volume, containing a description of the location and identity of a further volume. During normal name resolution, if an arc leads to a junction the name resolution mechanism notices this, interprets the junction's data, contacts a new server for the identified volume, and continues name resolution at the root of the identified volume. The Echo notions of junctions and volumes are analogous to the mounting of file systems in a centralized Unix system or in NFS. But in Echo there is an important difference. The junction is a static object giving the identity of the child volume. All clients of the system see the same volumes in the same places in the name space. Client programs and their operating systems take no explicit "mounting" action; the volumes are always in their places. (AFS version 4.0 is using a similar arrangement [17].)

Echo's mechanism of volumes and junctions is open-ended. There are several different classes of Echo volumes, with slightly differing semantics and properties. They are all glued together with this one junction mechanism, transparently to clients. The most important volume classes in Echo are the Echo name service and the Echo filestore. There is also a volume class providing access to NFS file systems.

Figure 1 (overleaf) shows the overall structure of the parts of Echo dealing with name resolution. When the client's system wants to perform name resolution (e.g. when the client application calls "open"), the system presents the name to the Echo file system switch. The switch directs the call to a *clerk*, specialized to one class of volume (initially the clerk for the global root volume). This clerk calls appropriate servers to access the volume. If a server encounters a junction, it returns the junction data to the clerk, which gives the data and the remainder of the path name to the switch. The switch examines the junction to determine the volume class, and calls another (or the same) clerk to continue the name resolution.

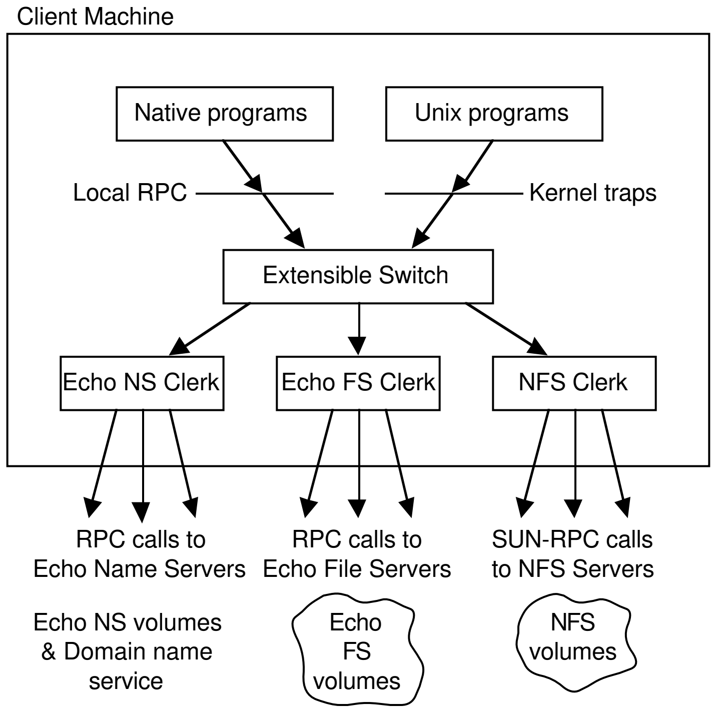

Figure 1: The structure of Echo in a client computer

The global root volume in Echo has to provide world-wide service. So we chose a world-wide name service: the Domain Name Service (DNS) provided in the Internet. DNS provides a tree with essentially the properties just described. You resolve arcs and get to nodes. It's sufficiently open-ended that you can represent junctions (as text resource records, for example).

Echo clients don't actually communicate with DNS servers. Instead, the Echo name servers provide surrogate access to data from DNS. When an Echo client wants to access the root (DNS) volume, it uses the Echo name service clerk to talk to an Echo name server. This server either returns cached data to the client, or talks to the real DNS servers on the client's behalf. Thus Echo clients need not understand how to locate and talk to DNS servers—instead they use our RPC protocols to talk to Echo name servers. More importantly, though, using the Echo name servers as intermediaries allows us to provide better availability.

Since we're relying entirely on global naming, our overall availability is limited to the availability of the root volume. It would be unacceptable if failures in the high levels of DNS caused failures for all Echo clients. To avoid this, the Echo name servers *stash* [3] data from DNS in stable storage. If a client asks an Echo name server for data from the root (DNS) volume, and the server's cached data has expired, but the appropriate DNS server is not available, the Echo name server returns its cached copy anyway. This stash is long-term, persistent across restarts.

The Echo name servers implement a second class of volumes. We use these name service volumes as the next level in our name space, immediately below the root volume. We maintain these volumes using traditional techniques that provide high availability, but low consistency [2,13]. The name service volumes are replicated. An enquiry can be satisfied by any replica. An update can be made at any replica, and is committed there before returning to the client. But the update propagates asynchronously to other replicas of this volume. In principle this update propagation can take a long time, but in practice updates almost always reach all the replicas in under one second.

Most of the files and directories in Echo are stored in a third class of volumes, the Echo Filestore volumes. These behave more like a conventional file service. Although they are replicated (to provide our fault tolerance), they provide tight consistency—all clients see the same data all the time. These volumes use a scheme involving an elected *primary.* (The section on fault tolerance gives more details.) All enquiries are made at the primary, as are all updates. The primary propagates updates to all available replicas (necessarily a majority) before returning to the client.

To see how this works, consider figure 2. A client in a workstation wants to resolve the path name “/-/com/dec/src/x/y/p/q”. The symbol “/-” means that the path starts at the global root. The arcs “com/dec/src” are resolved in DNS, much the same as you would resolve the Domain name “src.dec.com”. But as described earlier, this step of the name resolution is performed by an Echo name server and is cached and stashed. At this point the name resolution encounters its first junction (defined by a Text resource record in DNS), which describes a volume in the Echo name service. The client’s system proceeds by contacting the appropriate Echo name server (one of several storing this volume), and presenting a request to resolve “x/y/p/q” relative to this Echo name service volume (specified by UID). The name server can resolve “x/y”, and returns the junction data found there. This junction specifies an Echo filestore volume, so the client’s system uses the Echo filestore clerk to contact the appropriate Echo file server, and asks to resolve

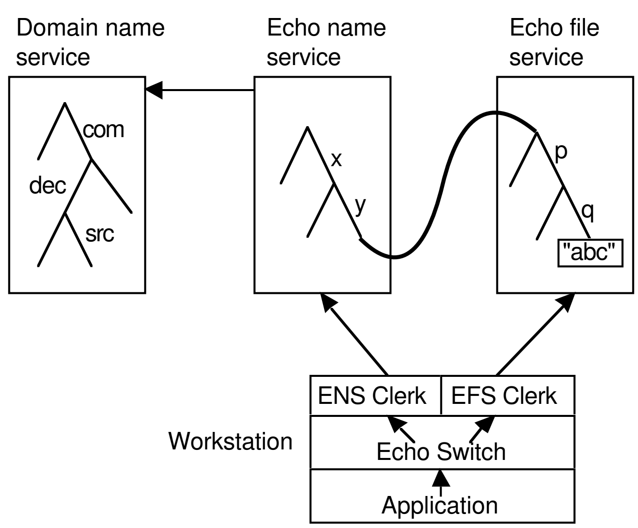

Figure 2: Resolving an Echo global name

"p/q" relative to the identified volume (specified by UID again). This resolution succeeds, and the Echo file server performs whatever operation the client is requesting.

The preceding description ignores the effects of caching in the clerks. Each clerk caches data from its servers; indeed, most names are resolved entirely from the clerks' caches, without contacting any servers at all. The caching algorithms are discussed in detail in a later section.

Why not just use DNS and abolish the Echo name service *per se?* While DNS is very effective at naming hosts in a widespread network, it is too rudimentary to be the only name service in a distributed system. There is no widely accepted update mechanism, no plausible security mechanism, and only marginal support for enumeration. It really doesn't map very well into Unix file semantics. We use DNS at all only because of its world-wide presence. It is essential as the top-level glue, but at the earliest opportunity, we use a junction into our own name service—which *does* map well into Unix semantics.

Why not just use the Echo filestore and abolish the Echo name service *per se?* The higher level parts of the name space have quite different requirements from the lower level parts. At the high level, the organization and contents of the name space change quite rarely, and when they do change it is often because of some significant event or upheaval—adding or removing a user, or re-organizing a department. At the lower levels, the organization and contents are changing rapidly as users create, modify and delete files. Further, the availability of the higher levels is more important—if the high level names are unavailable, all the descendant volumes become unavailable too. There are no known algorithms that simultaneously provide tight consistency guarantees and extremely high availability. So we decided that at the higher levels we would use traditional name service algorithms to maximize availability at the expense of consistency, and at the low levels we would use the Echo filestore to provide tight consistency at the expense of somewhat lower availability.

The different consistency guarantees provided by the different classes of volumes are mostly invisible to our clients and users. For example, a user can apply the Unix "ls" and "mkdir" commands equally well in an Echo name service volume and in an Echo filestore volume. When a program is running on a single machine the looser consistency of the name service is completely invisible unless a name server crashes. But if a distributed algorithm is running on multiple computers and communicates with different name service replicas the looser consistency can be visible. We have had examples of this happening, and it is quite confusing. In each case, the programmer has agreed that he should have been storing his data in the filestore instead; but the programmer remained distressed by the occurrence.

Some of our names aren't global names. We offer two alternatives. As with other Unix-like systems, you can present names to be resolved relative to a per-process working directory. But in addition, names can be relative to a per-process "local root". For example, the path name "/bin/gcc" is local-root-relative. The local root, "/" is chosen when a user logs in, and is inherited when forking child processes. Of course, "/" actually maps into some node in the single global naming tree; just the choice of which point is process-dependent. This mechanism allows you to make group decisions, such as when to upgrade to a new version of gcc. Since every process created by everyone in the group uses the same local root, "/" , they all resolve "/bin/gcc" to the same file. But processes belonging to outsiders, with other values of "/" might get a different compiler. Again, here we have deviated from our principles—not all names are actually global names, because you don't really want global naming all the time.

Notice that our notion of local roots is quite different from the multi-rooted name space offered by systems such as NFS and AFS [17,21]. Our local root is just a pointer into the global name space; the other systems are truly multi-rooted hierarchies.

The Echo junction mechanism is quite open-ended. The implementation of the name resolution algorithm uses an object-oriented registration mechanism, so that additional volume classes can be added. For example, we have a volume class that implements "/dev", and one that provides access to a specialized repository for our source control system. Other possibilities that we have not yet explored include a volume class for naming processes or jobs.

GLOBAL ACCESS

To provide uniform global access we must ensure that all clients see the same data, regardless of where they are in the network. A modification made by one client must be immediately visible to all other clients. The Echo file servers use a replication scheme with tight consistency (described later), which satisfies this requirement. But if the file system is to perform well it must cache information in the client computers. To provide uniform global access these caches must be fully coherent.

Why use caches? One reason is to provide better service to the client, by avoiding network delays. (Notice that you can avoid disk delays more simply by caching in the server.) But another reason is probably more important. If data is cached extensively in the client's computer, the load on the server is reduced. In the limit, the server encounters only new data—written by one client, then read at most once by others. This benefit accrues even if the client cache is on a local disk, as in AFS [17]; although the client might not notice performance improvement when compared to uncached use of a lightly loaded server, the overall effect is that a server can handle many more clients.

The Echo filestore cache is implemented in main memory on the client. This is not fundamental, it's just an experiment. We could just as well place the cache in paged virtual memory or in an explicit local-disk file system without affecting the rest of this discussion. However it is important that the cache is large. Ideally it should contain the client's entire medium-term working set, so that the server is used only for reading or writing new data. In practice today we use about 20 megabytes.

There are two major issues in the Echo filestore cache design: how to achieve coherence, and what guarantees the write-behind mechanism should provide.

Caching for the Echo Filestore—Coherence

NFS uses a very simple coherence strategy for its client caches. Updates to directories are write-through—they go synchronously to the server, and to disk. Updates to files are write-behind, but are propagated to the server after the file is closed. This is an attractive engineering compromise, very simple and quite efficient. But it can produce very surprising results when executing a distributed algorithm involving shared files. And there are frequent complaints about the delays caused by the synchronous directory operations.

The Sprite system provides a totally coherent cache [19]. Although updates to files use write-behind, Sprite uses a token scheme so that if an application on another computer wants to access a file that has dirty pages in a client cache, those pages are first written back to the server for the other computer to access them. Sprite still uses write-through for directory operations. AFS version 4.0 uses a similar scheme.

The Echo filestore uses a caching scheme quite similar to Sprite's, but extended to deal with directories and to cope with replicated servers; we also deal differently with shared writing of files. Here is a simple outline of our scheme; we leave the more complex refinements for a separate paper [14]. The Echo filestore servers manage *tokens*, associated with each file. The servers issue these tokens to their clerks in the client computers. To hold a clean copy of data in its cache, a clerk must hold a read token; to hold a dirty copy, a write token. If any clerk holds a file's write token, no other clerk may hold read or write tokens on that file; but if not, multiple clerks can hold read tokens. (Note that these tokens have nothing to do with file locks or with which clients have a file open; they are needed only when a client actually reads or writes file data.)

Consider figure 3. If a client on WS1 wants to read a file, its clerk calls the server FS1 and obtains a read token on the file. Thereafter, the clerk can cache pages from the file. If WS2 also wants to read the file, it can get a read token at the same time. But if instead WS2 wants to write the file, the server calls back to WS1 revoking its read token, before granting a write token to WS2. Now WS1 has no access to the file, but WS2 can cache pages it reads from the file, and can also hold dirty pages for the file—thereby implementing write-behind. Finally, if WS1 again wants to read the file, the server calls back to WS2 revoking its write token. On receiving this call, WS2 synchronously writes any dirty pages back to the server, before returning from the revocation call. When this call returns, the server returns the read token to WS1.

This scheme is very efficient for most usage, although it behaves poorly if a single file is being updated frequently by multiple computers. In this "shared-write" case, Sprite decides not to cache the file at all. Our approach has proved satisfactory in our environment, where we run mostly Unix-style applications that do not make use of shared writable files.

Notice that choosing files as the granularity of the token is quite arbitrary. We could instead have used a token per page, or issued tokens for byte ranges. The per-file tokens require less data structure, so they are immediately attractive. Per-file tokens seem well matched to typical Unix usage patterns for files—each file is read and written in its entirety [18]. In some other systems the access patterns are different, and issuing byte-range tokens might be more important. There is experience of this in other systems [4], and it is planned for AFS 4.0.

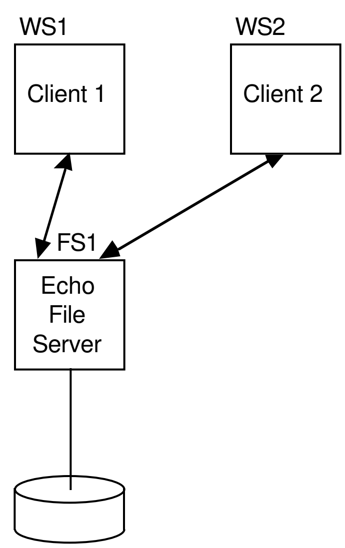

Figure 3: Token traffic between two clients and one server

We use exactly the same scheme for directories as for files. In this case we are less convinced that per-file tokens are the right answer. Our experience so far is leading us to consider whether we should adopt a more complex token scheme for directories, recognizing their specific update patterns.

Now consider figure 4, which depicts a similar situation except that the file is being stored on a replicated server (details of replicated storage come later). In the Echo filestore replication scheme one replica is elected as primary and the others are standbys. The token operations just described are done by clerks calling the current primary. When the primary fails and a new primary is elected from the standbys, we need to recover the state of the token algorithms. To enable this, we dynamically replicate the token state (in each server's volatile storage). Whenever a client acquires or releases a token, the primary tells each standby (using a 2-millisecond RPC). So after a fail-over, the new primary immediately has the token state.

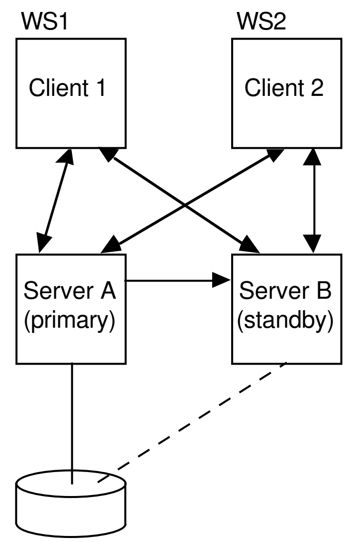

Figure 4: Token traffic with a replicated server

One alternative algorithm would have been to recover the token state by calling the clerks. We rejected this because of the vast load it would impose at recovery time, and because of the delays if a client was unavailable at recovery time.

The final part of the token algorithm is recovery when clients crash (or become inaccessible because of network partitions) while holding a token. We use timeouts to handle this. Tokens are issued to clients as part of a "session" (one session per client-server pair). The token is guaranteed to remain valid only as long as the session. The session remains valid only for a few seconds, unless explicitly refreshed by the client's clerk. So if a client becomes detached by a network partition, within a few seconds the server and the clerk both agree that the clerk's tokens can be implicitly revoked. This combination of tokens and timeouts is sometimes called a *lease* [6].

The server doesn't actually revoke the tokens unless they conflict with a token requested by another clerk—this allows us to ride through many transient network partitions without reporting any errors. If a clerk has lost session and one of its clients wants to access a file, the clerk must first contact the server to re-establish its right to have the appropriate token. If the clerk cannot then contact the server it reports an error to its client. Lost sessions also interact with the write-behind scheme—described the following section. Notice that this scheme does not require synchronized clocks, only clocks that agree within a known bounded error rate.

Caching for the Echo Filestore—Write-behind

The Echo filestore cache uses write-behind: when a client application creates, writes or deletes a file, or updates a directory, the system call returns to the client before the operation is transferred to the server or committed to stable storage. We have deliberately gone further than previous systems in using write-behind, as an experiment. For example, both NFS and AFS use write-through for directories, using write-behind only for file data. Our write-behind mechanisms are sufficient that a client application could create a file, write to it, re-read it, and delete it, entirely within the client cache. The write-behind mechanism could perform these operations without involving the server at all.

To make such elaborate write-behind useful to our application programmers, we need to give them strong guarantees about what operations actually reach the server, and in what order, so that the application can understand and control the states that it might encounter after a crash and restart.

The guarantee we give is specified as a partial order on the writes and updates that application programs make. We guarantee that the updates made to the filestore's stable storage will satisfy this partial order (i.e., although we might merge or eliminate operations before they reach stable storage, any re-ordering we make will not violate this partial order). For the purposes of defining the order, we consider any sequence of writes to a single file that don't alter the length of the file to be a single item; within such a sequence the Echo filestore clerk and server can re-order arbitrarily—to increase performance. Otherwise, for any particular file or directory, update operations involving that directory reach stable storage in the same order that they were issued by the applications. Since some operations can involve multiple files and directories, this defines a partial order on all the operations requested by the applications. For example, a rename operation affects up to four objects: the named object, its old directory, its new directory, and the old occupant of its new directory entry. So a rename operation constrains the write-behind order of all operations involving any of those four objects. Further, since our cache is entirely coherent (see the previous sub-section), we can offer this ordering guarantee across updates made by all applications in the entire system, world-wide. Notice that we are constraining only the order in which operations affect stable storage; Echo is still permitted to merge multiple client operations into a single stable storage operation. For example, we use group commit in our disk updates. Notice too that although we have the flexibility to adjust the order in which operations affect stable storage, the caches are still fully coherent—all clients see the effects of the operations in the order that clients requested them.

We give the applications two additional controls. The system call *fsync* blocks until all outstanding updates to a given file—and all updates that ordering rules guarantee will precede them—have reached stable storage. The new nonblocking system call *forder* counts as an update to each of its (up to four) arguments, although the update alters no contents. The only effect of forder is to further constrain the overall partial order.

So far, our experience with this has been mostly satisfactory. Applications do indeed use these ordering guarantees to maintain quite elaborate invariants on their stable storage, and the guarantees do not overly constrain our update algorithms. The applications use forder to maintain their invariants, without the need to resort to the much less efficient fsync.

One part we are not yet sure about is how much we will gain from the extensive write-behind capability. So far we have performed only minor optimizations on the write-behind stream; for example, we have not yet eliminated temporary files from the server traffic.

We have observed one interesting tension in this scheme. With so much write-behind, the queues of updates that haven't yet reached stable storage can get quite long—several minutes, if an application is issuing file system updates in a tight loop. While this is good for reducing server peak loads by smoothing out the offered load, it is bad for users who expect their work to be saved promptly on disk. We have added a watchdog mechanism to the clerk to reduce the danger of losing large amounts of work. If the write-behind queued up for a particular volume is more than five minutes old, the clerk blocks all applications that request more updates on that volume until the server catches up. This watchdog never goes off in normal use, though we can trigger it with test programs.

The most distressing problem in designing a cache that has write-behind is what happens when a clerk loses its session (and its tokens) because of a network partition or because of server failures not masked by our fault tolerance mechanisms. In this situation the clerk has a queue of operations that it has accepted from clients, but that it cannot commit to the servers' stable storage. The clerk must decide what to do with the operations, and how to report the failure to its clients. Notice that this problem does not arise in centralized systems, because there the clients crash at the same time as the file system. We are still unsure of the best way to report such failures—our initial design has not proved entirely satisfactory. In it, when a clerk loses its session on a volume, the clerk invalidates all application open files and working directories in that volume, but allows applications to reopen their files (and directories) using absolute pathnames. This scheme is meant to stop naive applications from continuing to modify the volume under the false assumption that their previous modifications were successful, yet allow sophisticated applications to recover. But in practice we have found both applications that fail with confusing error messages when they should recover (such as interactive shells), and applications that continue (using absolute pathnames) when they should quit. As a result we have explored some alternative designs; we discuss this issue further elsewhere [14].

Caching for the Echo Name Service

The name service cache is vastly simpler than the filestore cache, for two reasons. First, updates are quite rare, so coherence is less important. Second, the servers do not provide tight consistency for updates, so a little inconsistency in the caches won't make matters any worse. The name service clerk caches data from its servers for up to 30 seconds. An enquiry will return cached data if the data is less than 30 seconds old, otherwise it will ask the server and then update the cache. An update updates the cache on return from updating the server. This very simple algorithm has been satisfactory—both in terms of the answers given to clients, and in terms of reducing server load.

GLOBAL SECURITY

The Echo file system uses a distributed security facility that was developed as part of a separate project at our research center and that is described in detail elsewhere [12]. So this paper gives only a few highlights of Echo's security, with emphasis on how they address our requirements.

All communication between Echo servers and their clerks is authenticated. In other words, the clerk and server always know the identity of the principal making or receiving an RPC call. These principals are identified by global names in the same name space as we are using for file naming—rooted in the domain name service, with sub-trees implemented in the Echo name servers and file servers.

All Echo objects—files, directories, volumes, and servers—are protected by access control lists (ACL's), which specify what principals may perform what operations on the objects. An ACL is a set of names and access rights. The names in ACL's can be principals, or the names of other ACL's ("groups") stored elsewhere in our global name space.

The authentication and access control schemes will work even across a world-wide distributed system. They provide for differing levels of trust at differing levels in the name space, and secure cross-links to by-pass untrusted levels of the name space [5].

Our servers enforce the security of the objects they contain. They do not trust their clerks. The clerks are responsible for multiplexing correctly amongst multiple principals on the same computer.

This design sounds wonderful, but you may not be convinced it's implementable. To become convinced, you need to read the papers on our security system cited above.

FAULT TOLERANCE

It is important to distinguish two concepts: "reliability" and "availability". Providing a reliable system means that we will not lose or corrupt your data. Providing an available system means that we will let you get at your data.

The basic technique for providing reliability is to replicate the data storage. In other words, write all data to disk more than once. How many times more is a parameter, selected by you based on the value of your data, the probability of disk errors, and the cost of disks. Modern disk storage is very good. It is extremely unlikely that it will corrupt data without reporting an error (i.e., it is fail-stop). It's also highly likely to store data correctly. Generally, having two copies of the data is sufficient—at that level, you're more likely to lose data through operator error or earthquake than through disk error.

The basic technique for providing availability is to replicate the storage access paths. At the extreme, this implies replicating the disk drive too, since the disk itself is part of the access path. But in reality, modern disk drives are a lot more reliable than computers, and vastly more reliable than computer software. So if you want high availability, but you are satisfied with single-disk reliability, it is attractive to replicate the server without replicating the disks.

Fault Tolerance in the Echo Name Service

As mentioned earlier, the Echo name service achieves fault tolerance with techniques that are by now traditional. The overall architecture is very much like that described in a previous paper [15]. For each name service volume, there is a set of replicas. Each replica uses disk storage for a complete copy of the volume. An enquiry can be made at any replica, as can an update. Updates propagate asynchronously to all other replicas. A name service volume is available and reliable as long as one replica is running. There is nothing new here.

Fault Tolerance in the Echo Filestore—Configurations and Elections

The Echo filestore is quite flexible in its provision of fault tolerance. Various parts of the filestore can be configured to provide more or less reliability (by replicating disk storage),and more or less availability (by replicating servers). The challenge here is to provide these fault tolerance mechanisms while retaining our strong consistency guarantees: that an update made by one client is immediately visible to any other client, and that an update once committed is never undone. (From the client's point of view, "committed" is defined according the the write-behind rules I described earlier; but at the server interface, each update operation commits before the operation returns to the calling clerk.)

Volumes are grouped into *boxes*. Each box contains the entirety of some set of volumes, with no other relationship required amongst the volumes. The allocation of volumes to boxes is purely a managerial decision. The box is the unit of replication. Each box is stored on one or more replicas, each of which occupies some number of physical disks. Each box replica can be accessed by one or more servers.

Figure 5 shows the possible configurations. Configuration 1 is the minimal one: neither the data nor the server is replicated. In configuration 2, we offer higher availability by using dual-ported disk hardware so that two servers can access the same disk. The servers use an election scheme (described later) to decide which is primary. The primary does all disk accesses, until it crashes. In configuration 3 we provide high reliability (without enhancing availability) by storing the data on two disks. On every update, the server records the update on both disks before returning to the calling clerk (and client). Configuration 4 is the combination of 2 and 3: it provides high availability and high reliability. Again, the servers elect a primary, which performs all updates, writing them to both disks.

Configurations 5 and 6 provide much the same level of availability and reliability as

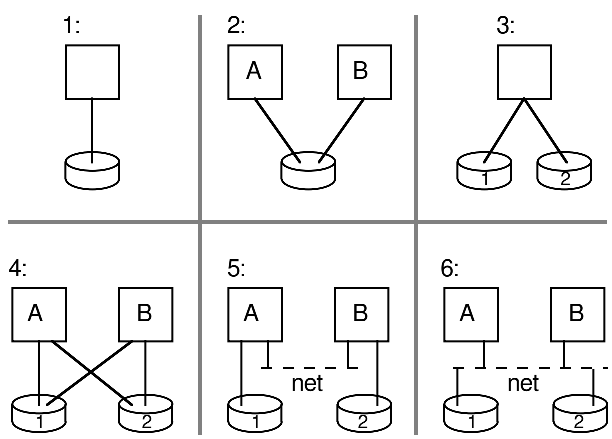

Figure 5: Echo filestore configurations

configuration 4, but with different hardware. Instead of using dual-ported disks, the primary must communicate with the second disk across the network. In configuration 5, the second disk is accessible only through the second server, but even without it the remaining server can provide service.

The algorithm for electing a primary amongst replicated servers is basically straightforward, although the details get quite complicated. We use a majority voting scheme: every disk replica gets one vote; whichever server gets a straight majority of the votes is the primary. The votes belong to disks, and they are accumulated by servers. A server can get the vote from a disk in various ways, depending on the configuration. In the dual-ported configurations (2 and 4) a server gets the vote by persuading the dual-porting hardware to give it ownership of the disk; or it allows the other server to get the vote by disconnecting from the disk. In configuration 5 (disks directly connected to single servers), the directly connected server controls where the disk's vote goes. In configuration 6 (disk servers on the network), the disk server hardware chooses where the vote goes.

The typical configuration has exactly two servers with exactly two disk replicas. If they are both connected and running, one will defer to the other (based on processor UID, for example). But if the network is partitioned so that there is no communication between the replicas and both are running, neither would get a majority. To avoid this we configure the system with three votes: one cast by each disk replica, and one cast by a bystander called a *witness*. The witness does not store files (at least, not for this box—in practice the witness is a file server for some other box), it just casts a vote. In any single network partition, the witness is accessible to one or the other server, so that server will get 2 votes—a majority. If there are multiple network partitions it is possible that no server will get a majority. In such a case no service will be provided.

Figure 6 shows an election in the face of network partitions. If there is a network partition at point X, the witness will vote with server B, but if there is instead a network partition at point Y the witness will give its casting vote to server A. If the network is partitioned at both X and Y, neither A nor B will get a majority, and no service will be provided.

There is one important improvement to this algorithm. If the primary that gets elected has an up to data copy of the token and session data structures, we will be able to

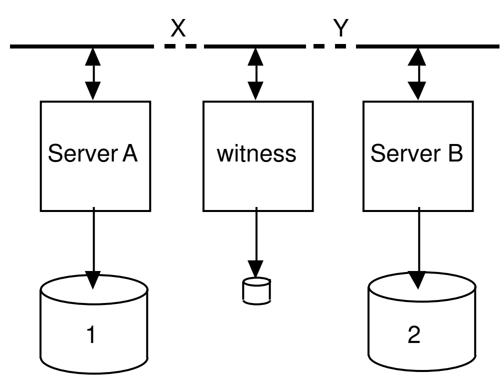

Figure 6: An election in the face of network partitions

provide uninterrupted service to our clients. Otherwise, the clerks will have lost their tokens and might have to report failures to their clients. So in reality our election algorithm biases the result toward a server that has this data.

Notice that this election just decides which server will be primary. We need a further algorithm to decide which disk replicas contain “truth” data and to reconcile the replicas. This is outlined in the next section.

Fault Tolerance in the Echo Filestore—Updates and Recovery

The Echo filestore uses a *log* (or *journal*) for recording updates on each box’s disk storage. When a client’s clerk calls a server to make an update, the server verifies the operation’s pre-conditions, updates the server’s volatile data structures, then writes a log record on one of its disks. If the box is replicated, the log record is written on all replicas. Then the server returns from the RPC call made by the clerk. When the clerk regains control, it knows the update has been committed to disk. Asynchronously, the server applies updates to their “home” location on its disks, and removes the update record from the log. (Actually, if the update is a large data write—more than 4K bytes—the write is made synchronously to the home location, with only a brief notation in the log.)

As an optimization, we have written our servers so that in the absence of crashes they never need to read the log. We do this by ensuring that the server’s buffer pool contains all the data affected by the log entries. This requires a substantial buffer pool in the servers—we use about 80 megabytes shared between this function and normal caching of pages to avoid disk reads—but the gains from never seeking or reading the logging disk can be substantial.This also requires a substantial buffer pool in the servers—we use about 80 megabytes (shared between this function and normal caching of pages to avoid disk reads).

Other file system designers have explained the attractions of logging, which are numerous [7]:

- Server throughput can be increased, because the log is written sequentially without seeks; the seeks happen asynchronously when the server writes the updates to their home locations, hopefully at a time of lower overall load.

- Server availability can be increased, because restarting from a log can be faster. There is no need to have a program such as “fsck” verify file system invariants, because each log entry by definition preserves the invariants.

- Server performance can be increased, because the server can use “group commit”: write a single log record in one disk operation to describe multiple client operations.

- Atomicity can be improved, giving stronger guarantees to clients: a single log record, written in a single disk operation, can describe an update (such as “rename”) that affects several parts of the file system simultaneously. Other mechanisms can achieve this atomicity, but logging is by far the simplest.

But even without those advantages, the use of a log would be attractive because it optimizes and simplifies our replication algorithms, in the following two ways. (More information about the logging techniques used in Echo is presented in another paper [10].)

In configurations 2, 4, and 6 described earlier, the primary server has direct access to all disks, without communicating with the secondary server. Since the primary can write those disks simultaneously, the performance penalty for replicating the data is slight. But if the primary crashes, the secondary would need to read the entire log in order to reconstruct the dynamic state that allows it to interpret the rest of the disks. This could take many minutes, during which no service would be available. We can use log records to accelerate this. After the primary has written the log record for an update (and returned to the clerk) the primary can asynchronously forward the log record to the secondary. The secondary can apply these forwarded log records to its dynamic state, so that its state tracks the primary—but lagging by a few log records. So after a crash of the primary and the subsequent fail-over, the new primary (old secondary) can recover the entire dynamic state by reading and applying just those few log records. This substantially reduces fail-over time.

The second way that logging helps replication is that the log can be used to reconcile replicated disks. The basic idea here is simple. During recovery, all we need to do is propagate log records from one replica to another, or discard log records on a replica, so that the logs of all the currently available replicas become identical. The only complexity is deciding which log records to keep. There are two constraints. First, we must keep all log records corresponding to updates for which some clerk has been told that the update was committed to stable storage. Second, if some replica recorded a log record then crashed, and subsequent updates were made without considering this log record (because this replica was not available), then this log record must be discarded.

A complete description of our recovery scheme would be too complicated for this paper. The following description is a subset of the actual algorithm—it gets the correct answer, but it omits several optimizations. These optimizations are essential if the system is to perform well and have good availability characteristics. If you want more information, read our research reports about the recovery algorithm [10] and about availability [23]. These reports include the optimizations, together with more rationale and an exploration of related work by other researchers.

Our recovery scheme is driven by epoch *numbers*; the algorithm ensures that these are unique and monotonic increasing over the entire set of replicas of a box, for all time. In the following description, “committed update” means an update for which some clerk has been told that the update is safely stored in our stable storage.

Whenever a replica might have left the set of servers providing service for a box (e.g. on a cold start or when one replica crashes) we stop offering service and take the following steps:

1. Choose a primary to carry out steps 2 through 5. In general this requires an election as described earlier, but sometimes the previous primary still has a majority of votes and no new election is needed. Call the set of replicas that voted for this primary the *active replicas*. If at any time the primary is unable to contact one of the active replicas to carry out a step, or an active replica‘s vote times out, immediately stop and return to step 1.

2. Choose a new epoch number E greater than any used previously by any previous run of this step as follows:

   a. Read the *possibly chosen epoch* previously recorded in the stable storage on each of the active replicas

   b. Let E be one greater than the largest value read.

   c. Write E as *possibly chosen epoch* on each of the active replicas.

   d. If we complete these steps without failure, E is chosen; otherwise we return to step 1.

3. Reconcile the active replicas (described below). After this step all the active replicas contain identical data, apart from some optimizations described later. This identical data contains all the committed updates.

4. Write E as the *service epoch* on each of the active replicas.

5. Now we can offer service again, using the primary and active replicas determined in step 1.

In step 3, the primary determines the latest service epoch recorded on any of the active replicas. Call this epoch S. Then for each active replica R there are two possible states:

a. If R's service epoch is equal to S, then R was an active replica in an epoch S that got at least to step 4 of recovery—this is the common case. Therefore R was reconciled in step 3 of that epoch, so it has all the committed updates performed prior to epoch S. Also, R has all the committed updates performed during S. But R might also have some log records for updates that were in progress when epoch S ended. Reconciliation is just a matter of applying these additional log records to the other active replicas for epoch E, or removing them from R.

b. If R's service epoch is less than S, then R was not active in the most recent epoch, and thus might not have the most recent committed updates—this happens if R was down while other replicas were providing service. Also, R might have log records for uncommitted updates that were in progress the last time R was active, but were discarded during a later reconciliation that R did not participate in; these must be discarded. It's easy to see that at least one active replica in step 3 must be in state (a), so at worst we can recover R by whole disk copy from such a replica.

One important optimization to this algorithm is to allow for adding a witness to make the number of replicas odd. With small changes to steps 3 and 4, some replicas can be witnesses; they do not keep a copy of the replicated files and do not participate in normal service, but they do need some stable storage to record epoch numbers for use in recovery.

Another useful optimization affects replicas in state (b) during the recovery algorithm. If a replica has been off line for a long time, the amount of updating needed to bring it up to date might be substantial, and could cause an unacceptable delay in offering service. So we mark such replicas as being temporarily out of date, and offer service using the other active replicas. In effect, an out-of-date replica is demoted to the status of a witness, then promoted again when it is brought up to date. This procedure can reduce our availability (or even reliability) if another replica fails before the out-of-date replica is updated, but in return we get faster recovery and thus better availability when there is only one failure.

A third optimization—too complicated to explain here—is to use a two-phase process for moving to a new service epoch in step 4. This change speeds the recovery algorithm in certain cases by improving the handling of some uncommon failure patterns that needlessly move replicas from state (a) to state (b) in the basic algorithm.

The full algorithm is also complicated by the possibility of adding new replicas, or destroying old ones.

There is a substantial literature on replication and recovery algorithms, but surprisingly few of the results are suitable for replication in a distributed file system. One similar algorithm has been described by Kazar [11]. Many of the other algorithms either assume that network partitions cannot occur (and can give incorrect results if they do occur), or rely on 2-phase commit algorithms without exploring their failure behavior.

We repeat: the above is a much simplified description of our recovery algorithm. Out other research reports [15,23] provide the details.

HOW WELL DID WE DO?

Echo was in full service within SRC from November 1990 until the summer of 1992. It was the file system used by about 50 researchers for almost all their daily work—mail, programming and entertainment. It contained about 25 gigabytes of data, using 50 gigabytes of disk (since we had configured all our volumes with two replicas). The mechanism described for stashing data from DNS was not fully implemented—we didn't actually make calls on the DNS and didn't actually store junctions there. Instead we have manually stashed this information in the Echo name service.

The Echo filestore included a real backup system—so users stored real files in Echo. Actually we had two backup systems. One was integrated into the replication mechanism, and effectively worked as a replica that is almost always off-line. The "back-up algorithm" was just the recovery algorithm. This meant that the information stored on the backup tapes formed an instantaneous snapshot of some previous state of the volumes. Our other backup system was "tar" tapes. While they didn't form a snapshot, they were insensitive to bugs that might have cropped up in the Echo implementation. All our users are willing to believe we could get their files back from a tar tape, but it required a lot more faith to believe we could get them back from an off-line replica.

Development on Echo stopped in early 1992 when the project to port Taos, the operating system hosting Echo, to more modern hardware was canceled. Functionally it satisfied the requirements laid out in this paper. Work remained to be done if we were to achieve appropriate levels of availability and performance. Much of that work would have required faster and more reliable hardware. One of the design assumptions of Echo was that computation was cheap relative to communication. Unfortunately computation was not cheap enough in our environment of 3 MIPS processors connected with a 100

Megabit-per-second switch-based network. Current and prospective environments provide a more balanced picture: workstations at 100 MIPS, servers at 400 MIPS or more, and networks at 160 or 1000 Megabits-per-second. In that sort of environment, the Echo design would work well.

Even so, Echo's performance was comparable to a reasonable quality centralized Unix implementation on similar hardware. Its performance was better than most NFS implementations of the time—few of those could support 50 demanding users reading a single volume.

Echo's availability was reasonable, and was improving when we went out of service. We made frequent use of the fail-over mechanisms to install new server versions, and to debug servers; only the most observant clients noticed when we did that. When under active development, the availability was limited by our recently introduced bugs. At other times it was limited by a combination of flaky experimental hardware and inadequate system management. The most difficult availability problem we experienced was load control. Our servers had limited resources (128 megabytes of memory, 16 megabytes of DMA address space) and it proved remarkably difficult to avoid crashes or deadlocks caused by over-load. We got those problems under control, but we have little confidence that an additional doubling of the user community would not cause additional problems.

Echo's reliability was very good. Though during our period of service we had more than 10 disk failures, none of them caused us to lose data. (However we once lost some updates due to a software bug in the reconciliation code.)

The advent of global naming was very satisfactory. It was very pleasant to know that you could log in to any workstation in the building and see the same files. We had originally intended to integrate a remote site into the Echo name space, but this goal was never realized. Surely running over a wide area network would have caused some changes to the timeouts on the caching protocols.

The caching worked out quite well. For example, all our users shared the same repository of released code and programs. The caching was good enough that we handled the load with a single server, even though each client workstation had the same processing power as each of our file servers.

Our guarantees on write-behind order did indeed make it easy for applications to maintain their invariants without resorting to excessive "fsync" or "sync" calls. On the other hand, as discussed earlier we are still uncertain of the best design for reporting write-behind failures.

We believe that the Echo design addresses many, but not all, of the problems of scale. For example, it would probably be attractive to add some form of load-sharing for reads—so that we could handle more clients of a single volume. A scheme of automatically updated read-only replicas such as is included in AFS 4.0 would probably be satisfactory [17]. We also believe that there is a place for higher-level replication techniques to cover wide-areas, such as the *siphon* mechanism that has been described elsewhere [20]. This seems necessary because there is no design for tightly synchronized data that performs well in the presence of high latency and low reliability connections.

Technology trends continue to favor systems based on the same assumptions as Echo. Though the Echo implementation did not achieve everything we had planned for it, we think that the Echo system embodies many of the principles and techniques that will be pursued in future distributed file systems.

REFERENCES

1. Birrell, A. et al. Global authentication without global trust. *Proc. IEEE Symposium on Security and Privacy,* Oakland, 1986.
2. Birrell, A. et al. Grapevine: an exercise in distributed computing. *Comm. ACM 25,* 4 (Apr. 1982).
3. Birrell, A. Position paper for 3rd European SIGOPS Workshop. Abstracted in SigOps Review 23, 2 (April 1989). Page 16.
4. Burrows, M. *Efficient data sharing.* Ph.D. thesis, Churchill College, Cambridge, Sep. 1988.
5. Gasser, M. et al. The Digital distributed system security architecture. *Proc. 12th National Computer Security Conference,* Baltimore, 1989, 305-319.
6. Gray, C.G. and Cheriton, D.R. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *Proc. 12th Symp. on Operating Systems Principles* ACM SIGOPS, (Dec. 1989), 202-210
7. Hagmann, R. Reimplementing the Cedar file system using logging and group commit. *Proc. 11th Symp. on Operating System Principles,* ACM SigOps, Nov. 1987, 155-162.
8. Hisgen, A., et al. Availability and consistency trade-offs in the Echo distributed file system. *Proc. 2nd Workshop on Workstation Operating Systems*, IEEE, (Sep. 1989), 49-54
9. Hisgen, A., et al. Granularity and semantic level of replication in the Echo distributed file system. *Proc. Workshop on the Management of Replicated Data,* IEEE, (Nov. 1990), 2-4.
10. Hisgen, A. et al. New value logging in the Echo replicated file system. *SRC Research Report 104* Digital Equipment Corporation, Palo Alto, 1993.
11. Kazar, M. Ubik: replicated servers made easy. *Proc. 2nd Workshop on Workstation Operating Systems,* IEEE, Sep. 1989.
12. Lampson, B.W. et al. Authentication in distributed systems: theory and practice. *Proc. 12th Symp. on Operating System Principles,* ACM SigOps, Nov. 1989, 165-182.
13. Lampson, B.W. Designing a global name service. *Proc. Fifth Symposium on the Principles of Distributed Computing.* ACM, 1986.
14. Mann, T. et al. A coherent distributed file cache with directory write-behind. *SRC Research Report 103,* Digital Equipment Corporation, Palo Alto, 1993.
15. Mann, T. et al. An algorithm for data replication. *SRC Research Report 46,* Digital Equipment Corporation, Palo Alto, 1989.
16. Mockapetris, P.V. RFC1034 Domain names - concepts and facilities. (November 1987)
17. OSF, *OSF DCE 1.0 Application Development Reference, 2,* (Dec. 1991).
18. Ousterhout, J. et al. A trace-driven analysis of the Unix 4.2 BSD file system. *Proc. 10th Symp. on Operating System Principles,* ACM SigOps, Dec. 1985, 15-24.
19. Ousterhout, J. et al. The Sprite network operating system. *Computer 21,* 2 (Feb. 1988), 23-35.
20. Prusker, F.J. and Wobber, E.P. The siphon: Managing distant replicated repositories. *PRL Research Report 7,* Digital Equipment Corporation, Paris, 1991.
21. Satyanarayanan, M. The ITC distributed file system: principles and design. *Proc. 10th Symp. on Operating System Principles,* ACM SigOps, Dec. 1985.
22. Schroeder, M. and Burrows, M. Performance of Firefly RPC. *ACM Trans. Comput. Syst. 8,* 1 (February 1990).
23. Swart, G. et al. Availability in the Echo file system. *SRC Research Report 112* Digital Equipment Corporation, Palo Alto, 1993.
24 Tanenbaum, A. et al. Experiences with the Amoeba distributed operating system. *Comm. ACM, 33,* 12 (December 1990).