104

New-Value Logging in the Echo Replicated File System

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

June 23, 1993

digital

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

New-Value Logging in the Echo Replicated File System

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

Digital Equipment Corporation
Systems Research Center

June 23, 1993

Abstract

The Echo replicated file system uses new-value logging. Echo's use of new-value logging provides a clean separation of the internals of the system into one module that is concerned with logging and recovery, and another module that is concerned with accessing and updating the file system's on-disk structures. The logging module provides a restricted form of transaction. The restrictions simplify its implementation but impose constraints on the file system module. The file system module easily satisfies these constraints, resulting in a good match overall.

Contents

1 Introduction

Maintaining the consistency of on-disk structures in the face of system crashes is an important problem in file system design. This problem is difficult because file system on-disk structures can span multiple disk blocks and because file system updates can modify several on-disk structures. Because disk hardware writes each block individually, a crash can leave on-disk structures in inconsistent states. Therefore a recovery procedure must be designed as part of the file system.

Logging is a general technique for making a set of updates atomic with respect to crashes. Compared with more specialized techniques for maintaining the consistency of a file system's on-disk structures, such as scavenging and carefully ordered writing, logging has three primary advantages. Logging can keep recovery time short, improving availability. Logging can change random disk writes to sequential, improving performance. And logging localizes recovery concerns, simplifying the implementation.

Logging's ability to simplify the implementation allows a file system to be more ambitious and still be feasible:

- The file system can provide higher performance by maintaining more complex on-disk structures. Logging allows these structures to be designed without too many constraints.
- The file system can provide higher reliability by replicating disks. After a system crash, the log can be used to bring the disk replicas into agreement.
- The file system can provide higher availability by replicating servers. After a failover from one server to another, the file system needs to determine whether or not a particular update request was performed; logging makes it easier to maintain the extra information needed to determine this.
- The file system can provide stronger file system semantics to clients. Logging allows more complex operations, such as file rename, to be atomic.

Several file system implementations have demonstrated the advantages of logging. The Cedar file system implements directories as a B-tree, using logging to maintain the B-tree invariants [12]. The Harp file system and the HA-NFS file system provide replicated NFS service, and use logging to assist in replication and failover [3, 4, 19]. The Alpine file system uses logging to provide transactional semantics to file system clients [6]. The Sprite log-structured file system is radically different from these other systems and from Echo, in that all data lives in the log [26].

Echo is a distributed file system, one of whose major goals is to achieve high availability by replication of disks and servers.[1] We were therefore attracted to logging as an implementation technique.

We chose a particularly simple style of logging for Echo: new-value logging with a bounded log.[2] Only new values of data are recorded—no undo information is logged. On-line abort is not supported; that is, only a system crash can back out an uncommitted transaction.

Our new-value logging is economical and simple. Economical, because we cut logging in half by not recording the old values. Simple, because new-value logging provides a clean separation of the internals of the file system into one module that is concerned with logging and recovery, and another module that is concerned with accessing and updating the file system's on-disk structures (including controlling concurrent access to these structures).

This paper describes Echo's internal new-value logging interface, its implementation, and its use, all in enough detail to reveal the strengths and weaknesses of our approach. We believe that the restrictions in our logging interface made the implementation of logging and recovery particularly simple—simpler than for any less constrained interface. However, the implementation of the file system layer that uses the logging interface had to cope with its restrictions. Doing so was feasible because a file system provides only a small fixed set of operations. We believe that the benefits of restricting our logging interface (as opposed to providing general atomic transactions) justified the additional effort required to design and build the file system layer above it.

How do the restrictions at the logging interface create difficulties for the file system layer? Because of the lack of undo, the file system cannot abort an update in the middle—to break a deadlock, for instance. Therefore, any locks or resources that will be required during the update must be acquired in advance, in a deadlock-free order, before any changes have been made. A second consequence of the lack of undo is that all uncommitted changes must be held in primary memory, limiting the size of a transaction. Because the log is bounded, a file system update that is sufficiently large must be decomposed into a sequence of individual atomic actions at the logging interface. However, the bounds on the log size are large enough

[1] Other Echo goals include location-transparent global naming and distributed caching between clients and servers. These aspects of Echo are discussed more fully in an overview paper [5] and in several additional papers [13, 14, 15, 21, 28]. The Echo system is no longer under development or in use, but we speak of it in the present tense throughout this paper to avoid awkwardness. Echo was in active use from November 1990 to November 1992, when the evolution of our laboratory's hardware and software base led us to retire the system. At its peak, Echo had about 50 users.

[2] For a survey of logging techniques, see Bernstein [2] and Haerder [11].

that important file system operations, such as renaming a file, can be performed atomically. In none of these cases do the logging restrictions have a significant impact on file system performance, but they do make the task of implementation more difficult. Section 5 gives some detailed examples of the areas where such difficulties arose in Echo and how we overcame them.

Here is a summary of the rest of the paper.

In Section 2 we present a simplified version of EchoDisk, the interface that hides logging from the higher levels of Echo. We define the semantics of this interface using an abstract model. We also describe a realistic implementation of the interface.

In Section 3 we proceed to remove the simplifications one by one. Our simplified EchoDisk ignores the reality of bounded resources, so here we extend the interface and implementation to acknowledge the bounds. Some writes done by a file system do not need the strong semantic guarantees of EchoDisk, and should not have to pay for them, so we expand the interface to provide a weaker form of writing and change the implementation to match. Disks can be replicated below the level of EchoDisk, so we discuss the impact of implementing this. And servers written using EchoDisk can be replicated, with only minor changes to the interface, so we make these changes.

In Section 4 we present a representative sample of the procedures in EchoBox, the Echo file system layer above EchoDisk. EchoBox is the interface that Echo servers present for use by the operating systems of Echo client machines. It is a fairly conventional interface; it does not include transactions. This paper does not focus on the EchoBox interface, but some knowledge of it is required to understand the implementation issues discussed in the next section.

In Section 5 we present relevant portions of the EchoBox implementation in detail. We describe the major on-disk structures and the locking rules for accessing them. To avoid deadlock, the locking design must include strategies for claiming resources from EchoDisk. We give examples of how some update procedures in EchoBox are implemented using EchoDisk to provide atomicity, highlighting the ways in which the implementation copes with EchoDisk restrictions. We give special attention to the techniques used to ensure that EchoBox update procedures are restartable after a server failure.

Section 6 summarizes the paper and presents conclusions.

2 Simplified EchoDisk

In this section we present a simplified version of EchoDisk, the interface that hides logging from the higher levels of Echo. We define the semantics of this interface using an abstract model. We also describe a realistic implementation of the interface.

2.1 The interface

An EchoDisk is a persistent array of pages that provides a transaction capability. An EchoDisk transaction can modify a set of pages, changing any set of bytes within those pages and leaving the remaining bytes unaltered. A transaction is atomic: either all of the pages are updated, or none of them are.

EchoDisk does not supply any concurrency control, either at the page level or at the byte level. The effect is undefined if two concurrent transactions modify the same byte. It is perfectly okay for two concurrent transactions to modify different bytes on the same page.

Thus, the atomicity of EchoDisk transactions is just with respect to crashes of the EchoDisk implementation, not with respect to concurrent transactions.

A sensible client of EchoDisk is expected to use its own locks to prevent two concurrent transactions from modifying the same bytes. In addition, a sensible client will use its own locks to prevent readers from reading bytes that are being modified by a concurrent transaction, the motivation being that the transaction is in the middle of some update whose intermediate (and not yet persistent) state should not be visible to readers.

In presenting the simplified EchoDisk interface, we will give its semantics in terms of an abstract model implementation, which is not a realistic implementation. The abstract state of an EchoDisk has the following major components:

`PA`	a persistent array of disk pages
`VA`	a volatile array of buffer pool pages
`readPinCount`	an array of counters
`updatePinCount`	an array of counters
`updateQueue`	a queue of transactions that are not yet stable
`updateDemon`	a demon thread that processes `updateQueue`

This abstract state cannot be manipulated by the client directly; rather, procedures of the interface must be called.

The persistent array PA survives reboots of the computer; nothing else does. In our abstract model, updates made by a transaction are initially made to the array VA, and transaction commit copies them atomically to the array PA.

The basic procedures of the EchoDisk interface are as follows:

```
PROCEDURE OpenDisk(diskName: String): EchoDisk;
```

OpenDisk maps a disk name into an EchoDisk instance, initializes the data structures associated with that instance, and returns a handle for these structures.

Abstractly, the OpenDisk procedure copies the entire array PA into the array VA. The readPinCount and updatePinCount arrays are initialized to all zeros. OpenDisk will typically be called by the EchoBox layer during startup.

```
PROCEDURE PinPage(
   ed: EchoDisk;
   pageIndex: INTEGER): PagePointer;
```

PinPage returns a pointer to the page ed.VA[pageIndex] in primary memory, and increments ed.readPinCount[pageIndex]. As long as the readPinCount of this page is nonzero, the caller is guaranteed that dereferencing the result pointer will read the correct data. It is an unchecked error for the client to dereference the pointer when the readPinCount is zero, or for the client to modify the page (see instead the procedure UpdatePage below).

```
PROCEDURE UnpinPage(ed: EchoDisk; pageIndex: INTEGER);
```

UnpinPage decrements ed.readPinCount[pageIndex]. The client calls UnpinPage to inform EchoDisk that it is no longer using the page pointer from an earlier call to PinPage. Thus, every call to PinPage should be followed eventually by a call to UnpinPage.

```
PROCEDURE StartTransaction(ed: EchoDisk): Transaction;
```

StartTransaction creates a new transaction and returns a handle representing it. This handle is passed to other procedures.

```
PROCEDURE UpdatePage(
   tr: Transaction;
   pageIndex: INTEGER): PagePointer;
```

`UpdatePage` returns a pointer to the page `ed.VA[pageIndex]` in primary memory, and increments `ed.updatePinCount[pageIndex]`. It also adds `pageIndex` to a list of updated pages associated with the transaction `tr`. The caller is guaranteed that dereferencing the result pointer will allow reading and writing the correct page for the duration of the transaction. It is an unchecked error for the client to dereference the pointer after the end of the transaction. (There is no corresponding `UnpinPageForUpdate` because unpinning is done implicitly at the end of the transaction.)

It is okay for two transactions to make concurrent `UpdatePage` calls using the same page; the client is responsible for preventing these transactions from changing the same byte. Thus, the atomicity of EchoDisk transactions is just with respect to file server crashes, not with respect to concurrent transactions.

```
PROCEDURE DescribeUpdate(tr: Transaction;
    pageIndex: INTEGER;
    startingByteInPage: [0..PageSize-1];
    length: [0..PageSize]);
```

`DescribeUpdate` appends a tuple `(pageIndex, startingByteInPage, length)` to a list of updates associated with the transaction `tr`. This informs `EchoDisk` that the client has modified or will modify the specified byte range in `VA[pageIndex]` as part of the transaction. The client does the modification via the page pointer returned by `UpdatePage`. It is an unchecked error for the client to modify bytes outside of the ranges specified in its calls to `DescribeUpdate`.

```
PROCEDURE EndTransaction(tr: Transaction);
```

`EndTransaction` ends a transaction. Abstractly, for each tuple in the transaction, the value of the byte sequence is copied out of the page `VA[pageIndex]` and remembered with the tuple as part of the transaction's state. When this copying and remembering is complete, the transaction is appended to the queue `updateQueue`. The procedure `EndTransaction` then returns.

A single asynchronous demon thread `updateDemon`, internal to EchoDisk, repeatedly removes transactions from the head of the queue and processes them one at a time. To process a transaction, the demon processes all the tuples as one single atomic action: for each tuple, the demon copies the byte sequence value recorded during `EndTransaction` into page `PA[pageIndex]` at the position `startingByteInPage`. (The atomicity of this step may seem like magic; this is where the log enters into the real implementation, as explained in detail in the next section.) Then the demon decrements all of the `updatePageCount` entries that were incremented by the transaction.

```
PROCEDURE WaitTransaction(tr: Transaction);
```

`WaitTransaction` returns after the update demon has processed the transaction `tr`. Thus, a postcondition of `WaitTransaction` is that the changes have been made to the persistent array `PA`.

The motivation for providing separate `EndTransaction` and `WaitTransaction` operations is that certain client-level locks can be released by the client once a transaction has been serialized by `EndTransaction`, even though the transaction is not yet persistent. But the client thread cannot return successfully to its caller until the transaction has been made persistent by `WaitTransaction`.

2.2 The implementation

We now describe an implementation of simplified EchoDisk. The abstract array of pages `PA` is represented as an array of pages on disk, of the same length as the abstract array `PA`, plus a log on disk. Thus each page of `PA` has a *home* location on disk. The log stores all committed changes that have been made to `PA` and not yet recorded in the home location. During normal operation, the implementation writes committed changes asynchronously to the home location; after a crash, recovery uses the log to ensure that all committed changes are eventually written to the home location.

The update demon performs its atomic update by exploiting the log. For each tuple `(pageIndex, startingByteInPage, length, newValue)` in a transaction, the demon writes a corresponding Update record to the log. Then the demon makes the transaction atomic with respect to crashes by appending a Commit record after the last Update record: during recovery, if the Commit record is not present, the entire transaction is discarded. `WaitRecord` returns when the transaction's Commit record has reached disk.

We represent the array `VA` using a buffer pool of pages in primary memory. This buffer pool contains recently read and recently modified pages. Since there are generally many fewer buffer pool pages than disk pages, the buffer pool implementation needs to reuse buffer pool pages; that is, to change the mapping from buffer pool pages to disk pages. A buffer pool page is called *dirty* if it contains a committed update that has not been written to its home location. A buffer pool page can be reused if it is not pinned and not dirty.

We *clean* a dirty page by writing its value from the buffer pool to its home location. Without some activity to clean dirty pages, the buffer pool would quickly fill up with dirty pages. Therefore the EchoDisk implementation includes a cleaner thread that asynchronously cleans pages. The cleaner is free to clean any dirty page that is not pinned for update. (A page that is pinned for update may contain changes from a transaction that is still in progress, and our design could not undo such changes if they were to reach the home location. To allow undoing changes, twice as much information would need to be logged for each update.)

The log is kept in a fixed-size area on disk—ideally on a separate logging disk. The log is managed as a circular buffer. The start of the log is recorded in a *checkpoint record* in a fixed location on disk. The end of the log is detected by using a version stamp that is kept in a reserved location on every log page. The version stamp is incremented every time a log page is written. (We never extend a partially filled page by overwriting it; we use the next page instead.) Thus the end of the log is the point where the version stamp decreases.

This explains how the log fills up; how is log space reclaimed? The log must include all log records that would be needed by recovery. Recovery needs log records beginning with the oldest log record that updated a page which has not been cleaned since this update. Consider the log record that is at the current start of the log: we can clean the page that this record updated, and then, since this log record is no longer needed by recovery, we can advance the start of the log past this record.

The cleaner is responsible for advancing its idea of the start of the log as it cleans. Periodically it writes the current start of the log to the checkpoint record; this is called *taking a checkpoint.* Taking a checkpoint both reclaims log space and reduces the amount of log that recovery must process.

When log space runs low, the cleaner focuses on cleaning dirty pages from updates at the start of the log. When log space is plentiful, the cleaner is free to use other criteria, such as scheduling a large set of cleans to minimize disk arm movement.

The log is write-only during service. It is read during crash recovery.

Crash recovery reads the log start from the checkpoint record and then processes the log forward from this location. For each Update record that is followed eventually by a Commit record, recovery performs the update in the buffer pool. Because the Update record contains the new value of the byte sequence it changes, performing the update in the buffer pool is trivial: we just copy the byte sequence into the page. Because processing an Update record is idempotent, no additional machinery, such as per-page log sequence numbers, is needed to avoid doing it twice [2, chapter 6][10].

Since redoing updates is done entirely within the EchoDisk module, the client of EchoDisk does not have to write special redo procedures. This is simpler than logging algorithms that require redo procedures, because a redo procedure is different from the original "do" code. Of course, in return, the client is restricted to new-value logging.

3 Real EchoDisk

In this section we derive the real EchoDisk interface by removing the simplifications of the previous section one by one. Our simplified EchoDisk ignores the reality of bounded resources, so here we extend the interface and implementation to acknowledge the bounds. Some writes done by a file system do not need the strong semantic guarantees of EchoDisk, and should not have to pay for them, so we expand the interface to provide a weaker form of writing and change the implementation to match. Disks can be replicated below the level of EchoDisk, so we discuss the impact of implementing this. And servers written using EchoDisk can be replicated, with only minor changes to the interface, so we make these changes.

3.1 Reserving bounded resources

The EchoDisk implementation has limited resources—it has only a fixed number of buffer pool pages and a fixed amount of log space to work with. To prevent deadlocks over these resources between concurrent client threads, we will now augment the EchoDisk interface with procedures for reserving quantities of each resource. A client thread is expected to call these reservation procedures before it begins a high-level operation, supplying conservative estimates for the resources that will be sufficient to perform its entire operation. Requesting more resources in the middle of an operation is illegitimate because deadlock could result.

Of course, if the estimates supplied by the client are too high, EchoDisk will never be able to grant them. The client must be designed to have high-level operations with bounded size, small enough to fit in the bounds supported by EchoDisk's limited resources. To make this design feasible, the EchoDisk interface advertises bounds for each resource that it promises to support. That is, if a client thread makes a reservation request that is within the advertised bound, the request will eventually be granted.

In the interface, we split the buffer pool resource into two flavors, pages pinned for reading and pages pinned for update. The motivation for the split is that the two flavors behave differently. A page that is pinned for read may be unpinned in the middle of the client's high-level operation, and that buffer pool slot reused for pinning another page for read. But because of the lack of undo information in the log, a page that is pinned for update must remain pinned until the transaction is persistent. This will typically be near the end of the client's high-level operation, so there is no opportunity for the client to reuse the slot.

We add two procedures to control the pages pinned for reading:

```
PROCEDURE AcquireReadReservation(
   numberOfPages: INTEGER);
PROCEDURE ReleaseReadReservation(
   numberOfPages: INTEGER);
```

`AcquireReadReservation` blocks until `numberOfPages` is available in the buffer pool. The advertised bound that is guaranteed to be granted eventually is 500 pages.

Note that no handle is returned by the procedure, so there is no handle to pass in to the `PinPage` procedure. The interface assumes that all client threads are well behaved in that they will call `AcquireReadReservation` before calling `PinPage`, and that they will call `PinPage` at most `numberOfPages` times. If clients violate these conventions, EchoDisk may deadlock.

`ReleaseReadReservation` releases a client's read pin reservation. Every call on `AcquireReadReservation` should eventually be followed by a call on `ReleaseReadReservation`.

The other two flavors of limited resource are log space and pages pinned for update. Since these are associated with a transaction, we change the `Start-Transaction` procedure to accept requests for them:

```
PROCEDURE StartTransaction(
   ed: EchoDisk;
   logSpaceReservation: INTEGER;
   pinnedForUpdateReservation: INTEGER): Transaction
```

The `logSpaceReservation` parameter to `StartTransaction` is an upper bound on the log space the transaction will use. The advertised bound that is guaranteed to be granted eventually is one megabyte.

The log space used by a transaction is the sum of the log space used by all of the `DescribeUpdate` calls in the transaction. Each `DescribeUpdate` call uses log space according to the formula $\texttt{c} + \texttt{length}$, where `length` is the length of the byte sequence parameter that is passed to `DescribeUpdate`, and `c` is the constant 12. Observe that a megabyte of log space is sufficient to call `DescribeUpdate` on over a thousand byte sequences of 1024 bytes each.

The `pinnedForUpdateReservation` parameter to `StartTransaction` should be an upper bound on the number of different pages that will be pinned by calling `UpdatePage` during the transaction. The advertised bound that is guaranteed to be granted eventually is 500 pages.

`StartTransaction` blocks until the reservations can be granted. Besides competition from other client threads, reasons for blocking include waiting for the cleaner thread to clean dirty pages and remove them from the buffer pool, and waiting for the cleaner thread to take a checkpoint and free up old log space.

The resources reserved by `StartTransaction` are released implicitly. Any that were not actually used during the transaction are released by `EndTransaction`. For pinned pages that were dirtied and log space that was used, the releasing is performed by the cleaner.

3.2 Optimizing large writes

Many file system clients use the file system to store large files in simple ways. For instance, a compiler produces a huge object file. The compiler does not depend upon atomicity of its file writes—it either leaves garbage files around after a crash, or uses the `rename` system call to "commit" the new file.

Logging every new value written by such a client is an unnecessary expense. For a large enough write to a run of contiguous disk pages, the cost of doing the disk seek is minor compared to the cost of doing the disk write, so it gives better throughput to bypass the log. For shorter writes it is better to use the log.

This observation leads us to augment the EchoDisk interface with the following procedure that lets writes bypass the log, instead going directly to the home location. Without the procedure, data must first be written to the log, and then also later written to the home location.

```
PROCEDURE WritePages(
   tr: Transaction;
   startingDiskPage: INTEGER;
   buf: Address;
   bufLength: INTEGER);
```

`WritePages` has no immediate effect other than to record its parameters in the internal state of the transaction `tr`. Later, after the update demon has completed the normal processing of transaction `tr`, it processes all of `tr`'s WritePages calls. For each `WritePages` call, the demon writes the first `bufLength` bytes of the buffer `buf` to consecutive pages of `PA` beginning with `startingDiskPage`.

This write is not atomic. In case of a crash, the bytes of `PA` might contain any combination of their old value and new value. If `WaitTransaction` on `tr` returns, the write happened completely. It is illegal to call `WritePages` on a page that is pinned for read or write in the buffer pool.

The implementation of `WritePages` achieves its non-atomic writing by writing directly to the home pages, without going through the log. `WritePages` does not require the pages it writes to be in the buffer pool; if any of the pages written by `WritePages` happens to be in the buffer pool, it is overwritten with the new value.

`WritePages` complicates crash recovery. During service, a particular page may first be modified by a logged atomic transaction and then later modified by `WritePages`. In this scenario, the correct value of the page is its current home value, and we must be careful that recovery does not cause the page to revert to the earlier value in the log from the atomic transaction.

To make crash recovery work properly, the implementation of `WritePages` must do additional logging. Before writing to the home pages, the update demon appends a StartWrite record to the log. This record states that a certain run of home disk pages is about to be written. The demon forces the log to be persistent, then writes the home pages. When the home pages are all written, the demon appends a FinishWrite record to the log. This record states that the run of home disk pages has been written.

In addition to the logging by `WritePages`, the cleaner also needs to do some logging. After cleaning a page, the cleaner writes a Clean record to the log, saying that the page has been cleaned.

Here is how we use these new log records. At the start of recovery, the buffer pool is empty. As recovery processes the log forward, it must build up state in the buffer pool for each page.

- When recovery encounters an Update record that is followed eventually by a Commit record, it reads the page from home into the buffer pool (if it is not already present) and applies the modifications to the copy in the buffer pool. Recovery pins the page for update, so that it cannot be written back to its home, and also marks it as dirty.

- When recovery encounters a Clean record, it discards the page from the buffer pool if it is present.

- When recovery encounters a FinishWrite record, it discards each page in the run from the buffer pool if it is present.

- The StartWrite record is needed only for disk replication. It will be explained in Section 3.3.1.

When recovery reaches the end of the log, some pages are still in the buffer pool pinned for update and marked as dirty. We unpin these pages and start normal service, including starting the cleaner thread. Eventually, the cleaner will write each dirty page back to its home during normal service.

This algorithm keeps some pages pinned in the buffer pool during crash recovery. You might worry that the buffer pool could overflow with pinned pages during recovery, but it can't, because of the Clean records and the way we use them. Because we log the event of cleaning a page, the maximum number of buffer pool pages required is bounded by the maximum number of dirty pages that we had at any one time during the service period before the crash. Before the crash, the dirty pages all fit in the buffer pool; therefore, the number of pages required during recovery does not exceed the size of the buffer pool. Using our one-pass recovery algorithm but not exploiting the Clean records, the number of pages required during recovery would be the number of pages dirtied since the last checkpoint, which could be larger than the buffer pool.

There are two motivations for making the transaction `tr` be a parameter to the `WritePages` procedure. First, we save one log force. We don't need to force the log after the atomic part of the transaction if the `WritePages` processing is going to append a StartWrite record and force the log. Second, making the transaction be a parameter to `WritePages` gives EchoDisk the option of deciding to perform the operation by appending Update records to the log instead of bypassing it. In some cases this gives lower latency and has no significant impact on throughput.

3.3 Replication

In the Echo file system, data storage is implemented by server computers and disks. An Echo hardware configuration may have replicated disks, replicated server computers, both, or neither. Disk replication and server replication are described in the next two sections, respectively.

3.3.1 Disk replication

Disk replication is implemented within the EchoDisk module, below the EchoDisk interface. Disk replication is a configuration option. To clients of the EchoDisk interface, the disk replication is invisible; they see a single array of pages, with single-copy semantics.

Thus, replication of persistent storage is beneath the EchoBox file system layer. Another Echo paper discusses the advantages and disadvantages of this design choice [14].

The log is essential in keeping the replicas consistent. During crash recovery, the log reveals which data was in the middle of being updated, and thus may be different on the different replicas. Recovery uses the log to bring the replicas back into agreement.

In a configuration with replicated disks, both the array of home pages and the log are replicated. We use the log to keep the home pages consistent; how do we keep the logs consistent? We only append to the logs; therefore any inconsistencies will be in the tail portions of the logs. In appending to the replicated logs, the update demon lets the logs differ by at most a constant number of pages, `LogPagesInProgress`. Therefore the replicated logs are byte-for-byte identical, except for at most `LogPagesInProgress` pages at their tails.

The update demon finishes appending to both logs before allowing `WaitTransaction` to return. The unequal tail portions contain only uncommitted transactions, so we can ignore these portions during recovery.

Recovery reads from all the log replicas, starting from the position recorded in the checkpoint. If reading a log page from one replica gets a read error, we get the page from another log replica, and also attempt to rewrite the log page on the replica that got the read error. The logical end of log occurs when we hit the end of log on any replica, as determined by the per-log-page version number. However, after that point, some replicas could contain additional log pages which have good version numbers—these replicas were further ahead in their logging at the time of the crash (by at most `LogPagesInProgress` pages). These additional log pages on some replicas could confuse us on the next recovery, if not all disk replicas are accessible. To avoid this problem, we erase any additional pages that may be present by overwriting them with page images that contain bad version numbers. Instead of looking at each replica to see whether it has pages that need to be overwritten, we simply overwrite the `LogPagesInProgress` pages following the logical end of log on every replica.

If it weren't for `WritePages`, the recovery algorithm already described would be sufficient to bring the replicas back into agreement. We must extend the recovery algorithm to deal with `WritePages` in the presence of replication. Recall that StartWrite and FinishWrite log records are appended to the log while processing a `WritePages` operation. StartWrite is appended before the write begins, and FinishWrite after the write completes. During crash recovery, if the StartWrite appears in the log but is not followed eventually by the FinishWrite, then the replicas may have different values for the affected home pages; the crash may have occurred after writing a page to one replica but before writing it to the other replicas. For each page, the recovery algorithm reads the page from one replica and writes it to all the others. Since `WaitTransaction` never returned, recovery is free to choose any replica to read the page.

During service, a disk replica can become disconnected and miss some updates, and then later get reconnected. To detect that the replica does not have the most recent data, a version stamp (or *epoch*) scheme is used. Each disk replica stores an epoch number that says how up-to-date its data is. When a replica becomes disconnected, the epoch numbers of the remaining connected replicas are all set to a new value larger than any that has been used before. This process uses a multi-phase algorithm that is resilient to failures; see our other papers [5, 22] for details. When the replica reconnects, the fact that its epoch number is smaller reveals that it is out of date.

When a replica is disconnected and later becomes reconnected, we must bring its array of disk pages and its log into agreement with the up-to-date replicas. We do this concurrently with providing service. In outline, our algorithm is as follows; again, we refer the reader to our other papers for complete details. While a replica is being updated, reads avoid it, but writes (including log writes, cleaner writes, and WritePages) go to all replicas. A copier thread ensures that all pages of the out-of-date replica's disk page array and log are either copied from an up-to-date replica or overwritten with new data. When the copier thread finishes its work, we advance the replica's epoch number, marking it as up-to-date. This allows it to be used for reading pages during service and for reading the log during recovery.

If all the updates performed while a replica was disconnected still happen to be contained in the logs of the up-to-date replicas, it would be possible to exploit the log to bring the replica up to date without having to copy the entire array. This optimization would be especially important in hardware configurations with only single-ported disks, because disconnections are more likely in these configurations. With single-ported disks, a disk replica becomes disconnected whenever its directly connected server computer is down. We never implemented this optimization.

Our log is limited to a size set during server configuration. An effectively unbounded log, as might be implemented using a combination of disk and tape, would always allow the log to be used in bringing replicas up to date. Also, the log plus an old backup copy could be used to recover from media failure, a technique called *fuzzy dump* in the database literature [10].

Disk replication has a minor impact on writing pages to their home locations. The writing is performed sequentially to the different replicas. If it were overlapped, we would risk corrupting the page on all replicas (say, from power failure).

On unreplicated hardware configurations, EchoDisk is vulnerable to unreadable disk pages (media failure) in the middle of the log. If such a log read error occurs, recovery fails. The EchoDisk instance has failed to implement its specified interface. Some updates that were performed in a supposedly committed transaction may not have been done, while others may have been done, depending on which pages the cleaner was able to clean; thus the transaction atomicity semantics are violated. We could have partially protected against this vulnerability by choosing to write log pages twice on the same physical disk whenever the disk is not replicated. This would protect us against bad disk blocks, but not failures of the entire disk. The Cedar logging file system uses this technique [12]. Unreplicated disks are also vulnerable to media errors in the array of home pages. A file system scavenger [12, 18] at the EchoBox layer could partially recover from such errors (and also from unreadable log pages). We chose not to implement either of these techniques because they did not advance Echo's research goals, and because we had no practical need for them—all the hardware configurations we actually built or planned were replicated.

3.3.2 Server replication

Echo can replicate servers to improve availability. In a configuration with replicated servers, during normal operation each set of replicated disks is controlled by just one server, the primary. One or more secondary servers stand by to take control if the primary fails.

Server replication is based on the notion of disk ownership. Some disks are multi-ported and connected to several server computers; other disks are single-ported but shared via software on the connected computer. In either case, a server that owns a disk has exclusive control of it for both reading and writing. If a server other than the owner attempts to read or write a disk, the attempt will be rejected.

A primary server is elected by having each server try to claim ownership of a majority of the disk replicas: if a server succeeds in claiming a majority, it becomes the primary for that disk. In configurations with an even number of disk replicas, *witnesses* are used to break ties [25].

Ownership is subject to timeout. Therefore the current primary must refresh its ownership periodically. If the primary fails to do so for any reason (crash of primary, slowness of primary, loss of communication with disk, failure of disk) it loses ownership.

On any loss of ownership, a new election is held. To win an election, a new server must acquire ownership of a majority of the disk replicas, and ownership of a disk replica cannot change without the ownership timeout expiring. This dead time is required in order to guarantee that there is never more than one primary.

The heart of server replication is the election of a primary server. This is implemented below the EchoDisk interface. Thus both disk replication and server replication are handled by the EchoDisk implementation.

We extend the EchoDisk interface to include the concept of being primary as follows:

- `OpenDisk` blocks if this server is not the primary: it returns only when this server has become primary.
- Every other procedure of the EchoDisk interface raises the exception `Not-Primary` if this server is not the primary.
- We add a new procedure `CheckStillPrimary` which tests whether this server is the primary. EchoBox can call this procedure if it is about to perform an action which requires that this server be the current primary, such as answering a client machine query out of a cache.

When a server ceases being primary, it needs to gracefully reinitialize itself into the same effective state it had just after it was started. Losing primary status without crashing is fundamentally the same as restarting from a crash, so much of the same code is used for both cases. The update demon and the cleaner thread are simply terminated. As pages become unpinned in the buffer pool, they are discarded. It is okay to discard dirty pages because if they were from a committed transaction, the update is in the log and will be redone by the new primary during recovery, and if they were from an uncommitted transaction, we want to abandon the update, not write it back to home.

There is one tricky part about this reinitialization. EchoDisk cannot unpin buffer pool pages unilaterally, because client threads may hold pointers to these pages; only as the client calls `UnpinPage` and `EndTransaction` and the pin counts reach zero are the pages discarded.

4 EchoBox interface

In this section we present a representative sample of the procedures in EchoBox, the Echo file system layer above EchoDisk. EchoBox is the interface that Echo servers present for use by the operating systems of Echo client machines. It is a fairly conventional interface; it does not include transactions. This paper does not focus on the EchoBox interface, but some knowledge of it is required to understand the implementation issues discussed in Section 5 below.

The EchoBox interface provides a file system with hierarchical directories that is upward-compatible with Unix. The EchoBox interface is an RPC interface that provides high-level operations on files, directories, symbolic links, and hard links (as opposed to low-level operations on disk blocks). The interface includes a complete set of procedures for reading directories, files, and links, for looking up names in a directory, for accessing file properties (Unix `stat` system call), for creating, deleting, and renaming files, directories, and links, and for writing files.[3]

Echo employs a consistent distributed caching algorithm between client and server machines, in which servers keep track of which clients have cached what files and directories. The algorithm enables a client machine to cache files and directories, including caching of write-behind. The Echo distributed caching algorithm is similar to those of the Sprite and Andrew file systems [16, 17, 24], and is discussed in detail in a separate paper [21].

Thus, EchoBox is used by a client machine that handles caching. The client machine also handles failover between EchoBox server replicas. Caching and failover had several significant impacts on the design of the EchoBox interface.

A design impact relating to caching is the use of low-level file identifiers to name files and directories in the EchoBox interface. These identifiers are analogous to Unix inode numbers. The client does file path name resolution; that is, it traverses a file path name such as "/a/b/c/d" to obtain a low-level file identifier. To do this, the client consults its directory cache, fetching directories from the server as necessary.

Another design impact related to caching is the notion of a file's allocation as distinct from its logical length. Because the client cache is write-behind, many files are completely written within the client cache before being written to EchoBox. This means the client knows the eventual size of the file when creating the file using EchoBox, so the EchoBox file creation procedure accepts a recommended allocation parameter.

A design impact relating to failover is the need to support idempotent updates. To explain the EchoBox approach to idempotent updates, we must first explain the EchoBox approach to concurrent updates. The EchoBox client wants a simple, deterministic semantics, so the client's updates must be executed in a prescribed sequence. But the client also wants to overlap communication and EchoBox processing, which implies concurrency. Therefore EchoBox allows the client to make RPCs in parallel, but requires the client to supply a pipeline sequence number on each update RPC. EchoBox guarantees to execute the update RPCs in pipeline

[3]The additional functionality beyond Unix includes support for richer access control lists and support for gluing together individual file system volumes (i.e., subtrees) into a single global name space [5, 15].

order. An update RPC returns to the client machine only after the update is persistent on disk.[4]

Because of the pipeline, each client may have several RPCs in progress to a server. After a failure, the client resubmits the entire pipeline to the new primary server, starting with the oldest unanswered RPC.

To make the resubmitted sequence of updates idempotent, EchoBox requires the client to compute a version stamp for each update call. Each file or directory has a monotonically increasing version stamp; we use the Unix *ctime* for this purpose. Each update procedure in the EchoBox interface includes a version stamp parameter of type Time. For each update, the client machine is required to supply a Time that is bigger than the maximum of the ctimes of the update's operands. The client knows the ctimes of the operands because it has cached the operands. Generally, when clocks are well synchronized, the client's current clock time will be large enough to serve as the version stamp parameter, but the client may have to choose a larger time to meet the requirement. On the server, executing the update advances the ctime of all the operands to the value of the version stamp parameter.

A minor design impact relating to failover comes from the need for the client to know which server is primary. Each EchoBox procedure raises the `NotPrimary` exception if the server is not primary. A normal return from an EchoBox procedure tells the client that the server is primary.

In addition to these changes related to caching and failover, we decided to strengthen the semantics of file system update in two cases. First, EchoBox file write in the special case of appending to a file is atomic. We thought these strong semantics would be useful to clients, for instance in implementing logs. Second, EchoBox file rename is atomic; in Berkeley Unix, the rename system call guarantees only that if the target name was bound before the rename was attempted, it will be bound to something afterwards [1]. Unix programmers often use rename to commit a larger update. Providing these strengthened semantics was easy, given that we were building on EchoDisk.

We now present a representative set of procedures from the EchoBox interface.

```
PROCEDURE NewSession(): SessionID;
```

`NewSession` initializes a client machine's session with this server. The result is an identifier for some state the server holds for the client. The only session state relevant to the procedures below is the new file identifier list (see `GetFIDs` and `CreateFile`.)

[4]The reader may wonder why we used RPC for the pipeline, rather than a stream protocol such as TCP. RPC support in our environment was significantly better than TCP support [13, 27].

```
PROCEDURE ReadFile(
    sid: SessionID;
    fid: FileID;
    filePosition: INTEGER;
    nbytes: INTEGER;
    VAR (*OUT*) buf: ARRAY OF BYTE);
```

`ReadFile` copies `nbytes` of the file `fid` starting at `filePosition` into the buffer `buf`.

```
PROCEDURE GetDirectory(
    sid: SessionID;
    dir: FileID;
    VAR (*OUT*) buf: ARRAY OF BYTE)
```

`GetDirectory` returns a snapshot of the directory `dir`. The directory must fit in the buffer `buf`; otherwise, an exception is raised. Logically, the directory is a set of pairs `(name, fileID)`. The directory is actually returned in a binary format that is understood by both the client and server (using common library code).

```
PROCEDURE GetFIDs(
    sid: SessionID;
    VAR (*OUT*) newFids: ARRAY OF FileID);
```

`GetFIDs` returns a pool of FileIDs that this client uses in creating files. This pool of FileIDs permits the client to create files in its cache before calling the server (for write-behind).

```
PROCEDURE CreateFile(
    sid: SessionID;
    pipelineSequenceNumber: INTEGER;
    containingDirectory: FileID;
    name: String;
    createdFileID: FileID;
    ac: AccessControl;
    recommendedAllocationBytes: INTEGER;
    versionStamp: Time);
```

`CreateFile` creates a new file with `createdFileID` as its FileID. The directory `containingDirectory` is modified by adding the pair `(name, createdFileID)`. The Unix ctime and mtime of the file and of the `containingDirectory` are all set to `versionStamp`.

```
PROCEDURE WriteFile(
    sid: SessionID;
    pipelineSequenceNumber: INTEGER;
    fid: FileID;
    filePosition: INTEGER;
    nbytes: INTEGER;
    length: INTEGER;
    allocation: INTEGER;
    versionStamp: Time;
    VAR (*IN*) buf: ARRAY OF BYTE);
```

`WriteFile` copies the array `buf` into positions `filePosition` through `filePosition` + `nbytes` − 1 of the file `fid`. Before performing the copy, it sets the file's allocated size to `allocation`. The quantity `filePosition` + `nbytes` − 1 must be less than `allocation`. After the copy, it sets the file's length to `length`, and its ctime and mtime to `versionStamp`.

Appending writes are atomic, as are non-appending writes of less than 1024 bytes; there is no guarantee for other writes.

```
PROCEDURE DeleteFile(
    sid: SessionID;
    pipelineSequenceNumber: INTEGER;
    containingDirectory: FileID;
    name: String;
    deletedFileID: FileID;
    versionStamp: Time);
```

`DeleteFile` deletes the pair `(name, deletedFileID)` from the directory `containingDirectory`. The hard-link count on the file `deletedFileID` is decremented by one. If it is now zero, and if no client machines still have the file open, the file is destroyed. (The Echo distributed client-server caching algorithm keeps track of how many clients still have a file open [21].) The space that had been consumed by this file is not available immediately for use by other calls; rather, there is a short non-deterministic delay. The ctime and mtime of `containingDirectory` are set to `versionStamp`. The ctime of `deletedFID`, if it isn't being destroyed, is also set to `versionStamp`.

```
PROCEDURE RenameFile(
   sid: SessionID;
   pipelineSequenceNumber: INTEGER;
   fromDir: FileID;
   fromName: String;
   fromFID: FileID;
   toDir: FileID;
   toName: String;
   toExists: BOOLEAN;
   toFID: FileID;
   versionStamp: Time);
```

`RenameFile` is complicated. It removes the directory entry `(fromName, fromFID)` from `fromDir`. If `toExists` is true, it removes the pair `(toName, toFID)` from `toDir` and decrements the hard-link count of the file `toFID`. If the count is now zero, and there are no client machines that still have the file open, the file is destroyed. If `toExists` is false, the parameter `toFID` is ignored. The pair `(toName, fromFID)` is added to the directory `toDir`. The ctime and mtime of `fromDir` and `toDir` are set to `versionStamp`, as are the ctime of `fromFID` and `toFID`.

`RenameFile` is atomic.

5 EchoBox implementation

In this section we present relevant portions of the EchoBox implementation in detail. We describe the major on-disk structures and the locking rules for accessing them. Then we give examples of how some update procedures in EchoBox are implemented using EchoDisk to provide atomicity, highlighting the ways in which the implementation copes with EchoDisk restrictions. We give special attention to the techniques used to ensure that EchoBox update procedures are restartable after a server failure.

EchoBox is layered on EchoDisk. EchoBox uses EchoDisk transactions to maintain the invariant that the EchoBox file system is always in a well-formed state, that is, that the contents of the disk pages can be interpreted as a file system and the EchoBox on-disk data structures are mutually consistent.

Let us briefly recap the properties of the EchoDisk interface and its restrictions, which EchoBox must obey. EchoDisk provides new-value logging, at the granularity of byte sequences within a page. Concurrent transactions should access disjoint byte sequences, with locking being the responsibility of the EchoBox layer.

Transactions are bounded in the number of pages they may modify and the amount of log space they may consume. On-line abort is not supported, so EchoBox must avoid deadlock, over both its own resources and those of EchoDisk.

5.1 EchoBox on-disk data structures

The five major EchoBox on-disk data structures are:

- Allocation map. The allocation map contains a bit vector that records, for each page on disk, whether that page is allocated or free. The allocation map also contains several summary counters that record the number of free pages for the entire disk, and for sub-regions of the disk, to facilitate picking a good sub-region in which to search for free pages.

- Fid map. The fid map is a hash table mapping low-level file identifiers to the disk addresses of the files. File creation inserts a pair `(fileID, diskAddress)` into the hash table; file deletion removes one. When an EchoBox file system is first created, the disk size is used to choose a fid map hash table function with a fixed number of different hash values, the number being proportional to the disk size. Each hash value is preallocated its own disk page, and this page stores every pair whose `fileID` hashes to that value. Overflow of a page is handled by allocating an additional page and chaining it to the original, without changing the hash function.

- Files and directories. Each file and directory has a header page. The disk addresses stored in the fid map are addresses of header pages. A header page contains file properties, including the access control list, and a file run table. Large fragmented files (or directories) may have multiple run tables in multiple header pages, chained together.

- Hashed directories. At the lowest level, a directory is stored in an ordinary file, so we already have machinery for growing and shrinking directories. An Echo directory consists of a header, followed by an array of hash table pointers, followed by a unordered sequence of directory entry records.

  Each element of the array of hash table pointers is the head of a linked list of entry records whose names hash to the same hash value, with each entry record having a `next` field; thus, hash collisions are handled by *external chaining*. The array of hash table pointers has a fixed size, but its size can change during directory reorganization (described in Section 5.3 below).

Directory entry records are of variable length to accommodate names of different sizes. The format of the sequence of directory entry records is upward-compatible with Berkeley Unix, and the system call for reading a directory, `getdirentries`, returns this portion of the Echo directory. Directory entry records may cross page boundaries (unlike Berkeley Unix). All unoccupied entry records are linked together in a free list, whose head is stored in the directory header.

This directory representation relies on the EchoDisk atomicity mechanism: for directories that are larger than a page, insertion and deletion may modify multiple pages. However, the number of directory pages modified by a single insertion or deletion is bounded, even for large directories.

- Orphan list. An orphan file is a file that is no longer in the name space, in that no entry in any directory points to it, but it is still open on some client machine and therefore must still be kept in existence. (We use the term "orphan" because an orphan file has no parent directory.) The orphan list lists all orphan files. When the Echo distributed client-server caching algorithm [21] reveals that nobody has an orphan file open, EchoBox can remove the file from its orphan list and delete the file, returning its pages to the disk space allocator and removing it from the fid map.

  EchoBox represents the orphan list as a doubly-linked list that runs through the header pages of the orphan files, plus a distinguished file that is the head of the list. We considered other representations for the orphans, such as moving them to a special directory, but we rejected representations that could, in the worst case, require allocating storage in order to add an orphan to the data structure, since we must be able to delete files when the file system is full, and this may create orphans.

The use of EchoDisk had several effects on the design of the EchoBox on-disk structures:

- We were able to keep the disk allocator's summary counters on disk, since logging made updates to these counters cheap. Otherwise, the counters would have to be recomputed during EchoBox recovery, slowing recovery down.
- Similarly, we were able to keep the orphan list on disk. This avoided storage leaks or the alternative of scanning the entire fid map on system startup, looking for files whose hard-link count is zero.
- We were able to employ hashed directories without concern that they might need to be rebuilt after a system crash.

- We considered the idea of making file headers smaller than a disk page, but rejected it as unnecessary because our pages are small. But byte level logging would have made it easy to do this if we had chosen a larger page size.

We designed the EchoBox on-disk structures on the assumption that EchoDisk was providing high reliability via replication. The structures have some redundancy, but not enough to support a complete file system scavenger. If we had needed to support unreplicated disk configurations, we would have had to design more systematic redundancy at the EchoBox level.

5.2 Locking

We now present the locking at the EchoBox layer. This locking is used to ensure the data independence of concurrent transactions, and to ensure that queries do not access data that is being modified by a transaction. Executing an EchoBox update RPC can involve modifying files and directories, the fid map, the allocation map, and the orphan list, so we need locking for each. The locks are not persistent; they exist only in the primary memory of the server. Locks are claimed in a predefined order to avoid deadlock. Thus no mechanism is required for deadlock detection at runtime.

Each file and directory has a readers/writer lock. Each procedure in the EchoBox interface must acquire a lock on each of its FileID operands; a read lock if the procedure only reads, a write lock otherwise. The lock is held for nearly the entire duration of the procedure. (Full details are provided later in this section.) The file and directory locks ensure the data independence of concurrent calls on the EchoBox interface.

Some procedures have several FileID operands, so the locks must be acquired carefully to avoid deadlock. The operands are sorted by order of increasing file identifier, and locks are acquired in that order.

The fid map and the allocation map are shared data structures with the potential for high concurrency. Therefore EchoBox uses specialized locking for them.

For the fid map, we have an array of exclusive locks, one for each value of the hash function. To map a file identifier to its disk address, we first compute the hash function of the file identifier and lock that entry of the array. We access the fid map and then immediately release the lock. The immediate release does not lead to trouble because the readers/writer locking at the file level prevents conflicting fid map access to a file identifier.

Locking for updates to the fid map is different than for queries. The organization of the data structures within a single fid map page requires us to prevent concurrent updates within a fid map page, in order to ensure the data independence of concurrent transactions. Therefore fid map updates acquire a lock just as for queries, but hold the lock until after `EndTransaction` has returned. Because this lock is held for more than the duration of the fid map update, deadlock avoidance requires that when a transaction does multiple fid map updates the locks be acquired in a specific order (like the readers/writer locks). In our implementation, we deferred the fid map updates (and the locking) until just before we called `EndTransaction`. This both reduces the time that the lock is held and makes it easier to achieve an ordering that is deadlock-free. Since each value of the fid map hash function has its own lock and there are many values, the probability of one transaction having to wait for another because of needing the same lock is small.

The allocation map involves our most complicated locking. The complexity has three sources. First is our desire for high concurrency. Second, when deleting a file, we must not deallocate its storage for re-use by a concurrent transaction until the delete is committed. Third, there is a serious mismatch between the byte-level granularity of the EchoDisk interface and the desire of the allocator to modify its bitmap with bit-level granularity.

To explain the allocation map we'll first describe the volatile data structures, including locks. Then we'll describe how EchoBox's internal allocate, deallocate, and commit procedures use these data structures.

We maintain two versions of the allocation bit vector. One version, `pv`, is kept on disk (and in the buffer pool as needed). The other version, `vv`, is kept only in primary memory. At system startup, `vv` is initialized to be a copy of `pv`. Allocations are done by searching and modifying `vv`; the modifications are propagated to `pv` at end of transaction. Deallocations are done in the reverse order: first to `pv` at end of transaction, then to `vv`. Keeping two versions of the allocation bit vector solves the problem caused by the byte-level granularity of EchoDisk, as we'll explain later.

There is an exclusive lock associated with `vv`; it protects all accesses to `vv`. There is another exclusive lock associated with `pv`; it both protects accesses to `pv` and serializes all EchoBox calls on `EndTransaction`.

As an EchoBox update executes, it calls internal procedures to allocate and deallocate disk pages. The allocate procedure acquires the `vv` lock and searches `vv` for free pages. It allocates pages by flipping bits in `vv` and remembering the pages on a list associated with the update. When done, the procedure releases the `vv` lock and returns. A single EchoBox update may call the allocate procedure multiple times, acquiring and releasing the `vv` lock each time. The deallocate procedure merely records pages on a list associated with the update, without updating `vv`; no locking is involved.

To commit a transaction, EchoBox calls the allocator's commit procedure. This procedure first acquires the `pv` lock and processes the list of allocated pages, making the same updates to `pv` that the allocate procedure made earlier to `vv`. It also processes the list of deallocated pages, flipping bits of `pv` corresponding to deallocated pages. Then it calls `EndTransaction`, and after `EndTransaction` returns it releases the `pv` lock. If the list of deallocated pages was not empty, the commit procedure acquires the `vv` lock, makes the same updates to `vv` that it made to `pv`, and releases the `vv` lock. Finally the commit procedure returns.

We said it would be complex, and it is; why does it work? `pv` records exactly the allocations and deallocations performed by committed transactions. `vv` sees allocations before `pv`, but `pv` gets the same information before a transaction commits; `pv` sees deallocations before `vv`, but this only means that there is a delay before pages can be reused. When the system is quiescent, `vv` and `pv` are equal.

Thus, `vv` is a superset of `pv`. It includes allocations by committed transactions plus allocations by in-progress (not yet committed) transactions. Because the allocate procedure searches `vv` for free pages, a call on allocate will skip over pages that have been claimed by another in-progress transaction. We can regard the vector `vv` as being the disjunction of `pv` and a vector of lock bits that prevent in-progress transactions from allocating the same pages. The lock bits have the same granularity as the allocator, namely, one per page.

Concurrent transactions can modify the same byte of `vv`, but the corresponding updates to `pv` take place serially and in the same order as transaction commit, under `pv`'s exclusive lock.

We said that the complexity of allocation map locking has three sources: concurrency, deallocation, and bit-level data structures; we'll now discuss each source in turn.

- To increase concurrency, we wanted to permit multiple concurrent transactions to allocate storage. We force them to serialize only for `EndTransaction`. Our desire for high concurrency is motivated by the fact that our server hardware is a shared-memory multiprocessor, the Firefly [29]. On a uniprocessor, it would be adequate for an update to acquire a long-term exclusive lock the first time it needs to allocate or deallocate storage, releasing it only after `EndTransaction`. By this point in the update, most disk input (e.g., fetching file header pages and directory pages) has already been done, so the transaction is being processed at CPU speed.

- Deallocation is deferred until `EndTransaction` because we cannot permit another concurrent transaction to allocate those pages until the deallocating transaction has committed. If we did not defer, but instead changed `vv` immediately, another transaction could reallocate the pages and commit first, before the deallocating transaction committed. If the system crashed at that instant, the file system would be malformed in that some pages would be allocated to two files. Deferring work until `EndTransaction` is a common technique for handling actions that cannot be undone.[5]

- Having two versions of the allocation bit vector is the result of the byte-level granularity of EchoDisk updates and logging. If the granularity of EchoDisk updates and logging were bit-level, that is, if concurrent transactions could modify bits in the same byte without interference, then we would not need the vector `vv`. The allocate procedure would modify bits of the vector `pv` directly in the buffer pool, with the changed bits being logged at `EndTransaction`. Of course, recovery would also have bit-level granularity. Deferring deallocation until `EndTransaction` would still be necessary.

In addition to maintaining the persistent vector `pv`, the allocator also maintains several persistent summary counters that record the number of free pages in the entire disk and in sub-regions of the disk. To increase concurrency, we defer the update of these persistent counters until `EndTransaction`; they are changed by the allocator's commit procedure while it holds the `pv` lock.

There is yet another locked resource in the allocation module. We cannot permit an update to run out of disk storage in the middle, because we lack on-line abort. So near the beginning of an update, before calling `StartTransaction`, we reserve with the allocation module the right to allocate some number of pages. The number we reserve is an upper bound based on the parameters to the update; at the end of the update, we release any we did not use. The reservation does not commit the allocator to specific disk pages, only to this update's right to allocate up to the reserved number.

The orphan list is protected by an exclusive lock. `DeleteFile` and `RenameFile` obtain this lock if the relevant file is becoming an orphan. The lock is also obtained by the background orphan list demon, which deletes an orphan file once it has been closed on all client machines.

This concludes our description of the locks at the EchoBox layer. Figure 1 summarizes the order in which the EchoBox locks and the EchoDisk resources are claimed during an EchoBox update procedure.

[5] In principle, deallocating a file could be made undoable by logging the entire value of the file, at great expense.

```
Acquire ReadPinReservation
    Acquire write locks on fileIDs, in fileID order
        Lookup fileIDs in fid map, acquiring and releasing
                    the fid map hash code lock
        Acquire orphan list lock (DeleteFile, RenameFile,
                    and orphan list demon only)
            Acquire allocation reservation
                StartTransaction with parameters
                        reserveLogSpace and
                        reservePinnedForUpdate
                    Calls on PinPageForUpdate, DescribeUpdate. Also
                            calls to insert/delete pairs in fid map and
                            to allocate/deallocate pages.
                    Acquire fid map hash code lock
                        Perform fid map deferred work—actual insertions
                                and deletions
                        Acquire allocator pv lock
                            Perform allocator deferred work—update pv to
                                    reflect page allocations and deallocations
                            EndTransaction
                        Release allocator pv lock
                        Perform allocator deferred work—update vv
                                to reflect page deallocations
                    Release fid map hash code lock
            Release allocation reservation
        Release orphan list lock
        WaitTransaction
    Release write locks on fileIDs
Release ReadPinReservation
```

Figure 1: Locking for an EchoBox update

Several locks are released in the interval between `EndTransaction` and `WaitTransaction`. This was the motivation for separating the procedures. We don't have any hard evidence for the benefit of this optimization.

EchoDisk's design anticipated that each EchoBox update procedure would compute accurate upper bounds on its usage of the three EchoBox resource classes, and pass these estimates to EchoDisk. In this way EchoBox could make the best use of EchoDisk resources and improve EchoBox's concurrency. In fact, most EchoBox update procedures simply pass EchoDisk the largest resource bounds that EchoDisk guarantees to grant; we performed a coarse upper bound computation to show that the procedures will not exceed these limits. Performing a more precise upper bound computation would have made EchoBox fragile, since the bounds depend upon EchoBox implementation details.

An EchoBox query procedure (i.e., a read-only procedure) acquires and releases locks in the same pattern as above, but with some operations omitted. A query is a read-only transaction, but there is no reason for it to call `StartTransaction`, because EchoDisk transactions do not provide concurrency control. To provide mutual exclusion from updates, a query acquires a read lock on its FileID argument. (No query procedure has more than one FileID argument.) A query will need to pin pages for reading, so it needs to call `ReadPinReservation` to avoid deadlocking over that resource. The number it passes to `ReadPinReservation` is a small number based on the parameters to the RPC and the code path of the query. For example, the ReadFile RPC passes the number 2, which lets it pin one file header page and one file data page concurrently. During the execution of ReadFile, it may access more than one header page and more than one data page, but it never needs to keep more than one of each pinned, since it processes the data sequentially.

The locking order for an EchoBox query procedure is summarized in Figure 2.

5.3 Use of EchoDisk transactions by EchoBox

We now describe how the implementations of several EchoBox update procedures make use of EchoDisk transactions to keep the on-disk structures consistent.

The `CreateFile` procedure is executed using one or more EchoDisk transactions. Usually, the work is accomplished in a single transaction, but when `recommendedAllocationBytes` is larger than a fixed threshold, additional transactions are used to grow the file to that size. The fixed threshold keeps the work done in a single transaction bounded within the limits dictated by EchoDisk.

`CreateFile`'s first transaction does all of the following actions. It creates a new file with file identifier `createdFileID`. Creating the new file calls the

```
Acquire ReadPinReservation
    Acquire read lock on the fileID
        Lookup fileID in fid map, acquiring and releasing
                the fid map hash code lock
        Call PinPage and UnpinPage to read pages
    Release read lock on the fileID
Release ReadPinReservation
```

Figure 2: Locking for an EchoBox query

disk space allocator to obtain space for the file header page plus a portion of the `recommendedAllocationBytes`, up to the fixed threshold. The file header page is filled in with the access control and other properties, and its run table is set to describe the space allocated so far. (We describe this update to EchoDisk as an update to the entire header page, rather than trying to tell EchoDisk precisely what bytes changed.) The FidMap module is called to add the pair `(createdFileID, disk address of file header page)` to the fid map. Note that if the fid map page overflows, the fid map module will call the allocator to obtain another page. The pair (`name`, `createdFileID`) is entered into the directory `containingDirectory`. If the directory was already full, adding this entry requires growing the directory, which entails a call to the allocator. The ctime and mtime of the new file and the containing directory are all set to `versionStamp`.

Next, if `recommendedAllocationBytes` is larger than the fixed threshold, we execute one or more additional transactions: each grows the file, by a maximum of the fixed threshold, until `recommendAllocationBytes` is reached.

The `WriteFile` procedure is executed with one or more transactions. If the number of bytes being written is large, or if the new allocation size is growing or shrinking the file by a large amount, the work is split into several transactions. First, the file is grown or shrunk to the new allocation size specified in the call. Doing this requires one or more transactions, using the same fixed threshold as for `CreateFile`. Next, the bytes from the buffer are written into the file, using the non-atomic EchoDisk `WritePages` procedure. We wait for those writes to become persistent, by calling `WaitTransaction`. Then we execute a transaction to set the new length of the file to the parameter `length` and the ctime and mtime of the file to `versionStamp`. If the number of bytes being written is small, and if the new allocation size is growing or shrinking the file by a small amount, a single transaction does all of the work. The bytes of the buffer are copied into the file simply by pinning pages for update, assigning to the pages via the pin pointer, and calling `DescribeUpdate`.

The `DeleteFile` procedure is executed with a single transaction. This transaction removes the file from the `containingDirectory`. The ctime and mtime of the `containingDirectory` are set to `versionStamp` and the ctime of the file to `versionStamp`. The hard-link count of the file is decremented. If it has not reached zero, we are done with `DeleteFile`. Otherwise, what we do depends on whether some client machines still have the file open and on whether the file is larger than the fixed threshold. If no client has it open and the file is smaller than the threshold, then we go ahead and destroy it in this transaction, by removing it from the fid map and by returning its pages to the disk space allocator. Otherwise, this transaction moves the file onto the orphan list, and `DeleteFile` returns.

When the last client machine closes the file (or immediately if there were no clients), the background orphan list demon deletes the file. The deletion is done in stages. First, if the file is larger than the fixed threshold, one or more transactions are executed: each truncates the file by the fixed threshold (or to zero size), returning the freed pages to the allocator. A final transaction removes the file from the orphan list, returns its remaining pages to the allocator, and removes it from the fid map. The background orphan list demon is created during system startup.

The `RenameFile` procedure is executed in a single transaction. For the case where an object with the new name already existed, it is deleted, using logic similar to `DeleteFile`.

Directory reorganization is not tied to a procedure in the EchoBox interface—it is done in background—but it is an interesting update nonetheless. Reorganization improves the efficiency of directory access. Reorganization is triggered as a side effect of a large change in the directory size since creation or the last reorganization.

The background thread performing a directory reorganization holds a write lock on the directory. Reorganization is done by enumerating the directory and building an image of the desired new directory in virtual memory. Using a single EchoDisk transaction, the image is copied on top of the old directory using normal EchoDisk operations. This same transaction also grows or shrinks the underlying directory file as necessary.

The boundedness of EchoDisk transactions places a limit on the size of directory that we can reorganize, but the limit is large: we can handle 256K byte directories. We were prepared to handle larger directories by growing them incrementally without reorganization, but we never saw such a large directory except in our tests.

5.4 Failover and restartability of updates

In this section we explain how failover between servers is supported by EchoBox, especially how updates are made restartable.

EchoBox relies on the concept of "primary" provided by the EchoDisk interface. EchoBox executes a request from a client machine by calling on the EchoDisk procedures. If one of these EchoDisk procedures raises the exception `NotPrimary`, then the exception is propagated all the way back to the client machine.

The client machine then probes all the servers to find out which is the new primary, and resubmits the call to the new primary. Query calls (i.e., read-only procedures) are trivially restartable. What about updates?

We have already described how the EchoBox interface is designed to support the restartable updates required for server replication. To review briefly, update procedures are ordered using a pipeline sequence number specified as a parameter to the update; after a failure the client resubmits the entire pipeline starting with the oldest unanswered call. Each file and directory stores a ctime, which EchoBox uses as a version stamp. Each update procedure has a `versionStamp` parameter, and executing the update advances the ctime of the operands to this value.

Now, let us explain how the EchoBox implementation makes it possible to restart updates.

For updates that EchoBox performs in a single transaction the story is quite simple. Each update procedure compares the `versionStamp` parameter supplied by the client against the ctime of any of the update's file or directory operands. If the `versionStamp` parameter is greater than the stored ctime, the procedure performs the update; otherwise the procedure returns without performing the update, since the ctime shows it was already done.

For the two updates that EchoBox performs in several transactions, the story is slightly more complex:

For `CreateFile`, if the `versionStamp` parameter is equal to the server's stored ctime, then we know that the first transaction of `CreateFile` occurred, creating the new file and adding it to the directory, but we do not know whether the transactions required to grow the file to `recommendedAllocationBytes` were done. The server looks at the size of the stored file and grows it if necessary, using one or more transactions. If the `versionStamp` parameter is either strictly less or greater than the stored ctime, the standard rule above applies.

For WriteFile, if the `versionStamp` parameter is less than the server's stored ctime, it is possible that some of the work of `WriteFile` already has been done, even though the final transaction, which sets the file's length, mtime, and ctime, did not complete. The work that may already have been done consists of growing or shrinking the file to the size specified by the `allocation` parameter and then copying bytes of the buffer into the file. Observe that this work is idempotent: doing it again achieves the same result. So when the `versionStamp` parameter is less than the stored ctime, we simply execute the `WriteFile` call from the beginning, without trying to figure out whether some of it already was done. If the `versionStamp` parameter is greater than or equal to the server's stored ctime, then the earlier `WriteFile` call completed in its entirety (the standard rule above).

For all the other updates, the fact that the ctime of the operands is advanced as part of the same transaction that does the work of the update makes the advancing of the ctime a clear demarcation point. Even though `DeleteFile` and `RenameFile` can move a file to the orphan list for subsequent processing, the movement onto the orphan list is done in the same transaction that advances the ctimes. And because the orphan list is persistent, the orphan demon that is started on the new primary server will find the list and process it.

6 Summary and conclusions

We believe our overall design based on the EchoDisk interface was a success. The restrictions we made in the EchoDisk interface are sensible and appropriate for building a file system.

To recap, EchoDisk provides a simple but constrained interface for performing atomic updates. EchoDisk supports new-value logging, at the granularity of byte sequences within a page. Because pages that are updated during a transaction must be kept pinned in primary memory until the transaction commits, and because the log has a fixed size, transactions have a bounded size. EchoDisk itself provides no concurrency control to prevent concurrent transactions from accessing the same byte. On-line abort is not supported. These restrictions permit an extremely simple implementation of EchoDisk; simpler, we believe, than for any less restricted interface.

The EchoBox implementation was designed to satisfy the constraints of EchoDisk. The lack of on-line abort means that EchoBox had to be carefully designed to avoid deadlock. File systems without logging are usually designed to avoid deadlock too, so our use of logging has made this aspect of the design neither easier nor harder. We claim that the nature of file system updates, in particular the fact that all transactions are initiated by the EchoBox layer, which supports only a fixed set of possible update operations, makes our deadlock-avoidance design feasible and sensible. We contrast the file system case with that of general databases, in which users control the code that runs in transactions, and thus nothing prevents the user code from deadlocking.

The EchoBox implementation exploits EchoDisk transactions to maintain the invariant that the file system is always in a well-formed state; that is, that the contents of the disk pages can be interpreted as a file system and the file system data structures are mutually consistent. The bounded size of EchoDisk transactions was not a significant impediment to implementing EchoBox. The bound is large enough that complicated file system updates, such as renaming a file and creating a file (except for the final part of growing the file to a recommended allocation), can be performed in a single atomic transaction. If these complicated updates did not fit in a single transaction, we would forfeit the important advantages of transactions for simplifying file system code.

We did have to break up some large file system updates into a sequence of transactions—namely, those updates that involved growing or shrinking a file by a significant amount. But each transaction in the sequence leaves the file system data structures (e.g., file run table and allocation bit map) in a mutually consistent state.

EchoDisk transactions also helped with implementing server replication. After a primary server crash, client machines resubmit any unacknowledged EchoBox updates to the new primary server. The new primary must figure out which updates have already been done or partially done. Our solution uses version stamps plus some additional information. Because each EchoBox update is either atomic or composed of a small sequence of atomic actions, only a small number of cases had to be considered in the implementation.

Within EchoDisk, logging is the basis for data replication. For crash recovery, the log reveals which data was in the middle of being updated at the time of the crash, and therefore might be different on the different replicas. Recovery exploits the log to bring the replicas into agreement.

One EchoBox data structure does not quite fit the EchoDisk framework, namely, the file allocation bit map. For this structure a bit-level granularity of logging, rather than byte-level, would have been preferable.

EchoDisk provides new-value logging. The IBM AIX file system also seems to be based on new-value logging, provided as part of their virtual storage management, which is more elaborate than EchoDisk. The published descriptions of their design are vague on how the necessary locking at the file system layer, beyond that provided by their virtual storage manager, is achieved [7, 8].

Transarc's Episode file system uses old-value/new-value logging with a bounded log and support for abort. Episode has a radically different approach to locking: instead of defining a hierarchical locking order with two-phase locking to guarantee the data independence of different transactions, Episode merges concurrent transactions that access the same data items into a single umbrella transaction. This umbrella transaction either commits or aborts in its entirety [9].

Echo's use of new-value logging allowed a clean separation of our overall design, into a layer that implements logging and recovery (EchoDisk), and a layer that implements file system accesses and updates (EchoBox). The EchoBox layer is not involved in recovery. We can contrast our approach with operational logging, in which redo procedures must be written. We believe that operational logging with redo procedures is harder to implement correctly than our new-value logging. The redo procedures are, in general, different from the original "do" procedures, requiring more code. And they are more complicated than the simple byte copying that recovery performs in new-value logging. In return, operational logging allows more cleverness, including more compact logs and fancier type-specific locking [23].

We would like to see a file server design of similar aspirations to ours based on operational or multi-level logging [20], worked out to a similar level of detail.

Acknowledgements

Extensive comments from Mark R. Brown improved the organization of the paper.

References

[1] Berkeley Unix 4.3 Reno release. *rename(2) system call manual page*, 1990.

[2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] Anupam Bhide, Elmootazbellah N. Elnozahy, and Stephen P. Morgan. Implicit replication in a network file server. In *Proc. Workshop on the Management of Replicated Data*, pages 85–90. IEEE Computer Society, November 1990.

[4] Anupam Bhide, Elmootazbellah N. Elnozahy, and Stephen P. Morgan. A highly available network file server. In *USENIX Winter Conference Proceedings*, pages 199–205. USENIX Association, 1991.

[5] 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.

[6] Mark R. Brown, Karen N. Kolling, and Edward A. Taft. The Alpine file system. *ACM Transactions on Computer Systems*, 3(4):261–293, November 1985.

[7] A. Chang, M. F. Mergen, R. K. Rader, J. A. Roberts, and S. L. Porter. Evolution of storage facilites in AIX Version 3 for RISC System/6000 processors. *IBM Journal of Research and Development*, 34(1), January 1990.

[8] Albert Chang and Mark F. Mergen. 801 storage: Architecture and programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988.

[9] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham. The Episode file system. In *USENIX Winter Conference Proceedings*, pages 43–60. USENIX Association, 1992.

[10] James Gray. Notes on data base operating systems. Research Report RJ 2188 (30001), IBM Research Laboratory, San Jose, California, 1978.

[11] Theo Haerder and Andreas Reuther. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[12] Robert Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proc. 11th Symp. on Operating Systems Principles*, pages 155–162. ACM SIGOPS, November 1987.

[13] 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 Computer Society, November 1992.

[14] 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 Computer Society, November 1990.

[15] 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 Computer Society, September 1989.

[16] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, 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.

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

[18] Butler W. Lampson. Hints for computer system design. *IEEE Software*, 1(1):11–28, January 1984.

[19] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Luiba Shrira, and Michael Williams. Replication in the Harp file system. In *Proc. 13th Symp. on Operating Systems Principles*, pages 226–238. ACM SIGOPS, October 1991.

[20] David Lomet. MLR: A recovery method for multi-level systems. In *Proc. SIGMOD Conference*, pages 185–194. ACM SIGMOD, June 1992.

[21] Timothy Mann, Andrew Birrell, Andy Hisgen, Charles Jerian, and Garret Swart. A coherent distributed file cache with directory write-behind. Research Report 103, Systems Research Center, Digital Equipment Corporation, June 1993.

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

[23] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[24] 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.

[25] Jehan-Francois Pâris. Voting with witnesses: A consistency scheme for replicated files. In *Proc. 6th Intl. Conf. on Distributed Computer Systems*, pages 606–612. IEEE Computer Society, 1986.

[26] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proc. 13th Symp. on Operating Systems Principles*, pages 1–15. ACM SIGOPS, October 1991.

[27] Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[28] 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.

[29] Charles P. Thacker and Lawrence C. Stewart. Firefly: A multiprocessor workstation. In *Proc. 2nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM and IEEE Computer Society, October 1987.