103

# A Coherent Distributed File Cache With Directory Write-behind

Timothy Mann, Andrew Birrell, Andy Hisgen,
Charles Jerian, and Garret Swart

June 10, 1993

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

# Systems Research Center

The charter of SRC is to advance both the state of knowledge and the state of the art in computer systems. From our establishment in 1984, we have performed basic and applied research to support Digital's business objectives. Our current work includes exploring distributed personal computing on multiple platforms, networking, programming technology, system modelling and management techniques, and selected applications.

Our strategy is to test the technical and practical value of our ideas by building hardware and software prototypes and using them as daily tools. Interesting systems are too complex to be evaluated solely in the abstract; extended use allows us to investigate their properties in depth. This experience is useful in the short term in refining our designs, and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this strategy, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical flavor. Some of it is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. Other work explores new ground motivated by problems that arise in our systems research.

We have a strong commitment to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Robert W. Taylor, Director

# A Coherent Distributed File Cache With Directory Write-behind

Timothy Mann, Andrew Birrell, Andy Hisgen,
Charles Jerian, and Garret Swart

June 10, 1993

**Abstract**

Extensive caching is a key feature of the Echo distributed file system. Echo client machines maintain coherent caches of file and directory data and properties, with write-behind (delayed write-back) of *all* cached information. Echo specifies ordering constraints on this write-behind, enabling applications to store and maintain consistent data structures in the file system even when crashes or network faults prevent some writes from being completed. In this paper we describe the Echo cache's coherence and ordering semantics, show how they can improve the performance and consistency of applications, and explain how they are implemented. We also discuss the general problem of reliably notifying applications and users when write-behind is lost; we addressed this problem as part of the Echo design but did not find a fully satisfactory solution.

# Contents

# 1 Introduction

Echo is a distributed file system that incorporates replication, caching, global naming, and distributed security. Figure 1 gives a block diagram of the Echo system.[1]

- **Replication.** Echo replicates servers for availability and disks for data integrity, allowing a wide range of configurations and tolerating both server crashes and network faults. The interconnections between disks, servers, and client machines can be replicated as well.

- **Caching.** Echo client machines keep coherent write-back caches of data and properties for both files and directories in volatile memory, thereby reducing the latency seen by applications on file system operations and reducing the read load and peak write load on servers. The caches use *ordered write-behind;* that is, updates buffered in the cache are automatically flushed after a time delay and are written to server disks in a well-defined partial order, thus limiting the damage that can occur when a client machine crashes and its volatile memory is erased. We use the term *clerk* for the module in the client operating system that performs these functions.

- **Global naming.** Echo supports a globally scalable, hierarchical name space. The upper levels of the hierarchy are implemented using a replicated name service that achieves very high availability with loose consistency semantics, while the lower levels are implemented by the file system with tight consistency but somewhat lower availability.

- **Distributed security.** Echo has adopted a security model in which servers do not trust client machines; instead, client machines use a cryptographic protocol to authenticate themselves as acting on behalf of the users who have logged in to them [16].

This paper concentrates on Echo's client cache. Separate papers give a complete overview of Echo [4] and discuss various other aspects in detail [9, 10, 11, 12, 18, 25].

In the next section we discuss the motivation for Echo's cache design; then in Section 3 we present the cache's coherence and ordering semantics and the facilities for reporting lost write-behind. We show how these semantics are useful

[1] Although Echo is no longer under development or in use, to avoid awkwardness we speak of it in the present tense throughout this paper.

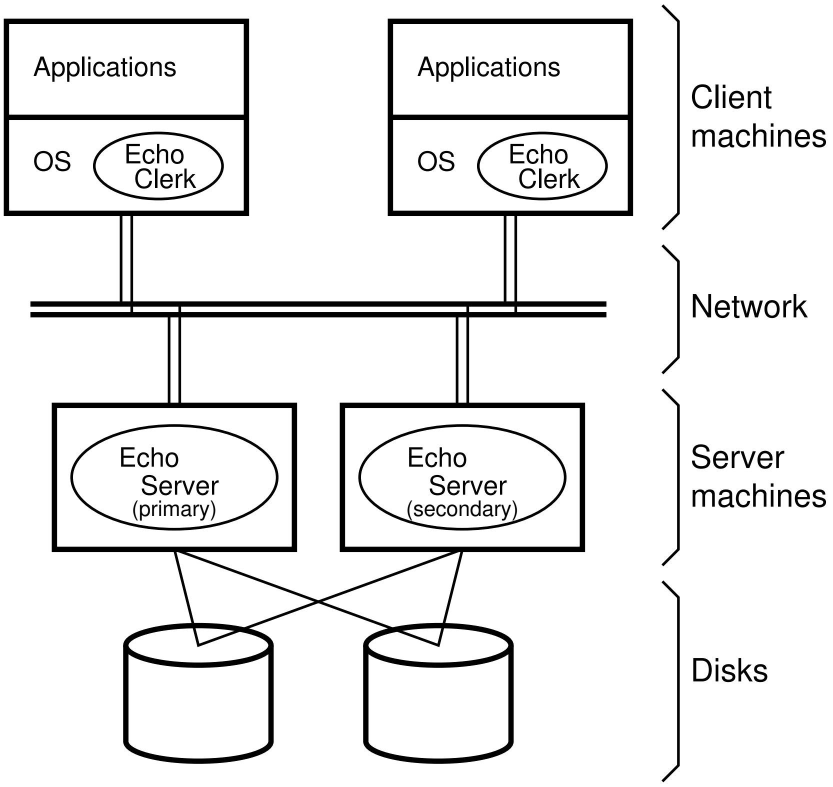


Figure 1: Block diagram of the Echo system.

to some applications in Section 4, and how directory write-behind can improve the performance of some applications in Section 5. In Section 6 we give details of the cache implementation. Section 7 compares the Echo cache with related work, while Section 8 summarizes our conclusions.

## 2 Design motivation

In this section we discuss four goals that motivated the Echo cache design and explain how they shaped the resulting system. We wanted the Echo distributed file system to meet or exceed the standards set by single-machine file systems in performance, semantics, fault tolerance, and security.

Echo's *performance* goal was to provide service comparable to or better than a single-machine file system on similar hardware. We strove not only for high throughput, but also for low latency on individual operations. We also wanted to maximize the number of client machines that could be handled per server.

This ambitious goal led us to an aggressive cache design. Most of our workstations dedicate more than 16 megabytes of main memory to the cache. Echo clerks cache both files and directories, so that users see good performance not only when reading file data, but also when opening files and listing directories. Clerks do write-behind on both files and directories, so that users see low latency not only when writing to files, but also when creating, deleting, renaming, or changing access permissions on files and directories. Write-behind also opens up the opportunity to reduce the load on servers (thus increasing their capacity) by optimizing out sequences of operations that cancel before they reach the server—for example, writing the same file page repeatedly, or creating and deleting a temporary file—though we have not gone far in implementing such optimizations.

Echo's *semantic* goal was to present a single-system view to applications. That is, so far as possible, the distributed file system should present the same interface and semantics to an application program as does a single-machine file system. Even an application distributed across many machines should see one consistent file system, not many slightly inconsistent ones.

This goal led us to implement a file cache with strict coherence and a replication scheme with strict consistency between copies. We chose not to explore the alternative approach used in the LOCUS [20, 27] and Coda [15] file systems, in which replicas or cached copies of files are allowed to diverge during periods when the network is not fully connected. Instead, we forbid both write/write conflicts (where two clients make different changes to copies of the same file), and read/write conflicts (in which one client changes a copy of a file while another client continues to read the old version).

Our desire to present a single-system image also led us to emulate a single-machine Unix[2] file system more carefully than some other distributed file systems have done. For example, unlike NFS [21, 24], Echo keeps a file that has been unlinked from the name space in existence as long as any application program has it open. Also unlike NFS, an Echo clerk allows applications to write data into its cache only if it knows there will be disk space available on the server to hold the data when the cache is flushed.

Echo's *fault-tolerance* goal was to mask faults in all system components wherever feasible, and to fail as cleanly as possible when faults occur that cannot be masked. Fault tolerance is becoming increasingly important in distributed systems

[2]Unix is a trademark of Unix Systems Laboratories

as they are built from more and more individual machines, each of which can fail independently.

The most visible fault-tolerance feature in Echo is the replication of servers and disks, which we do not discuss in this paper, but fault-tolerance considerations affected our cache design as well. The token directory that is needed to maintain cache coherence is replicated in the main memory of two different servers, so that the failure of one server will not invalidate client caches and force their write-behind to be discarded. If a server loses touch with a clerk, the server does not revoke the clerk's tokens until an agreed-upon timeout (or *lease* [8]) has expired. Therefore, two applications that read from different caches at the same time can never see inconsistent data, even if one is running on a machine that has been partitioned away from the rest of the network. To provide clean failure semantics, we guarantee that write-behind will reach disk in a partial order known to applications, and we allow applications to add their own constraints to this partial order. When we must discard write-behind, we give any application that may have read or written the discarded data an error return on any further reads or writes it attempts, thus halting its progress before it can observe the anomaly.

Echo's *security* goal was to protect the privacy and integrity of stored information without requiring all machines in the distributed system to trust all others, and without compromising the system's performance.

This goal led us to place a trust boundary between servers and clerks. Servers do not trust the operating systems on client machines to provide proper security. Instead, clerks must authenticate themselves to servers as acting on behalf of particular users, using a cryptographic protocol. An authenticated clerk is given only the privileges of the user it is acting for; it cannot touch data that the user is not permitted to access, and it cannot corrupt data structures in the file system implementation. Because access checks can be expensive, we were also led to develop a form of access check caching—an Echo server needs to do access checking only when a clerk requests a cache coherence token, not on every read or write. This security machinery was easy to add to the system during the design phase, but would have been hard to retrofit had we chosen to omit security from the initial design.

## 3 Coherence and ordering semantics

Echo is *single-copy equivalent;* that is, at any moment, the file system data is in a single, well-defined state. An operation that changes this state is called a *write.* An operation that returns information about the state without changing it is called a *read.* Each operation is *logically performed* (carried out) at a distinct point in real time, reading or changing the state as it exists at that moment. (We take the view that every operation is logically performed at a different time because we want these times to define a total order on the operations.) If an application issues a system call requesting an operation at time $t_1$ and the call returns at time $t_2$, the operation will have been logically performed at some time $t$ with $t_1 \leq t \leq t_2$.

Moreover, in the absence of faults, Echo caching is *transparent;* that is, if no network faults or machine crashes occur, nothing other than a write operation changes the file system state.

When faults do occur, Echo remains single-copy equivalent, but the caching is no longer fully transparent. A fault can change the state of the file system data by causing updates that were written behind to be discarded before they reach disk. Such writes are logically *undone* at a unique point in real time, moving the file system data into a new state that is reflected in all reads and writes performed after that point.

These semantics are the same as those of a single-machine file system with write-behind, but differ from those of NFS, in which caches are incoherent. With NFS, a process on one machine can see old data when reading a file if a process on another machine has the file open for writing, and can see old data in a directory if a process on another machine has modified the directory recently.

The Echo cache remains single-copy equivalent even if network faults cause one or more machines to be partitioned away from the rest of the system. If a partition (or server crash) prevents a clerk from accessing the file system's current state to perform an application read or write request, the clerk either blocks waiting for the state to become accessible or returns an error indication. (These two options are similar respectively to the *hard* and *soft* options in mounting an NFS volume.) In our environment, we found it most practical to have the clerks block such operations for a limited time (up to two minutes), then give up and return an error indication if the problem remains. This policy works well in Echo because Echo's fault-tolerance features correct many problems automatically in well under two minutes. If a problem is still present after two minutes, it is likely to be the kind that needs to be fixed manually, which may take a long time. So it seems best to unblock applications and give them error returns at that point.

## 3.1 Ordering constraints

Because crashes and network faults can cause write-behind to be discarded, the order in which writes reach disk is important. If Echo allowed writes to reach disk in an arbitrary order, applications that store mutable data in files could find their data inconsistent after a crash in which write-behind is lost. Therefore Echo specifies a partial order on writes, and guarantees that the actual order in which writes reach disk will be consistent with this partial order. Echo also provides a primitive that lets applications augment the partial order with additional constraints. With a knowledge of Echo's ordering guarantees, a carefully coded application can assure itself that the data structures it stores in the file system will remain consistent even when some write-behind is lost in a crash. Section 4 gives some examples; the remainder of this section describes the constraints themselves. Section 7 includes a brief comparison of this approach to fault tolerance with the alternative of providing atomic transactions in the file system interface.

We say that a write is *stable* when it has reached disk; we say that a write is *discarded* when it is logically undone and will never reach disk. All writes are eventually either stable or discarded (but not both). We call a write *unstable* when it has been logically performed but is not yet either stable or discarded.

Stated informally, Echo's guarantees are as follows:

A. If a write is requested by one client machine and the results are observed by another, the write is stable.

B. Writes to a given object become stable in the same order they are logically performed, except that an unbroken sequence of overwrites requested by a single client may be reordered. (An *overwrite* is a write operation to a file that does not change its length.)

C. Writes are stable when they are forced to disk by *fsync*.[3]

To state Echo's stability ordering constraints formally, we define two relations, $\rightarrow$ and $\Rightarrow$. Intuitively, the relation $\rightarrow$ expresses data dependency; two operations are related by $\rightarrow$ if and only if the first could have affected the result of the second. The relation $\Rightarrow$ is the partial order in which writes are guaranteed to reach disk. Viewed as a set of ordered pairs, the relation $\Rightarrow$ is a subset of $\rightarrow$. The formal definitions of $\rightarrow$ and $\Rightarrow$ are as follows:

1. If $o_1$ and $o_2$ are two operations on the file system, we say $o_1 \rightarrow o_2$ if

    - $o_1$ is a write operation.
    - $o_1$ and $o_2$ have an operand in common,

[3]When the Unix system call *fsync(f)* returns, all writes to file $f$ that returned before *fsync* was called are guaranteed to be on disk. Echo allows *fsync* on directories as well as on ordinary files.

- $o_1$ and $o_2$ both return successfully,[4]
- $o_1$ was logically performed before $o_2$, and
- $o_1$ was not already discarded when $o_2$ was logically performed.

2. The $\rightarrow$ relation is transitive; if $o_1 \rightarrow o_2$ and $o_2 \rightarrow o_3$, then $o_1 \rightarrow o_3$.

3. We say $o_1 \Rightarrow o_2$ if

   - $o_1 \rightarrow o_2$ and
   - $o_1$ and $o_2$ are both write operations, but they are not both overwrites (data writes to a file that do not change its length).

4. The $\Rightarrow$ relation is transitive; if $o_1 \Rightarrow o_2$ and $o_2 \Rightarrow o_3$, then $o_1 \Rightarrow o_3$.

5. If $o_1 \Rightarrow o_2$, and $o_1$ is now discarded, then $o_2$ is now discarded.

6. If $o_1 \rightarrow o_2$, and $o_1, o_2$ were requested by applications running on different client machines, then when $o_2$ is logically performed, $o_1$ is stable. (This corresponds to informal guarantee A above.)

7. If $o_1 \Rightarrow o_2$ and $o_2$ is now stable, then $o_1$ is now stable. (Informal guarantee B.)

8. If the write operation *fsync* returns successfully, then it is stable. (Informal guarantee C.)

Echo's read and write operations include most of the usual Unix file system operations, plus a new operation for adding constraints (described below).

Read operations include getting the properties of a file or directory (the Unix *stat* system call), opening a file, reading file data, listing a directory, looking up a pathname component in a directory, and so forth. System calls that involve looking up a pathname are viewed formally as several operations that are logically performed in sequence, consisting of a component lookup in each directory along the path, followed by an operation on the final object found. All read operations have just one operand.

Write operations include writing file data, creating a file or directory, renaming, *fsync,* and so forth. We view *fsync* as a write operation because of the role it plays in constraining the order in which other operations are made stable, even though it

[4] We say an operation *returns successfully* if it returns control to its caller without indicating an error. Write-behind or other work queued by the operation may still fail later.

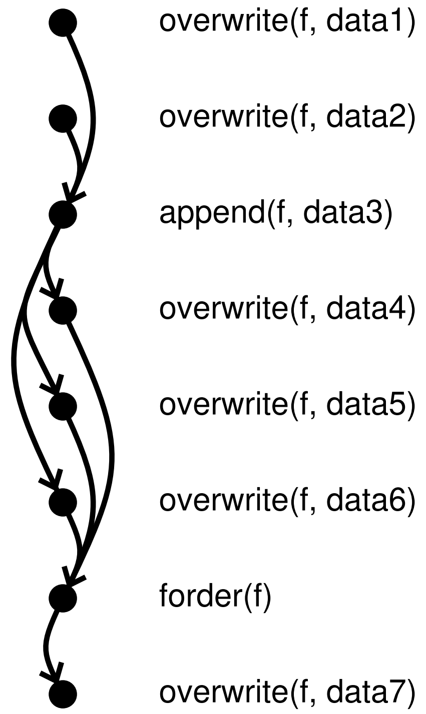


Figure 2: The $\Rightarrow$ relation for writes to a single file.

does no writing of its own. A file data write that both overwrites existing data and appends new data is viewed formally as two operations, a pure overwrite logically followed by an append. Also, a file data overwrite is guaranteed to be failure-atomic only if it modifies bytes in at most one block of the file, where the block size is a parameter set by the Echo implementation—currently 1024 bytes. An overwrite that does not meet this criterion may be implemented as an arbitrary sequence of shorter overwrite operations, where each of the shorter overwrites is failure-atomic, but the sequence as a whole is not failure-atomic nor even ordered. All other write operations are failure-atomic. Write operations may have multiple operands; for example, if we rename a file from `/a/f` to `/b/f` when some other file named `/b/f` already exists, this operation modifies the old parent `/a`, the new parent `/b`, the file `/a/f` being renamed, and the old file `/b/f` being displaced, for a total of four operands.

Echo adds one more write operation to the usual set, called *forder*. Like *fsync,* the *forder* operation does not modify any of its operands but does follow the stability

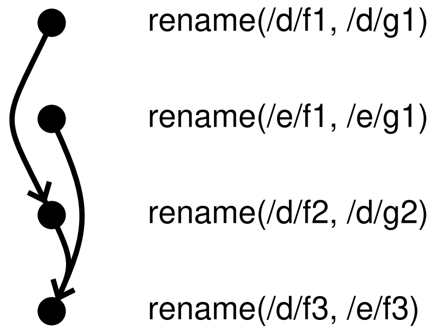


Figure 3: The $\Rightarrow$ relation for renames.

rules for write operations. Unlike *fsync,* however, *forder* returns immediately, not waiting for the operations ordered before it to become stable. The *forder* operation is useful for adding constraints to the order in which other writes are made stable, without delaying its caller as *fsync* does; we give some examples in Section 4.

Figures 2 and 3 illustrate some of the ordering constraints.

In Figure 2, a series of write operations is applied to a single file `f`. All the operations are ordered by $\rightarrow$, but only those joined by arrows in the figure are ordered by $\Rightarrow$. First, two different records within the file are overwritten; these writes are not ordered by $\Rightarrow$. Next, a new record is appended; this write is ordered after both of the overwrites. Three more overwrites occur next; these are ordered after the append but not amongst themselves. Finally, there is an *forder* call followed by another overwrite. The *forder* itself does not change the file in any way, but it causes the final overwrite to be ordered after the earlier ones.

In Figure 3, several files are renamed. The first two renames are not ordered because they affect different files in different directories—they have no operands in common. The third rename follows the first because both affect directory `/d`. The fourth rename follows all the others because it affects both directories `/d` and `/e`.

Echo limits the scope of its ordering constraints by requiring all the operands of a given operation to be in the same *volume.* An Echo volume is roughly similar to a Unix "file system"—it is a subtree of the global hierarchical name space and is stored on a single (possibly replicated) set of disks and managed by a single (possibly replicated) server. Echo volumes are tied together by *junctions,* which differ from Unix "mount points" in that they are stored stably in the joined volumes and are global to all clients of the file system, rather than being established individually (and perhaps differently) by each client machine at run time. Like Unix, Echo does not permit files to be renamed or hard-linked from one volume to another. It also does not permit *forder* operations whose operands are not all in the same volume. This restriction greatly simplifies the Echo implementation and Echo's interfaces for reporting discarded write-behind (discussed in the next section), but it requires applications to keep their file-based data structures within a single volume or to take extra care when cross-volume dependencies arise.

### 3.2 Reporting lost write-behind

In section 3.1, we said that a carefully coded application can take advantage of Echo's ordering guarantees to ensure that its file-based data structures will remain consistent even when write-behind is lost "in a crash," thus suggesting that applications can deal with lost write-behind by restarting and perhaps invoking crash recovery. But not all lost write-behind in Echo is caused by crashes. If a network fault cuts off communication between a clerk and a server, the server revokes the clerk's cache coherence tokens in order to allow other clerks to proceed. This forces the clerk to discard its write-behind in order to maintain cache coherence, but does not halt programs running on the clerk's machine. (Write-behind is also discarded if a double server crash causes the entire token directory to be lost.) By contrast, NFS and single-machine Unix discard write-behind only if the machine holding the write-behind crashes, which of course halts all applications running on that machine. Thus Echo applications have an additional problem to deal with—finding out when writes they depend on have been discarded and taking appropriate action. Echo provides some facilities to help; we describe them next.

To make this discussion more precise, we say that an application process $P$ *depends on* a write $w$ if $P$ issued the write, or if $P$ has done a read or write operation $o$ such that $w \to o$. In these cases (and in no others), $P$ has directly observed the effect of $w$ on the file system's state.[5] Thus if $P$ does not expect other processes to be changing the part of the file system that $P$ is using, discarding $w$ will make $P$'s internal state inconsistent with that of the file system. If $P$ continues running after $w$ is discarded and reads from the file system, it may observe the inconsistency and be confused; if it writes, it may write data that is inconsistent with the new file system state.

To deal with this problem, Echo normally halts the progress of each process that depends on discarded write-behind on a given volume by giving an error return on any further operations the process tries to invoke on that volume. A process that receives such an error return should immediately abort with an error message,

[5]Other processes may have observed the effect of $w$ indirectly—for example, they may have been told about it by $P$—but the system cannot detect this. In general, it is not safe for a process to communicate information about the file system's state outside itself until that information is stable.

effectively converting the error to a "crash." (We considered also sending the affected processes an asynchronous Unix signal, so that processes that access the file system infrequently would halt more quickly; this seems like a good idea, but we did not find time to try it.) Processes are notified of lost write-behind in this way if they are in *standard recovery mode;* we discuss two alternative modes next.

If a process is in *self-recovery mode,* and write-behind that it depends on is lost, all its open files in the affected volume are marked so that new operations attempted on them will give an error return. The process's working directory is also marked in this way if it is in the affected volume, so that new operations that specify relative pathnames will give an error return. But operations that specify absolute pathnames (including changing the working directory) are allowed without restriction. Our vision was that a process could use this mode by making all of its file references (after initialization) through open files and relative pathnames, until it received its first error return reporting lost write-behind. At that point the process would run its own recovery code, which would reopen its files and working directory using absolute pathnames.

Self-recovery mode turned out not to be very useful in practice. Existing programs make no distinction in their usage of absolute and relative pathnames, so they generally do not behave reasonably in this mode—either they use absolute pathnames in unfortunate places and thus fail to notice lost write-behind, or they use relative pathnames everywhere and thus fail to recover. Moreover, an application that maintains state in the file system and wants to recover from lost write-behind errors is easily structured as two processes—a child process that does all the real work and that halts when lost write-behind is detected, plus a parent process that restarts the child after each such halt. The child runs the same recovery code regardless of whether it was restarted due to lost write-behind or due to a machine crash and reboot. Therefore self-recovery mode was not useful to newly written programs either.[6]

A different recovery mode, which we might call *null recovery mode,* would have been useful for interactive shells. In this mode, if write-behind that a process

[6]Self-recovery mode is ugly largely because it is an attempt to fit a new concept into an existing Unix-like file system interface. Had we been free to change this interface, we might have adopted a cleaner approach to self-recovery using explicit *failure handles.* In this approach, each read or write operation accepts a failure handle as an additional parameter. The semantics of lost write-behind reporting are changed to replace *process* with *failure handle* in the concept of dependency on writes—if a write $w$ is issued using failure handle $h$, or an operation $o$ is issued using $h$ and $w \rightarrow o$, then $h$ depends on $w$. If a write is discarded, subsequent operations issued using any failure handle that depends on that write will give an error return. Standard recovery mode then becomes a special case in which a process chooses a new failure handle on its first access to each volume and uses it for all subsequent accesses.

depends on is lost, all the process's open files in the affected volume are marked so that new operations attempted on them give an error return. But new files can be opened by name without restriction, and the working directory remains valid. This mode would be useful for interactive Unix shells because they do not keep files open for long and do not remember file system state. An unmodified Unix shell, which simply prints a message and continues when it receives an error return reporting lost write-behind, would work nicely in null recovery mode. Unfortunately, we did not implement null recovery mode in Echo, so we ran shells in self-recovery mode instead. (In fact, for historical reasons, we ran most programs in self-recovery mode, which was clearly a mistake.) Users did not like the results—they were confused, not enlightened, when lost write-behind caused a shell to "forget" its working directory. But running shells in standard recovery mode would have been even worse—users would have been quite unhappy if their shells had crashed or lost all access to the file system whenever write-behind was lost.

The Echo file system interface allows applications to obtain locks on files. Echo provides Berkeley-style advisory locks, and also uses a form of internal lock to implement the Unix feature that keeps a file from being deleted as long as at least one process has it open, even if it is removed from the name space. In the implementation, such locks are obtained and cached in much the same way as file data. The locks are discarded along with write-behind when a client machine crashes or has its cache coherence tokens revoked by the server due to a network fault. Thus when a client machine crashes, its locks are released, allowing other machines to obtain them and make progress. Locking operations do not participate in the $\rightarrow$ or $\Rightarrow$ relations, but lock acquisitions do count as file accesses for the purpose of lost write-behind reporting. That is, if a process depends on lost write-behind on a volume, it receives error returns on attempts to acquire new locks in the volume. Conversely, if a lock on a given open file is discarded, all further attempts to read or write it receive error returns; there is no timing window during which the lock is released but processes can still read or write the file.

When write-behind is discarded, besides notifying the affected processes, it is also a good idea to notify the user directly. Programs that have not been specially coded to deal with the possibility of lost write-behind may not handle it cleanly. For example, a process that writes to the file system will usually exit without calling *fsync* to make its changes stable, so if those changes are discarded later, there is no process left to notify of the error.[7] In other cases, a process that ought to abort

[7] Because of this problem, we considered changing the semantics of process exit to include calling *fsync* on all the files and directories that the process modified; however, we judged that the additional safety would not be worth the performance penalty. Also, most existing programs are not prepared to handle an error return when they attempt to exit!

when write-behind is lost may instead print a message and continue, or may simply ignore the error returns. Unfortunately, it is hard to notify users about discarded write-behind in a way that makes sense to them. Echo prints a console message when this happens, but users find these messages cryptic—or miss them entirely because the console window is iconized. Worse, we have no way at all of notifying a user when write-behind is lost because his machine crashed. (Single-machine Unix and NFS also have this problem.) We have no really good ideas for improving this situation; inherently, when we accept write-behind and store it unstably in client machine memory, we violate the simple abstraction of a stable, on-disk file system that programs and users expect to see. We can ameliorate the problems this causes, by providing ordering guarantees and notification of discarded write-behind, but we cannot eliminate them.

# 4 Using the semantics

In this section we describe some ways in which applications can make use of Echo's caching and ordering semantics.

First and most important, a file system with coherent caching is much easier to use when writing a distributed application than one with incoherent caching. For example, suppose you would like to build a distributed, parallel *make;* that is, a tool that speeds up the recompilation of large programs by farming out the compilation of individual source files to different machines, then linking the results together on one machine. With a coherent distributed file system, such a tool can work much like an ordinary, nondistributed *make*—the compilation steps can simply write their results into the file system, and the link step can simply read them, knowing it will get the correct data. With a system like NFS that has incoherent caches, things are not so simple. Although recent implementations of NFS give *close-to-open coherence,* meaning that once a file is closed, readers that open it later will get the correct data, NFS does not provide coherence on directories. So in this example, when the linker tries to open the files that the compilers have written, it may not find them all in its cached copy of the directory. Perhaps this example sounds contrived, but we are aware of real-world instances of the same phenomenon. Our colleagues in the Hector lexicography project [7] tried to use NFS to maintain a shared file of dictionary definitions being read and updated by multiple lexicographers. After running into intractable bugs caused by NFS's incoherent caching, they were forced to rewrite their code to get the data from a centralized server instead of going directly to the file system. Their original design would have worked fine with Echo.

Echo guarantees that if data is written on one machine and read on another, that data is stable. (This is a consequence of the rules pertaining to the $\rightarrow$ relation given above.) Thus a distributed application whose component processes communicate only through the file system can be sure that when one Echo clerk's write-behind is lost, only the processes running on that clerk's machine will be affected. The lost data will not have propagated to processes on other machines. The effect will be much the same as if the affected machine crashed.[8]

Making use of Echo's ordering semantics is more difficult than making use of its cache coherence, but the effort is worthwhile for applications that store data structures in the file system and want to make sure these structures do not become corrupt if write-behind is lost. One way to use the semantics is as follows. First, carefully order the application's file system write calls so that if a crash halts the application at any point, the data structures it has written will be consistent (or at worst automatically repairable) when it is restarted. This first step produces an application that would be robust against crashes if run on a file system with no write-behind. Second, check whether Echo's ordering constraints forbid all reorderings of writes that would leave data structures in an inconsistent (or unrepairable) state. If not, add *forder* calls to the application to forbid the unwanted reorderings. In the worst case, this can always be done by adding enough *forder* calls to forbid any reordering at all. A few examples follow.

Figure 4 gives a simple example of writing a file, then replacing it atomically with a new version. The arrows in the figure show which operations are related by $\Rightarrow$ (omitting arrows that can be inferred from the transitivity of $\Rightarrow$). In this example, Echo's built-in ordering constraints provide exactly what is needed, with no calls to *forder*. First, we create a file `/d/f` and write some data to it. Next, we create a file `/d/f.new`, which is intended to be a new version of `/d/f`, and write some new data to it. Finally, we rename `/d/f.new` to replace `/d/f`. Because the rename is a write operation on both the new file (changing its name) and the old file (deleting it), it is ordered by $\Rightarrow$ after all the other operations. Therefore, we can be assured that even if some write-behind is lost, the new file will replace the old one only if its intended contents have reached disk.

An ordinary Unix file system does not provide this guarantee. In most Unix systems, directory operations are write-through while file data is write-behind, so chances are excellent that `/d/f.new` will be renamed before its contents have reached disk. If there is a crash before the contents reach disk, upon reboot the

[8] Providing this guarantee exacts a certain cost—applications that communicate across machines through the file system can do so only at disk speed, not at network speed. Note that NFS effectively provides this guarantee as well, at the same cost.

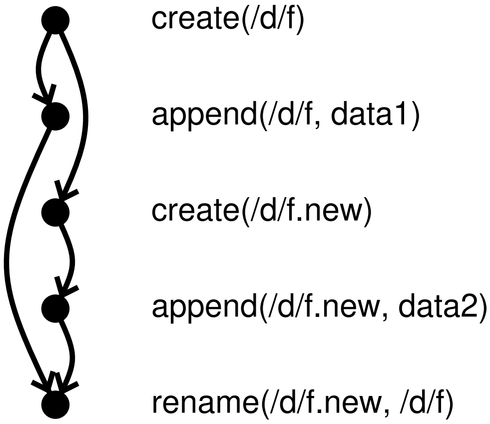


Figure 4: Replacing a file with a new version.

name `/d/f` will point to a garbage file and the name's old referent will have been destroyed.

Figure 5 gives an example of replacing a directory `/d` with a new version. The left-hand column shows an initial attempt at coding the procedure. First, we create the new directory `/d.new` and write two files into it, `/d.new/f1` and `/d.new/f2`. Then we rename the old version of `/d` to `/d.old` and rename the new version to `/d`. Finally, we can remove `/d.old` and its contents (not shown). If this procedure is interrupted by a crash, we do a small amount of recovery upon restart—renaming `/d.old` back to `/d` if `/d` does not exist, and then removing `/d.new` or `/d.old` if they exist.

This initial attempt would work if write-behind could not be reordered—after recovery, `/d` would be a complete copy of either the old version or the new version—but reordering introduces a problem. Because the final rename operation does *not* have any of the new files in `/d.new` as operands, the $\Rightarrow$ relation does not order the data writes to those files before the rename. (The file creations are ordered because they have `/d.new` itself as an operand, but the data writes do not.) So when the rename reaches disk, there is no guarantee that the file data is on disk, and an inopportune crash or network fault could leave `/d` as a directory full of garbage files.

This problem is easily fixed by inserting an *forder* call, as shown in gray in the right-hand column of Figure 5. The *forder* call establishes a synchronization barrier, such that all operations on `/d.new`, `/d.new/f1`, or `/d.new/f2` that were logically performed before the *forder* are ordered by $\Rightarrow$ before any operations

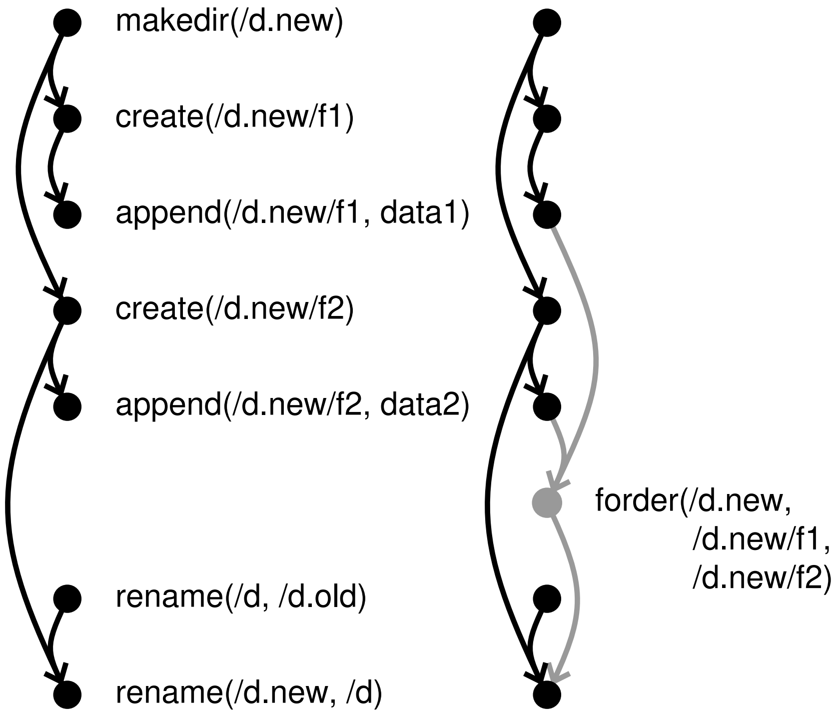


Figure 5: Replacing a directory with a new version.

on these operands that are logically performed later. In particular, the three ordering arrows shown in gray ensure that the contents of the files reach disk before the final rename operation.

Applications that make complex changes to data structures stored in the file system may use write-ahead logging. The application first appends an intentions record to a log file, then proceeds to make its changes in any order. When the application is restarted after a crash, it reads the log file and redoes the changes recorded in the log. From time to time the application reclaims log space (and speeds up the next crash recovery) by trimming off a prefix of the log, after making sure that the changes up to that point have reached disk.

Figure 6 shows how Echo's ordering constraints can be used to improve the performance of write-ahead logging. With a conventional Unix file system, the only way to be sure that a log record reaches disk before the changes it describes is to call *fsync* on the log file after writing the record and before making the changes. Doing this slows down the update by introducing an additional wait for the disk.

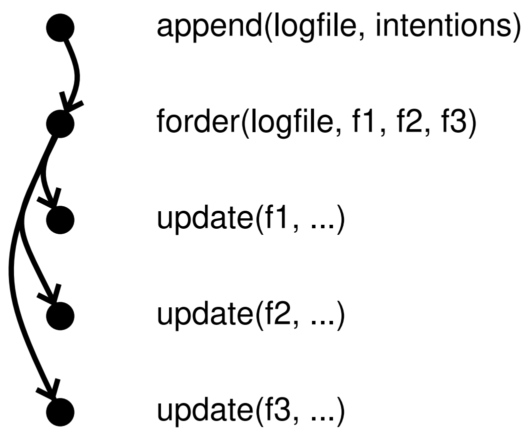


Figure 6: Write-ahead logging.

Although an application using write-ahead logging generally does need to force its log to disk at certain *commit points*—points where the application reports to the user that the updates he requested are stable—there are typically many writes to the log between commit points, so one does not want to force the log after every log write. With Echo, one can use *forder* in place of *fsync*, eliminating the need to wait. In the figure, after writing the log record, we issue an *forder* whose operands are the log file and each of the files or directories that are to be updated. This *forder* ensures that none of the updates will reach disk before the log record does. When the application does need to force the log at a commit point, it simply calls *fsync* on the log file.

This example is incomplete. We could elaborate it to include a means for trimming the log, perhaps using *forder* here too so that we could trim the log without first forcing updates to disk. This is possible, but more complex, so we will not go into the details in this paper.

Echo's ordering constraints make *fsync* useful not only to ensure that previously written data is stable, but also that previously *read* data is stable. Figure 7 illustrates this. First, one application writes some data to file `f`; later, a second application does a read that returns this data. Suppose the second application wants to make sure the data is stable before printing results, aborting instead if the data is discarded. It can do so by calling *fsync* on `f` at some point after the read and aborting if the call gives an error return. Why does this work? Because the read returned the written data, the read is ordered after the write by $\rightarrow$, as shown by the broken arrow in the figure. Therefore the application process *depends on* the write in the sense defined

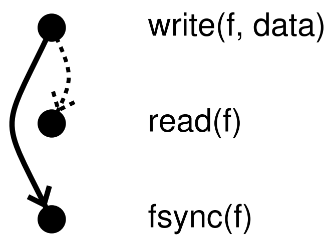


Figure 7: Checking that data read is stable.

in Section 3.2 above, so Echo's lost write-behind reporting rules guarantee that if the write was already discarded when the *fsync* is logically performed, the *fsync* call will give an error return. Hence if the *fsync* call returns without error, the write must not have been discarded at the time the *fsync* was performed, and so by Echo's ordering rules, the *fsync* is ordered after the write by $\Rightarrow$ (as shown by the solid arrow in the figure) and the write is stable.

Echo's advisory locks provide a convenient building block for implementing other replicated services on top of the Echo replicated file system. One technique is to store the service's data in an Echo volume and provide access to it through a pair of servers. In normal operation, one server acts as the primary and holds an exclusive advisory lock on some agreed-upon file in the volume. The second server acts as a backup, and is blocked waiting to acquire its own exclusive lock on the same file. If the primary server crashes or is partitioned away from the file system, Echo revokes its lock. As a result, the backup server acquires the lock and begins running as the new primary; it can then start a new backup if desired. Because Echo's lost write-behind reporting removes the old primary's ability to access the shared volume at the same time it revokes the lock, the new primary can be certain that the old one is no longer modifying the shared data when it takes over. Conversely, an existing primary can be sure that no new primary has taken over as long as it is still able to access the shared data.

A number of the techniques for using Echo's ordering constraints that we have just described have been used in real applications. In particular, the Vesta software configuration management system [6, 17] developed by colleagues at our research center uses variants of both the file and directory replacement techniques. (Vesta uses the techniques to atomically create new files and directories full of files, not to replace existing ones.) Vesta could also have used the write-ahead logging technique, but unfortunately the Vesta procedures that need to do logging involve updates to more than one Echo volume.

| | mean time (20 runs) | range | relative time |
|---|---|---|---|
| Full write-behind | 240 sec | +12/-4 | 1.00 |
| Directory write-through | 341 sec | +18/-6 | 1.42 |
| Write-back on close | 366 sec | +10/-7 | 1.52 |

Table 1: Elapsed time to compile 100 one-line C programs.

# 5 Performance

In this section we show how Echo's directory write-behind can improve the performance of some applications by giving benchmark measurements. The benchmarks include compiling 100 one-line C programs, running the *make* phase of the Andrew benchmark [13], creating a new version of a software package in the Vesta system, and some simple loops creating and deleting files and directories. The benchmarks all run considerably faster with Echo's directory write-behind enabled than with it disabled. One benchmark also shows Echo with write-behind providing better performance on directory operations than a production-quality Unix system, even though Echo is much slower without write-behind.

These benchmarks measure the kind of file system performance that an ordinary application observes. Most applications that write data to files on Unix-like systems do not concern themselves with when the data actually reaches disk; they are quite satisfied to report success and exit as soon as the system has their data buffered in memory for write-behind. Hence, except where specifically stated, the running times measured in these benchmarks do not include the time to flush writes to disk with *sync* or *fsync*.

In all the benchmarks discussed below, the Echo clerk and the benchmark programs themselves were running on a 4-processor Firefly [26], with each processor a 3 MIPS MicroVAX chip. The Echo servers were running on 6-processor Fireflies and used DEC RA-90 disks. Client and server machines were connected by the 100 Mbit/sec AN1 network [23]. The one Unix benchmark was run on a 1-processor VAX, again using the 3 MIPS MicroVAX chip, but with DEC RA-82 disks.

Table 1 shows the results of a simple compilation benchmark. One hundred C source files, each containing a single one-line function, were compiled under the control of the Unix *make* program. Thus during each run, *make* tested for the existence of 100 object files and created 100 processes running the C compiler. Each C compiler instance read one source file and wrote one object file. The table gives

| | mean time (20 runs) | range | relative time |
|---|---|---|---|
| Full write-behind | 546 sec | +6/-4 | 1.00 |
| Directory write-through | 566 sec | +5/-5 | 1.04 |
| Write-back on close | 592 sec | +32/-8 | 1.08 |

Table 2: Elapsed time for the *make* phase of the Andrew benchmark.

the average elapsed time for three different versions of the benchmark. The first line of the table shows the time to run the benchmark in Echo's normal configuration, with write-behind for both files and directories. The second line shows the running time when the Echo clerk was modified to write through changes to directories, but to continue to write behind changes to file data. (In this benchmark, file creation was the only operation involving a directory change.) The benchmark took 42% longer to run in this case. The third line shows the running time when the clerk was forced both to write through directory changes and also to write back changes to file data whenever a file was closed. In this case, the benchmark took 52% longer to run.

Table 2 shows the results of a similar benchmark, but with a more realistic workload. The *make* phase (Phase V) of the well-known Andrew file system benchmark from CMU was run in the same three Echo configurations. This phase involves compiling seventeen C source files, creating two libraries, and linking one application program. As the table indicates, the benchmark runs took 4% longer with directory write-through, and 8% longer with both directory write-through and file write-back on close.

What do these results mean? In both benchmarks, directory write-behind gave a clearly measurable improvement in running time. Also, file write-back on close gave a clearly measurable penalty.[9] It is clear that the benefits of directory write-behind depend on how much time an application spends modifying directories. In the first benchmark, the files being compiled were very short, so little time was spent running the compiler. Thus speeding up file creation gave a large improvement. The second benchmark compiled larger files and also spent time running the linker and library builder, so there was much less benefit in speeding up file creation. It seems likely that real applications will see benefits closer to the 4% of the Andrew

[9] We think of directory write-through with file data write-behind as the base case in this comparison, because it is what single-machine Unix file systems provide, as do the newer distributed file access protocols emerging as successors to NFS (see Section 7).

| | mean time (18 runs) | range | relative time |
|---|---|---|---|
| Full write-behind | 9.9 sec | +1.3/-0.4 | 1.00 |
| Directory write-through | 24.1 sec | +0.4/-0.3 | 2.43 |
| Write-back on close | 25.1 sec | +2.4/-1.0 | 2.54 |

Table 3: Elapsed time for the Vesta benchmark.

benchmark than the 42% of the simple compilation benchmark. But it is important to note that these benchmarks were run on relatively slow machines by today's standards. In the future, as processors get faster and disk speeds lag further and further behind, applications are likely to see more and more benefit from directory write-behind.

Table 3 shows the results of a benchmark run on the Vesta software configuration management system. This benchmark was developed by the Vesta research group to evaluate the performance of Vesta's code repository. It simulates storing a new version of a software package in the repository, in the case where all the source files in the package have actually changed since the previous version, and where the user has already compiled and linked the sources to produce a new set of derived object files. Thus a whole new set of files must be copied into the repository, but there is very little else to be done. In this test, there were 8 source files comprising 16 Kbytes of data, and 10 derived object files comprising 1478 Kbytes.

As the table shows, the benchmark took 2.43 times as long with directory write-behind turned off, and 2.54 times as long with file write-back on close. This result may seem strange. Why should directory write-through slow down the benchmark so much when only 18 files are being written? We can explain this by recalling that the Vesta implementation makes heavy use of Echo's write-behind ordering guarantees to ensure that its data structures remain consistent despite crashes, as was discussed in Section 4 above. In particular, Vesta makes sure that a file has been completely written before it appears in the repository under its permanent name, and that all the files in a directory tree are completely written before the tree appears under its permanent name. So in this benchmark, there was at least one rename operation ordered after each file was written. When these renames were written through, the files had to be written first, making the benchmark run almost as slowly as if the files themselves had been written through (or written back on close).

Thus this benchmark makes an interesting point about file system design

| | Unix with local disk | Echo with write-behind | Echo with write-behind plus *sync* | Echo with write-thru |
|---|---|---|---|---|
| Create files | 4.95 | 2.94 | 7.89 | 13.1 |
| Delete files | 1.72 | 3.28 | 4.18 | 13.8 |
| Create directories | 18.6 | 2.17 | 9.33 | 18.7 |
| Delete directories | 7.21 | 1.53 | 5.45 | 15.0 |

Table 4: Elapsed times for 100 operations, in seconds. (Mean over 20 runs.)

choices. When we were designing Echo, we first decided to do directory write-behind, then later decided to give ordering guarantees on the write-behind. But as we have said, operation reordering and lost write-behind are problems even on conventional Unix systems where only file data is written behind. So one might consider adding ordering guarantees even to systems without directory write-behind. Unfortunately, as this benchmark shows, directory write-though and ordering guarantees are a bad combination; together they can reduce the performance of file writes to the point where they might as well be write-though too.

As a final benchmark, we did some absolute speed tests against a version of Unix running with local disks on hardware similar to Echo's. Table 4 displays some cases where Echo showed an advantage—creating 100 empty files, deleting 100 files, creating 100 directories, and deleting 100 directories. In each case the 100 operations were performed by a single program executing 100 system calls. Comparing the first two columns of the table, the Unix system was fast at deleting files, but in every other case, Echo was considerably faster than Unix. In fact, in some cases Echo was faster even when the timing included the time to flush all the write-behind using the *sync* system call, as shown in the table's third column. In all cases, however, Echo was much slower than Unix when write-behind was turned off (fourth column).

Some additional information about the internals of Unix and Echo helps to explain the numbers in Table 4. All the measured operations are write-through to disk in the Unix implementation, so the numbers in the "Unix with local disk" column each include the time for at least one disk write, and in most cases more than one. It appears that file deletion requires only one synchronous disk write on the Unix system we measured, since 1.72 seconds is roughly the time required for an RA-82 disk drive to rotate 100 times. The numbers in the "Echo with write-behind" column primarily reflect the time needed to update the clerk's in-memory data structures; no disk writes were required. The numbers in the last column, "Echo with write-through," are the sum of the time required for the clerk to update its in-memory structures, the time for 100 remote procedure calls, the time for the server to process these calls, and the time for 100 disk writes. The tests reported in the third column, "Echo with write-behind plus *sync*," involved the same amount of work as those in the fourth column, but the measured times are much smaller because of pipelining—the client, server, and disk drive were all working in parallel. In addition, group commit to the log could have come into play on the server [10], reducing the total number of disk writes.

As is evident from these measurements, both the Echo clerk and server required considerably more CPU time to do comparable operations than their Unix counterparts. The server in particular was heavily CPU bound. We do not believe these performance problems were caused by any fundamental flaw in the Echo approach or algorithms; they were merely an artifact of our prototype implementation.

What overall conclusions can we draw from the benchmarks in this section? We have seen that the speedup an application gets from directory write-behind depends strongly on the amount of computation or other work it does between directory writes. From the final benchmark we see that the Echo prototype benefits more from directory write-behind than most file systems would, because Echo's servers are relatively slow. These two factors make it difficult to tell just how worthwhile it would be to add directory write-behind to a production file system. There would undoubtedly be some benefits, however. Even though the implementations of the Echo clerk and server are both relatively slow, directory write-behind made some operations faster in Echo than in Unix. Therefore it is reasonable to expect that adding directory write-behind to a fast file system implementation would make it even faster. Also, even a fast file server can become slow when it is loaded down with requests from many clients. Though we did not measure this effect, it seems clear that write-behind can give clients faster responses to their requests when the underlying server is slowed by heavy load.

# 6 Implementation

In this section we discuss some significant features of the Echo caching implementation.

## 6.1 Coherence

*Tokens* are the mechanism Echo uses to keep its caches coherent. The basic token algorithm is simple. Before an Echo clerk can hold data in its cache, it must obtain an appropriate token from the Echo server that stores the data.[10] Holding a *read token* on a data item permits the clerk to read the data from the server and cache it, but not to modify it. Holding a *write token* on a data item permits the clerk to modify the data in its cache and to write the changes back to the server. Reading or writing without a token is not permitted. The server keeps track of which clerks have which tokens and prevents conflicts that could cause cache incoherence. In particular, if any clerk holds a write token on a data item, no other clerk is permitted to hold a read or write token on the same data. Thus the caches are single-copy equivalent: Either there are no write tokens outstanding on a data item, in which case the server's copy and all the cached copies of the item are identical, or there is exactly one clerk with a write token, in which case that clerk's copy of the item is the only one accessible.

Whenever a clerk needs a token that it does not have, it makes a remote procedure call to the server. If there are no conflicting tokens, the server grants the new token immediately. If there are any conflicting tokens, the server calls each clerk that holds one, asking it to give the token back. When a clerk is asked to give up a read token, it does so immediately. When asked to give up a write token, the clerk immediately writes back any changes it has made to the data that the token covers, then gives back the token. In either case the clerk discards its cached copy of the data.

In our implementation, the data item covered by a token is a whole file or whole directory. However, clerks can read, write, and cache individual file blocks or directory entries. This token granularity is a convenient choice, but makes for artificially poor performance if two or more applications on different machines are concurrently accessing different parts of the same file or directory, and at least one of them is writing. In such a case the object's token is continually moving back and forth between the two machines and their writes are continually being flushed to disk, even if there is no real communication going on between the applications.

This behavior is not a problem on files in our environment, because our users present Echo with a Unix-like workload in which shared, mutable random-access files are rare. In environments where files are often accessed in this way—VMS, for example—one could change Echo's token mechanism to work on byte ranges instead of whole files, using the technique Burrows describes in his thesis [5].

We did have a few problems with widely shared, mutable directories, but we were able to work around them. For example, much of the shared software at our installation is kept in *packages,* which are stored as subdirectories

[10]For the moment, we ignore the fact that Echo servers may be replicated; the impact of replication on the token mechanism is discussed in Section 6.2 below.

of `/proj/packages`. The directory `/proj/topaz/bin` contains symbolic links to the executable programs that are stored in these packages. Every user has `/proj/topaz/bin` on his shell's search path, and some of the programs in it are frequently used. As a result, every machine needs a read token for `/proj/packages` most of the time. But the tool that ships new versions of packages to `/proj/packages` works by first creating a new directory with the new contents, then using two rename operations to replace the old version with the new, as in the example of Figure 5, Section 4. These rename operations of course require the write token on `/proj/packages`. When Echo was first put into use, users often noticed a delay in service when a package was shipped, as their clerks waited to reacquire their read tokens on `/proj/packages`. At first we were puzzled by this delay, but we soon discovered the cause. The final rename calls in a package installation forced the write token on `/proj/packages` to be acquired, but since all the file creations and writes were ordered before the renames, the renames could not be performed, and the token could not be released, until all this work was completed. This could take a long time—and during this time, any client that needed the read token on `/proj/packages` would have to wait. Fortunately, modifying the package tool to call *fsync* just before the final rename operations proved to be a satisfactory work-around. With this change, the write token on `/proj/packages` needs to be held only long enough to do the renames themselves. The package tool runs a bit more slowly, but all other users see better performance, so the tradeoff is well worthwhile. If we had encountered more instances of this problem, we could have fixed it in a more general way, perhaps by extending the token mechanism to make directory tokens cover individual entries or alphabetical ranges.

Some write operations in Echo modify more than one file or directory, and therefore require more than one token, so care is needed to avoid the possibility of deadlock or livelock. For example, a deadlock could occur if two different clerks both needed the same two write tokens, each clerk had one, and neither clerk would release its token before acquiring the other. On the other hand, a livelock could occur if each clerk was willing to release its first token before acquiring the other; the tokens might bounce back and forth indefinitely with neither clerk able to get both at once.

We solved these problems by defining a fixed order in which tokens must be acquired and held. Unfortunately, the tree structure of files and directories does not define a natural total order that matches the order in which operations find their operands—some operations work down the tree, some work up, and some operate on two unrelated directories—so we had to choose an arbitrary order. We chose to sort the operands by their unique numeric identifiers.

A write operation with several operands (such as rename) therefore proceeds in two phases. In the first phase, the clerk finds all the operands; this may involve reading a number of directories and acquiring a number of tokens. During this phase, the clerk is willing to immediately release any of the tokens it has acquired. In the second phase, the clerk rechecks that it is holding each of the tokens it needs and reacquires any it has lost since the first phase, proceeding in the defined order for deadlock avoidance. When a token is found or reacquired in the second phase, the clerk marks the object it covers as *dirty* by incrementing a counter associated with the object. After the operation is completed and written back to the server, the dirty counter is decremented. Thus, whenever the clerk has write-behind for an object, the object's dirty counter is greater than zero. If the server asks the clerk to give back a token for an object whose dirty counter is greater than zero, the clerk does so only after driving the counter to zero by sending all the object's buffered write operations to the server. (If the last of these operations is still in phase two, the clerk of course finishes processing it locally before sending it to the server.) If the clerk finds it must reacquire a token during the second phase, it checks to see whether the object that the token covers has been modified since the first phase; if so, the clerk aborts what it is doing, decrements all the dirty counters it has incremented, and retries the operation from the beginning. This abort and retry are necessary to ensure that the current operation and the operation that modified the object are serialized.

Token-based schemes are flexible. Once we had the basic token scheme designed, it was easy to extend it to handle more details of the Unix file system interface.

Our first addition was the *open token*. In a Unix file system, a file can be deleted from the name space while one or more processes still have it open. When this occurs, the file continues to exist (even though it is nameless) until it is no longer open. To implement this behavior in Echo, a machine's clerk acquires and holds an open token on every file that is open on that machine. Unlike read and write tokens, open tokens are never called back by the server; they do not conflict with any other type of token.

The Echo clerk uses two heuristics to reduce the overhead of acquiring and releasing open tokens. First, the clerk does not release an open token just because no process on its machine still has the file open. Thus if the same file is repeatedly opened and closed, the token is acquired only once. Second, whenever an application opens a file for reading, the clerk requests both a read token and an open token in one call to the server; this reduces overhead when the file is actually read, which is likely to happen soon. The clerk takes similar action when a file is opened for writing.

Two further heuristics are needed for releasing open tokens, so that deleted files are not kept in existence when no one has them open. First, when an application deletes a file from the name space, if no process on the same machine has the file open, the clerk releases its open token in the same call to the server that deletes the file. Finally, if the server calls back the token that permits deleting a file (a special kind of write token; see Section 6.3), and no application on the machine has the file open, the clerk gives up its open token in the response to the call.

More additions to the set of tokens are discussed in Section 6.3 below.

## 6.2 Fault tolerance

Fault tolerance in the Echo clerk is implemented using *leases, sessions,* and *token replication.*

Echo uses *leases* to allow servers to reclaim tokens from clerks that have crashed, while at the same time ensuring that caches remain single-copy equivalent even when network faults cut off communication between servers and clerks.[11] A lease is an agreement between server and clerk that a clerk's token will remain valid for a given period of time. If the clerk does not renew its lease on a token before the lease expires, the server is free to revoke the token, even if network faults prevent the server from communicating with the clerk. But until the lease expires, the server is not allowed to revoke a token unilaterally; it must ask the clerk to release the token and wait for a response. Without leases, there is no safe way for a server to reclaim tokens held by a clerk it cannot communicate with. The clerk machine may have crashed, or it may remain out of communication with the server for a long time, so we certainly want to have the server take the tokens back when other clerks ask to access the files they cover. Yet, if the server unilaterally revokes the tokens while the clerk machine that holds them is still running, then single-copy equivalence may be violated: Applications running on the machine that is out of touch with the server may read (or write) values in the local cache that disagree with the values seen on machines that are still in touch. But with leases, a clerk can maintain single-copy equivalence simply by checking, each time it reads or writes cached data, that the lease on the token that covers the data is still in force. In effect, real time is being used as a communication channel, letting the know when its tokens are definitely valid and when the server may have revoked them, even if the network connecting the clerk and server is broken. (For this application, the clerk and server clocks do not actually have to be synchronized; it is sufficient for them to run at the same

[11] The term *lease* was first used by Gray [8]; we compare his system with ours in Section 7.

rate within some known error bound—which is fortunate since there is no way to synchronize clocks on two machines that cannot communicate!)

Echo uses *sessions* to reduce the amount of network traffic needed to keep leases up to date. Whenever a clerk wants to begin caching data from a new server, it calls the server to establish a session. Individual tokens do not have leases; instead, each token is associated with a particular session, and there is a lease on the session as a whole. This technique dramatically reduces the number of lease renewal messages that must be sent; it also reduces the bookkeeping burden on both clerks and servers.

Echo provides a fast way for the system to recover when a server crashes by *replicating the token directory* of each server on a backup server. This approach is natural for Echo because (as mentioned in Section 1), Echo already uses replicated servers for high availability. Server replication in Echo uses a primary/backup scheme; at any given moment, one server is the primary for a given set of disks, and all requests from clerks are directed to it. A backup server takes over if the primary crashes. The backup has a path to at least one replica of the data that can function even when the primary is down. Token replication is easy to add on top of this base. Whenever a clerk asks the primary for a token, the primary makes a nested remote procedure call to the backup, which records the token in its copy of the directory; only when this call returns does the primary return the token to the clerk. Both the primary and backup keep their token directories in main memory, not on disk, so they are fast to access.

One alternative to this scheme would be for the primary server to record the token directory on its (possibly replicated) disk. The main advantage of this alternative is that the token database is not lost even if both servers crash. We did not find this advantage compelling, since it is much slower to write the tokens to disk than to record them in memory on two servers. Another apparent advantage of writing tokens to disk is that it does not require two servers, so it works even when servers are not replicated. But token replication does not require the backup server to be a dedicated machine or to have access to a disk, so there is little cost in configuring a backup server for use only by the token machinery.

Another alternative would be for a server that crashes to recover tokens from clerks when it reboots, as is done in the Sprite file system [19]. This scheme could also be used when a backup server takes over from the primary. With this scheme, normal operation is slightly faster, because the primary does not have to call the backup on each token acquisition. But failure recovery is much slower, because many clerks must be contacted and a substantial number of tokens must be recovered from each—two minutes is a typical time for this process in Sprite [2]—and if some of the clerks are down, the server may have to wait for an additional timeout period to discover this. Also, due to Echo's model and implementation of security (discussed in the next section), the recovering server would have to check the clerks' token lists for consistency with each other and for legality according to file access control lists, slowing down recovery even more. Moreover, if the lists are all legal but are not consistent, there is no way to determine which clerk has made a false claim, so in the worst case a faulty clerk could cause a nonfaulty one to lose its cache tokens and associated write-behind.

Weighing the advantages and disadvantages, we prefer token replication as the first line of defense against server crashes. However, we encountered enough double server crashes in Echo to convince us that token recovery would have been useful as a second line of defense. It would have considerably reduced the disruption to users in these cases.[12]

### 6.3 Security

As mentioned in Section 1, Echo servers do not trust Echo clerks. For each operation a clerk requests from a server, the clerk must authenticate itself as acting on behalf of some user who is authorized to perform the operation. For example, to read part of a file into its cache, the clerk has to authenticate itself as acting on behalf of some user, and the clerk has to check the file's ACL (access control list) to see whether that user has read access to the file. A clerk is able to authenticate itself as a particular user if (and only if) that user has logged into the clerk machine.[13] The authentication and login protocols Echo uses were developed as part of a separate security architecture project and are described in detail in another paper [16].

We use two kinds of caching to make access control decisions fast. First, there is caching within the authentication protocol implementation, so that most remote procedure calls are authenticated with no extra packets or cryptographic overhead. Authentication caching is beyond the scope of this paper. Second, we extended the Echo token mechanism to provide caching for ACL checks. We extended the set of tokens so that there is a separate token for each kind of access permission a clerk may have on a file or directory. Thus the server needs to do ACL checking only on token acquisition requests. On actual read or write requests, the server checks only that the clerk has the appropriate tokens, which is considerably faster (and which

[12]Perhaps the Sprite research group has reached a similar conclusion; Sprite has recently been modified to replicate tokens in a segment of the server's own memory that is usually preserved across reboots, recovering tokens from clients only when this memory is lost [2].

[13]Thus a user who logs into a machine obviously must trust its clerk, since the clerk is able to request any operation it pleases on behalf of the user. Because the clerk is part of the machine's operating system, it seems reasonable to require this kind of trust.

the server would have to do anyway, since it does not trust the clerk).

In the extended set of tokens, the read token described in Section 6.1 is replaced by three tokens—*InfoToken, SearchToken,* and *ReadToken.* Holding an InfoToken on an object allows a clerk to read and cache a record of information about the object, corresponding to what is returned by the *stat* system call in Unix. There are no access control restrictions on obtaining an InfoToken. Holding a SearchToken on a directory allows looking up individual names in the directory and caching the results, but not reading the entire directory. Holding a ReadToken allows reading file data or reading a directory in its entirety. A clerk can obtain a SearchToken or ReadToken only if acting on behalf of a user with the corresponding access permission.

There is no token corresponding to execute-only file access, because there is no way for the server to enforce the distinction between reading and executing. For a user to execute a program, the clerk on his machine must be able to read it; but if the clerk is able to read the program, there is no way for the server to keep the clerk from letting the user read it. Therefore a clerk is allowed to obtain a ReadToken when acting on behalf of a user with execute-only access.

The write token is replaced by three tokens—*WriteToken, ChangeAccessToken,* and *ChangeParentToken.* Holding a WriteToken on a file allows writing data or changing the file's length; on a directory, it allows modifications such as creating, deleting, or renaming objects named in the directory. Holding a ChangeAccessToken on an object allows changing the object's ACL. To obtain either of these tokens, a clerk must be acting on behalf of a user with the corresponding access permission—write access for WriteToken; ownership for ChangeAccessToken. Finally, a clerk must hold a ChangeParentToken on an object to rename it or delete it. There are no access control restrictions on obtaining a ChangeParentToken.

The ChangeAccessToken has a special property—a clerk that holds this token on an object has blanket permission to acquire any other tokens on the same object that it asks for, with no ACL check. This property is needed because the clerk may have written behind a change to the ACL that makes it legal to obtain the tokens it is asking for, even if the server's copy of the ACL says it is not. There is no security hole here. If the clerk is acting on behalf of a user who can change the ACL, then the server must allow the clerk to change the ACL whenever it asks, so there is nothing to be gained by forcing the clerk to write back such a change before it can take advantage of the changed permissions.

Table 5 summarizes the compatibility rules among all the types of cache coherence tokens. Where "No" appears at the intersection of a row and column in the table, it is not permitted for two different clerks to hold the two tokens named at the head of the row and column on the same object at the same time; where "Yes"

| | Open | Info | Search | Read | Write | Change Access | Change Parent |
|---|---|---|---|---|---|---|---|
| Open | Yes | Yes | — | Yes | Yes | Yes | Yes |
| Info | Yes | Yes | Yes | Yes | No | No | No |
| Search | — | Yes | Yes | Yes | No | No | No |
| Read | Yes | Yes | Yes | Yes | No | No | No |
| Write | Yes | No | No | No | No | No | No |
| ChangeAccess | Yes | No | No | No | No | No | No |
| ChangeParent | Yes | No | No | No | No | No | No |

Table 5: Compatibility matrix for cache coherence tokens.

appears, it is permitted. OpenToken applies only to files and SearchToken only to directories, so there is no table entry at their intersection.

Our security implementation departs from single-machine Unix semantics in one detail. In Unix, a file's ACL is checked only when the file is opened. So if a process has a file open for writing, it can continue to write the file indefinitely, even if the file's ACL is changed to remove the process's write access; and similarly for reading. We could have emulated this feature by splitting the OpenToken into OpenReadToken and OpenWriteToken, with the property that holding such a token allows a clerk to obtain the corresponding ReadToken or WriteToken without an ACL check. We chose not to do so because it seemed too complicated, and because we feel that this Unix feature is not a good idea in a fault-tolerant distributed system. On a single-machine Unix system, one can always force all processes to close their open files by rebooting. But there is nothing corresponding to rebooting in Echo, so there is no way to ensure that no process is holding a file open when its ACL is changed to remove access. Unfortunately, it turns out that a few applications we wanted to run depend on the Unix semantics, so we adopted a compromise: Echo servers allow the owner of a file to get a read or write token even if the file's ACL does not give the owner the corresponding access. This feature allows the owner (and only the owner) to continue accessing an open file after its permissions are taken away. (It could also allow a file's owner to open the file without having access permission, but the Echo clerk does not allow this.) As with the special rule for ChangeAccessToken, there is no security hole here, because a file's owner has permission to change the ACL and give himself access. Restricting this feature to the owner has not been a problem for any Unix application we have tried. NFS has essentially the same feature for the same reason, and also restricts it to a file's owner.

### 6.4 Resource reservations

Creating or enlarging a file or directory requires resources on the file server—chiefly disk space. Echo allows clerks to reserve resources in advance, so that when a clerk buffers an operation for write-behind, the user can be assured that the necessary resources will be available when the write is performed.

Disk space reservations are fairly simple. Each write operation that the clerk sends to the server uses up some of the clerk's reserved disk space. The clerk is linked with a library routine provided by the server that tells it how much space each operation requires. (For complex operations that require varying amounts of space, the routine computes a simple, conservative estimate.) Whenever an application asks the clerk to do a write, the clerk first checks whether it has the necessary space reserved; if not, it asks the server for more. The clerk specifies two numbers in its request—a minimum amount and a desired amount. The minimum amount is what is needed to complete the current application request. If the server grants less than the minimum, the clerk returns a "Disk full" error to the application. The desired amount is the amount the clerk would like to reserve ahead to reduce the number of future reservation requests it has to make.

A clerk's disk space reservation is associated with its session, so if the session's lease runs out, the server can reclaim the reserved space. Unlike tokens, however, we do not replicate space reservations; instead, we use a lazy form of recovery for them. (We chose to do this because write requests are more frequent than token acquisitions, and we did not want to slow them down by adding an extra remote procedure call to the backup.) After a primary server crashes and the backup takes over, each clerk's reservation is recovered the next time the clerk sends a write request to the server. The server responds to the write request by asking the clerk to send its current reservation value and retry the request. There is a minor security hole in this mechanism. A malicious clerk could lie about its reservation, claiming to have reserved much more space than it really did. This could cause the server's disk space to be overcommitted, so that other clerks would not be able to get back their legitimate reservations. We decided not to worry about this problem, because it is unlikely to arise in practice and we could not think of a solution short of replicating the reservations.

Unique identifiers for files and directories are another resource that clerks reserve. Echo servers do not trust clerks to correctly generate these identifiers, because a malicious clerk might give the same identifier to two different files, thereby corrupting the file system's tree structure. Therefore, so that file and directory creation can be write-behind, clerks are allowed to reserve a stock of these identifiers ahead of time. A clerk's reserved identifiers are associated with its session, so the server can reclaim the space needed to keep track of them if the session's lease expires. A clerk is automatically granted all the coherence tokens on each object that it creates using a reserved identifier; this avoids an extra call to the server upon object creation.

## 6.5 Ordering constraints

The Echo clerk's implementation of ordering constraints falls out naturally from the way it does write-behind. For each cached object that has been locally modified, the clerk keeps a representation of the object's current state (or at least the modified portion), plus a write-behind queue—a list of operations that have not yet been written back to the server, in the order they were logically performed. An operation with more than one operand is on the write-behind queue of each operand. A sequence of file overwrites unbroken by any other write operation appears as a single element in the file's write-behind queue. An *forder* operation appears in the write-behind queue of each of its operands, but is treated as a no-op when it reaches the head of the queue. This queue data structure corresponds precisely to the $\Rightarrow$ relation for write-behind ordering that we defined in Section 3.1 above. Before the clerk sends any write operation to the server, it checks that the operation is at the head of the write-behind queue for each of its operands. If not, the clerk first sends out the operation's predecessors, after recursively checking that each of them is at the head of the write-behind queue for each of its operands.

To improve performance, we use a pipeline to send write requests from clerk to server. Each write request is a remote procedure call, but several calls may be issued in parallel by separate threads. Each call carries a sequence number, however, so that the server knows the order in which they must be physically carried out. Thus under load, the clerk is able to send a continuous stream of requests to the server—it does not have to wait for one request to complete and the reply to arrive over the network before it sends out another.

## 6.6 Reporting lost write-behind

To identify the processes that depend on discarded write-behind, the Echo clerk uses a simple, conservative technique. When a process accesses a volume for the first time, the clerk's current session on that volume is recorded as part of the process state. When a process accesses a volume it has accessed before, the saved session identifier is compared with the current session identifier for the volume; if the identifiers differ, the process may depend on write-behind that was discarded, so it receives an error return.

| | Shared | Exclusive |
|---|---|---|
| Shared | Yes | No |
| Exclusive | No | No |

Table 6: Compatibility matrix for advisory lock tokens.

This technique is more conservative than necessary, however. First, there may not actually have been any write-behind lost in the old session. Second, even if write-behind was lost, some processes that accessed the volume may not actually have been affected by the loss. It would be easy to fix the first problem with a small amount of added bookkeeping in the clerk, but fixing the second would require substantial extra bookkeeping.

Whether fixing these problems is worthwhile depends on how often they occur, which in turn depends on how often network faults occur and how often there is actually write-behind buffered when one does occur. When we were designing Echo, we believed that token revocation due to network faults would be very rare, so the extra bookkeeping would not be worthwhile; unfortunately, this turned out not to be the case. Fixing the first problem would have had a substantial payoff in Echo, because the workstations in our installation are lightly loaded, so at any given moment a workstation usually has no write-behind buffered.[14] But clearly this property could easily change if Echo users started doing more computing or running a different mix of applications.

### 6.7 Advisory lock tokens

The Echo clerk implements advisory locks using another set of tokens for each file and directory. If any process on a machine holds a shared advisory lock on an object, that machine's clerk must have a SharedToken for the object. If any process holds an exclusive lock, the clerk must have an ExclusiveToken. Table 6 gives the compatibility matrix for advisory lock tokens.

As with cache coherence tokens, a clerk may hold an advisory lock token even when no process on its machine needs it. Thus if advisory locks on an object are repeatedly acquired and released by applications running on one machine, no communication with the server is necessary.

[14]The Echo clerk schedules a write to be flushed to the server within 15 seconds of the time it was logically performed, and it blocks further write calls on a volume if any write to it has been waiting to reach disk for more than 300 seconds. Thus when an Echo workstation is not in active use, it typically has no write-behind buffered.

When a clerk requests an advisory lock token that conflicts with one held by a second clerk, the server calls back the clerk holding the conflicting token. If the conflicting token is not in use (that is, no process on the second clerk's machine is holding an advisory lock that needs it), the second clerk returns it immediately. If the token is in use, however, the callback blocks until the token is no longer in use. The blockage propagates back through the server to the first clerk, and ultimately causes the application that called the first clerk requesting the conflicting lock to block until the lock is released. Other threads within the server and clerks continue to run.

A problem with this implementation is that conflicting locks result in remote procedure calls that remain outstanding for a long time. In the RPC implementation we are using, such calls consume excessive resources in the RPC runtime library. We know of several ways to fix this problem, either by changing the RPC system or by changing the way we use it, but have not found time to implement any of them.

A clerk's advisory lock tokens are associated with its session. Thus if the session's lease expires, the server can reclaim the tokens and grant them to another clerk. Error returns due to discarded locks are generated using the same mechanism as those due to lost write-behind. This implementation gives the desired semantics for advisory locks, as described in Section 3.2 above.

# 7 Related work

Throughout this section (and in other parts of this paper) we use Echo terminology to discuss related systems, even when the papers describing those systems use different terminology. In particular, other systems often use different terms for what we call *volumes, clerks,* and *tokens.*

Like Echo, the Sprite file system uses tokens to maintain coherence in a distributed file cache [19]. Sprite's caching differs in several respects from Echo's, however.

Sprite clerks cache only files, not directories. When a Sprite application process asks to open a file, the local clerk sends the request on to the server machine that stores the file, and the pathname lookup is done there. Sprite clerks do *prefix caching* so that most requests can be sent directly to the correct server, without the need to broadcast or consult a name server first [28]. A prefix cache is not a full-fledged directory cache; it simply maps prefixes of absolute pathnames to the servers that store the files whose names begin with those prefixes. For example, if the directory subtree rooted at `/usr` is stored as a single volume on file server `fred`, a clerk's prefix cache might contain the mapping from `/usr` to `fred`, but the clerk would have no other information about `/usr`, and no information at all about longer pathnames such as `/usr/include`.

A Sprite clerk is required to hold a read token on a file to have it open for reading, or a write token to have it open for writing. This differs from the Echo scheme, where read and write tokens are needed only when a clerk is actually caching file data. If two Sprite clerks want to hold the same file open in conflicting modes, the server detects the conflict and turns off caching for the file, requiring all reads and writes to go to the server. Doing this seems like a good idea if two machines are really communicating data through the file—if one machine is only writing and the other is only reading, caching is pointless, and the overhead of moving tokens back and forth between the machines is wasted. (On the other hand, if two machines are accessing different parts of a random-access file, turning off caching is not the best solution; a better idea would be to give each machine a token on just the range of bytes within the file it is using.)

When we were designing Echo, we looked for a way to avoid useless caching and token-passing overhead when two applications are actively communicating through the file system, but we did not find a solution we were satisfied with. We could certainly have adopted a scheme like Sprite's, so that caching on a file would be turned off when applications on two different machines have it open in conflicting modes. However, this scheme may turn off caching in some cases where it would be beneficial, and it does not catch cases where files are shared sequentially—one process opens a file, writes it, and closes it, then another process opens it, reads it, and closes it, and this pattern repeats. It is also unclear how to extend the scheme to directories. As with files, one would like to turn off caching for a directory when processes on different machines are actively communicating by modifying it. But processes do not announce the start and end of their access to a directory by opening and closing it, so one would have to use a heuristic to decide when to turn directory caching on and off. Perhaps such a heuristic could also detect sequentially shared files. Further research seems to be needed in this area; however, when Baker et al. [3] compared the performance of the Sprite and Echo schemes for handling write-shared files, they found that the choice made little difference on the workload traces they were able to gather.

Sprite does not replicate its token directory. When a Sprite file server reboots after a crash, it reconstructs its token directory by contacting all its clients and asking them what tokens they hold. The pros and cons of this approach were discussed in Section 6.2 above.

Sprite does not use leases. If a server loses touch with a clerk, it invalidates the clerk's tokens immediately. Therefore, for the reasons discussed in Section 6.2 above, Sprite does not provide strict single-copy equivalence.

The Andrew File System [13, 14] caches both files and directories on client machines. On directory updates, AFS does write-through, not write-behind. AFS caches files in their entirety, not block by block, and it delays writing changes to a file back to the server at least until the file is closed. The file cache is kept coherent using a scheme similar to Echo's; what AFS terms a *callback* is equivalent to an Echo token. The AFS paper just cited does not explain what happens if two different client machines attempt to hold the same file open for writing at the same time, or if one attempts to hold it open for reading and another for writing. Like Sprite, AFS does not use leases and thus does not provide strict single-copy equivalence.

Burrows has implemented a file caching service, called MFS [5], that is similar to Echo's in many respects. MFS caches both files and directories, with write-behind for both, and with coherence implemented using a token scheme similar to Echo's. Like Sprite and AFS, MFS does not use leases and thus fails to provide strict single-copy equivalence.

MFS does not provide any guarantees about the order in which updates are written back to the file server; in particular, updates to a directory can be reordered arbitrarily, as long as the reordered update sequence would (in the absence of faults) leave the directory in the same state as did the original logical sequence. For example, if a user creates file `/d/a` and then file `/d/b`, but a crash causes some write-behind to be lost, file `/d/b` may exist while `/d/a` is never created. Or if a user writes some data into file `/d/c` and then renames it to `/d/d`, and a crash causes some write-behind to be lost, the file may have been renamed without the data having been written to it.

MFS does an excellent job of saving work for the server by cancelling sequences of operations that have no net result before they leave the clerk's write-behind buffer. For example, if an application creates a temporary file, writes into it, reads it, and deletes it within a short time, MFS avoids sending any of the operations to the server. (Actually, this sequence changes the last-modified time of the directory in which the file was created, so MFS should tell the server about that change, but apparently this was not done.) Our original plans for Echo called for us to do this kind of optimization, but we did not find time to implement it.[15] MFS's lack of ordering guarantees makes it easier to implement these optimizations than it would have been in Echo.

Burrows introduced the concept of *byte-range tokens* in MFS. Logically, each byte of an MFS file has its own token, and tokens on different bytes within a file do not conflict. Changing the length of a file requires holding write tokens on the bytes

[15] As implemented, Echo collapses multiple overwrites to the same file bytes if they are not ordered by $\Rightarrow$, but it does no other work-cancelling optimizations.

being added or deleted. A file's other properties (such as owner and last-modified time) are covered by another token. In data structures and interfaces, sets of tokens on a file are represented as ranges; this representation is very compact except in pathological cases. As mentioned above, we believe that byte-range tokens would be a useful addition to the Echo token scheme under workloads where shared, mutable random-access files are common.

Gray coined the term *lease* in a paper about his file caching server [8]. We developed the lease concept simultaneously and independently of Gray, but chose to adopt his terminology when we learned of his work. Gray's system does not use write-behind at all, so it needs only one kind of token. If a clerk has a token on a file, it may cache a clean copy as long as its lease remains valid. If a clerk wants to write a file, it sends the write request directly to the file server, which recalls all tokens on the file before performing the write. (Alternatively, the server can delay the write until all the leases have expired, refusing to honor any lease renewal requests that come in during the waiting period.) Gray's system does not have sessions; each cached file has an independent lease.

QuickSilver [22] provides a transactional interface to its file system. Operations on multiple files and directories can be grouped into a transaction that is committed or aborted as a unit. Although we chose not to explore transactional file semantics in Echo, we view this area with interest and would be pleased to see research in it succeed in producing a practical system. The transactional programming model is cleaner and seemingly easier to use than Echo's model of ordered write-behind, while still allowing write-behind for transactions that have not yet been committed. Moreover, QuickSilver transactions can include operations on both files and non-file objects, and can cover files in different volumes managed by different servers. A drawback of the QuickSilver approach is that it requires considerably more machinery to implement, giving rise to concern about how well it may perform compared to more conventional approaches. Also, many applications that use the file system need to be modified to work properly with QuickSilver. By default, every file system action taken in a QuickSilver program is part of a single atomic transaction that commits when the program exits—but this behavior is not appropriate for many programs, for example, long-running text editors.[16]

For a more extensive bibliography on file systems that do caching on client machines, see Burrows's thesis [5].

[16]Of course, for some applications, using Echo's semantics requires modifying applications by adding *forder* calls. But these modifications are optional—if they are omitted, the only problem is that the application does not tolerate lost write-behind well, a problem that is most likely just as bad when the application is run on a single-machine Unix system with write-behind.

# 8 Conclusions

The Echo distributed file system has studied a collection of useful techniques for improving the performance and semantics of distributed file caches and demonstrated their feasibility. We believe that future file systems will benefit from adopting many of these techniques. In particular, Echo provides fully coherent file and directory caching on clients, with ordered write-behind on updates to both files and directories.

Coherent caching greatly simplifies the task of writing distributed applications that use the file system. Together with Echo's location-transparent global naming, coherent caching makes the distributed nature of the file system invisible to applications, while providing much better performance than an uncached file system could. The bookkeeping cost of maintaining cache coherence seems well worth the benefits.

For coherent caching with write-behind to work well, however, the underlying network must have good availability, and the medium used to store the system's token directory must be reliable.

If the network is often broken, clients will often be unable to use the file system, even if they have all the files they need cached, because they are unable to communicate with the server and thus are unable to be sure their caches are coherent. The Echo system uses the AN1 network [23], which achieves very high availability through redundancy, though Ethernet and most other non-redundant local area networks have high enough availability for the purpose. On networks that are often unavailable, however, file systems that allow controlled forms of incoherence, such as LOCUS [20, 27] and Coda [15], may be more practical.

If the system's token directory is lost, all client machines lose their write-behind, disrupting the work of many users. Therefore the token directory must be either replicated, recoverable from clients after a server crash, or both. We prefer token replication as the first line of defense because it makes for much faster recovery, at the cost of updating two replicas on each token acquisition. It seems worthwhile to implement token recovery as well, as a second line of defense when all replicas of the token directory are lost.

Write-behind is an important building block for file systems, whether distributed or not. Nearly every file system we are aware of does write-behind of some kind, though often only on file data. Write-behind certainly reduces latency, by eliminating the need for applications to wait for the disk on every write. It can also improve throughput, by smoothing out load peaks and by enabling sequences of operations that have no net result to be cancelled before they leave the write-behind buffer.

Echo does write-behind on directory modifications, including file and directory creation and deletion, and we have shown that this can improve the performance of some applications. For example, it speeds the compilation of programs made up of many small modules, like typical large C programs and libraries. This speedup is not large, but may grow as the disparity between CPU and disk speeds increases. Directory write-behind adds complexity to the file cache implementation, but we implemented it successfully and had few problems with the code once it was in place.

The major drawback of write-behind is that a write can fail long after the process that requested it has gone on to other things or even exited. Directory write-behind does not make this problem worse in any fundamental way, but it did make both the Echo designers and our users more conscious of the problem. We tried to provide good facilities for allowing applications to tolerate lost write-behind cleanly, and we did make considerable progress in this area; however, lost write-behind remains an ugly problem. Fundamentally, write-behind cannot provide a correct implementation of the natural semantics programs expect to get from a file system interface—when you write to a file, you want the bits to be stored stably every time, not most of the time. Perhaps the ultimate solution to this problem is to eliminate write-behind, instead using non-volatile RAM on each file server to shield clients from the latency of synchronous disk writes [1].

Echo took a two-pronged approach to dealing with lost write-behind. First, make it easy for an application writer to ensure that, when a crash halts the application and causes some of its write-behind to be lost, the data structures it stores in the file system remain consistent. Second, make it easy to ensure that whenever write-behind is lost, all affected applications are halted (or otherwise notified), so that they can recover cleanly instead of continuing to run with an incorrect notion of what is on disk.

Echo did well on the first prong. It actually provides better semantics than are commonly provided even on non-distributed file systems (such as conventional Unix). Echo's ordering constraints do the right thing automatically for many simple applications and provide enough power for more complex ones, while giving better performance than if only *fsync* were available. Ordering constraints were easy to implement as a part of doing directory write-behind. On the down side, ordering constraints do appear to be more difficult for application writers to use than transactions would be.

We were less successful on the second prong. Our current design is good at making directly affected processes halt, but it also often halts processes that did not really care about the lost write-behind, causing confusion for users. And processes that indirectly depend on lost write-behind cannot be detected—for example, in a distributed or multi-process application where only some processes access the file system, those that do not access it may not learn of lost write-behind.

Moreover, we have no solution for the problem of what to do when an application's write-behind is lost after it has exited. These cases escape our two-pronged approach entirely.

## Acknowledgements

Hania Gajewska, Jim Gettys, Mark Manasse, and Mike Schroeder contributed ideas early in the Echo clerk's design phase. Mike Burrows and Mike Schroeder helped in the selection of material for this paper and provided useful comments on the presentation.

## References


[1] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. In *Proc. 5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 10–22. ACM, October 1992.

[2] Mary Baker and Mark Sullivan. The recovery box: Using fast recovery to provide high availability in the UNIX environment. In *Proc. Summer 1992 USENIX Conference*, pages 31–43. USENIX Association, June 1992.

[3] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proc. 13th Symp. on Operating Systems Principles*, pages 198–212. ACM, October 1991.

[4] Andrew D. Birrell, Andy Hisgen, Charles Jerian, Timothy Mann, and Garret Swart. The Echo distributed file system. Research report, Systems Research Center, Digital Equipment Corporation, 1993. In preparation.

[5] Michael Burrows. *Efficient Data Sharing*. PhD thesis, University of Cambridge, September 1988.

[6] Sheng-Yang Chiu and Roy Levin. The Vesta repository: A file system extension for software development. Research Report 106, Systems Research Center, Digital Equipment Corporation, 1993.

[7] Lucille Glassman, Dennis Grinberg, Cynthia Hibbard, Loretta Guarino Reid, and Mary-Claire van Leunen. Hector: Connecting words with definitions. Research Report 92A, Systems Research Center, Digital Equipment Corporation, October 1992.

[8] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. 12th Symp. on Operating Systems Principles*, pages 202–210. ACM, December 1989.

[9] Andy Hisgen, Andrew Birrell, Charles Jerian, Timothy Mann, and Garret Swart. Some consequences of excess load on the Echo replicated file system. In *Proc. 2nd Workshop on the Management of Replicated Data*, pages 92–95. IEEE, November 1992.

[10] Andy Hisgen, Andrew Birrell, Charles Jerian, Timothy Mann, and Garret Swart. New-value logging in the Echo replicated file system. Research report, Systems Research Center, Digital Equipment Corporation, 1993. In preparation.

[11] Andy Hisgen, Andrew Birrell, Chuck Jerian, Timothy Mann, Michael Schroeder, and Garret Swart. Granularity and semantic level of replication in the Echo distributed file system. In *Proc. Workshop on the Management of Replicated Data*, pages 2–4. IEEE, November 1990.

[12] Andy Hisgen, Andrew Birrell, Timothy Mann, Michael Schroeder, and Garret Swart. Availability and consistency tradeoffs in the Echo distributed file system. In *Proc. 2nd Workshop on Workstation Operating Systems*, pages 49–54. IEEE, September 1989.

[13] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanaraynan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[14] Michael L. Kazar. Synchronization and caching issues in the Andrew file system. In *Proc. Winter 1988 USENIX Conference*, pages 27–36. USENIX Association, February 1988.

[15] James J. Kistler and M. Satyanaraynan. Disconnected operation in the Coda file system. In *Proc. 13th Symp. on Operating Systems Principles*, pages 213–225. ACM, October 1991.

[16] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. In *Proc. 13th Symp. on Operating Systems Principles*, pages 165–182. ACM, October 1991.

[17] Roy Levin and Paul McJones. The Vesta approach to precise configuration of large software systems. Research Report 105, Systems Research Center, Digital Equipment Corporation, 1993.

[18] Timothy Mann, Andy Hisgen, and Garret Swart. An algorithm for data replication. Research Report 46, Systems Research Center, Digital Equipment Corporation, June 1989.

[19] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[20] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: A network transparent, high reliability distributed system. In *Proc. 8th Symp. on Operating Systems Principles*, pages 169–177. ACM, December 1981.

[21] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun network filesystem. In *Proc. Summer 1985 USENIX Conference*, pages 119–130. USENIX Association, June 1985.

[22] Frank Schmuck and Jim Wyllie. Experience with transactions in QuickSilver. In *Proc. 13th Symp. on Operating Systems Principles*, pages 239–253. ACM, October 1991.

[23] Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, and Charles P. Thacker. Autonet: A high-speed, self-configuring local area network using point-to-point links. Research Report 59, Systems Research Center, Digital Equipment Corporation, April 1990.

[24] Sun Microsystems, Inc. NFS: Network file system protocol specification. RFC 1094, Network Information Center, SRI International, March 1989.

[25] Garret Swart, Andrew Birrell, Andy Hisgen, Charles Jerian, and Timothy Mann. Availability in the Echo file system. Research report, Systems Research Center, Digital Equipment Corporation, 1993. In preparation.

[26] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite Jr. Firefly: A multiprocessor workstation. Research Report 23, Systems Research Center, Digital Equipment Corporation, December 1987.

[27] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proc. 9th Symp. on Operating Systems Principles*, pages 49–70. ACM, October 1983.

[28] Brent Welch and John Ousterhout. Prefix tables: A simple mechanism for locating files in a distributed filesystem. In *Proc. 6th Intl. Conf. on Distributed Computing Systems*, pages 184–189. IEEE, May 1986.