Service Interface
and Replica Management Algorithm
for Mobile File System Clients

Carl D. Tait and Dan Duchamp

Computer Science Department
Columbia University
New York, NY 10027
tait@cs.columbia.edu, djd@cs.columbia.edu

Abstract
Portable computers are now common, a fact that
raises the possibility that file service clients might move
on a regular basis. This new development requires re-
thinking some features of distributed file system design.
We argue that existing approaches to file replica man-
agement would not cope well with the likely behavior
of mobile clients, and we present our solution: a lazy
“server-based” update operation. This operation facil-
itates fast, scalable, and highly fault-tolerant imple-
mentations of both read and write operations in the usual
case. To cope with the weak semantics of the update op-
eration, we propose a new file system service interface
that allows applications to opt for “UNIX semantics” by
use of a slower, less fault-tolerant read operation.

1 Introduction
This work investigates how to maintain replicas in a dis-
tributed file system, especially one supporting mobile
clients. While the topic of replica management within
file systems has received so much attention that one
might think there is no design point left unconsidered
[3, 16, 14, 2, 9, 18, 1, 10, 4], the notion of mobile clients is
a new development that alters operating circumstances
and therefore suggests new designs. The idea of mo-
 bile file service clients stems from the exploding popu-
ularity of portable computers. We argue that existing
file replica management schemes would not cope well
with this new circumstance, and we present our solu-
tion: a lazy “server-based” write operation and a new
service interface that allows applications to select strong
or weak consistency semantics.

The next section explains our design. The design is
not implemented, so the evaluation in Section 3 consists
of empirical data validating our ideas and a qualitative
comparison with existing systems.

Several assumptions underpin our thoughts:
1. Latency of remote operations degrades according
to the distance between hosts.
2. The load presented to the file service is that of
“engineering/office applications.”¹ This workload,
which is the “standard” workload assumed by most
research file system designs, consists of sequential
write-read sharing, but little simultaneous sharing.
We will argue that existing file system designs have
not taken maximum advantage of this assumption.

3. A client maintains a file cache of modest size. Section
3.2.3 elaborates on the required size.

4. The file system supports a standard hierarchical
name space.

Our model of operation is that a client moves around,
perhaps to areas it has never visited before; once in a
new milieu, it will negotiate with some local machine to
become a new replication site (server) for its files. That
site will then be added to the client’s set of servers.

Starting from this basis, we suppose that the exis-
tence of mobile clients will lend extra importance to
the design goal of minimizing synchronous multi-server
operations; a consequence of the first two assumptions
is that file systems that make frequent use of “global
communication”² will perform poorly if mobile clients
are mapped to a static set of servers. The reason is that
the latency of synchronously contacting several servers
can be no less than the latency of the slowest (farthest)
server. When a client moves away from some or all of
its current set of servers, file service operations imple-
mented with global communication will be slow.

¹Here we are referring to Ousterhout’s division of file access
patterns into three classes [15]:

(a) Scientific jobs—large datasets read and written sequentially.
(b) Transaction processing—frequent sharing, with the require-
ment of one-copy serializability.
(c) Engineering/office applications—many applications using
many small files, with little concurrent sharing.

²We define a global communication as a synchronous operation
in which more than one server must be contacted. Examples
are Coda’s close operation, which contacts all reachable servers
[9], or Echo’s write operation, which must contact a majority of
servers.
2 Approach

2.1 Server-Based Writes

To eliminate global communication we take two steps. First, we use a primary-secondary replication scheme; the client communicates only with its primary. Second, we employ write-back caching with the server, not the client, choosing when up dates are copied from the cache. We call this approach server-based writing, and it has two consequences. First, updates are propagated back to the replicas asynchronously, thus making the write operation fast. Second, by making pickups only at moments of its choosing, the file service has the opportunity to reduce (but not eliminate) the "inconvenient" times at which it accepts updates. An example of an inconvenient time for an update is when the service is partitioned; separate clients can produce conflicting updates during partitions.\(^3\)

The client treats whichever server performed the last server-based write as its primary. Because of this, the primary-secondary organization can be changed transparently simply by performing a server-based write from the new primary. Occasionally, as the result of client movement or server failure, a new primary will take over and start making pickups.

Although writes are server-based, it is possible for a client to request immediate pickup from its primary. We call this a forced write. Forced writes are necessary to avoid blocking due to a full cache during periods of heavy update activity.

After a pickup, the primary saves the updated files in non-volatile storage and multicasts them to the secondaries. Once some number (N) of secondaries have acknowledged saving the updates, the primary informs the client that the updates may be discarded from its cache. This purge notice guarantees that the updates have been replicated at the primary and N secondaries, and so the data is replicated widely enough to be N-fault-tolerant. Until it receives the purge notice the client must hold recent updates in its cache. Purge notices are typically piggybacked on the next pickup request. Waiting to purge is the client's only obligation to the service—it could even habitually forget the name of its primary, with the only negative consequence being that it must wait until the next pickup to learn of a server to read from.

If not enough secondaries acknowledge the updates, then the primary does not send a purge notice to the client. By this means, the service has some latitude (limited by the size of the client's cache) to "wait out" failures of secondaries without any apparent service disruption.

We call this model the "lazy tree" because of the lazy propagation of updates within the tree-like structure; see Figure 1 for a depiction of this organization.\(^4\)

\(^3\)Although this will never happen if clients always use the strict form of read described below.

\(^4\)Asynchronous update propagation from primary to secondary is not uncommon in file system designs. What sets apart our work is how we achieve fault tolerance and fast operation in the normal case through the use of (1) asynchronous update propagation from client to primary, and (2) allowing an application to choose either lazy update or one-copy semantics on a per-read basis, as described below.

---

![Figure 1: Lazy Tree Organization](image)

C is the client, P the primary, and S denotes the secondaries.

During normal operation, the pickup interval can be dynamically modified, based on the client's update activity. (Experimental results in Section 3.2.3 suggest bounds on the pickup interval.) The longer the pickup interval, the more short-lived files and associated update operations are never seen by the servers. Previous work on file caching [13, 6, 8] has shown that having clients lazily write back updates substantially improves scalability.

2.2 Interface

Lazy operation is unsuitable for applications that share a file simultaneously or that perform write-read sharing with the read closely following the write. Such applications will typically want to see each other's updates as soon as possible. In our design, an application program can choose between weak or strong semantics on a per-use basis. We propose a file service interface different from the usual read/write. Our interface has three calls:

- write
- loose_read
- strict_read

Two kinds of read operation are provided: loose and strict. There is no guarantee attached to the value returned by the loose read, although the implementation of write makes a "best effort" to promptly spread the latest value. In contrast, the strict_read call returns
the most recent _consistent_ value (the latest value written by a strict reader) by contacting all servers and retrieving the most up-to-date copy it finds. If the servers know of other clients that have read the file into their caches with a previous _strict read_, then _strict read_ must also read from those caches before choosing the most recent value. The next section explains a hint mechanism designed for a low-sharing environment that substantially improves both the performance and the fault tolerance of the _strict read_ operation.

Exclusive use of _strict read_ and _write_ ensures “one-copy UNIX serializability” (1USR), which duplicates the behavior that one would observe in a centralized UNIX system. This semantics is the one most commonly sought by research efforts in replicated file systems (e.g., [4]). 1USR is distinguished from true one-copy serializability (ISR) in that, because of caching, it is possible for 1USR clients to make irreconcilable updates to a file without any knowledge that they are doing so. For example, two clients may both strictly read a file and then make conflicting updates into their caches. ISR is not attainable using our design (or most others); Echo [18] is an experiment in providing ISR via a file system.

Division of _read_ into _loose read_ and _strict read_ ensures that the entire cost of establishing 1USR consistency is charged to the process (reader) that demands the consistent value. This approach is in contrast to the other systems we know of, which take the “immediate write-back” (IWB) approach to consistency and availability. That is, when a file is either written or closed (depending on the system), the new value is copied to one or more servers before the client’s synchronous call returns. The eager approach of IWB seeks to support sharing. Unfortunately, IWB incurs the cost of synchronous global communication on every write/dose regardless of when the next read happens. Since in most cases of sequential sharing the read occurs quite some time after the write, the IWB approach imposes extra overhead on a workload that includes little sharing.

### 2.3 Currency Tokens

The most important construct in our system is the _currency token_, or CT. Currency tokens have the powerful effect of allowing strict reads to be implemented as efficiently as loose reads in most cases. A client that holds a CT for a file is guaranteed to find the current version by performing the same sequence of actions as for a loose read. This sequence is:

1. Check the client’s cache.
2. If no copy is found, check the primary server.
3. If the primary does not have a copy, then check the secondaries.

The first copy found is guaranteed to be current.

The definition of CTs depends on the notion of a _potential consistent writer_ (PCW). A PCW is a client site that has performed a strict read of a file to which the reading process has write permission. In other words, a PCW is a client that demonstrated both the desire (strict read) and the ability (write permission) to make a consistent update. A client receives a CT along with a requested file in response to a strict read if it is the only PCW reading the file, or if there are no PCWs for the file.

When a strict read is performed by a client that does not have a CT, every server is contacted, and each returns its copy of the file. (A simple optimization would be for a secondary to return its replica only if it is more recent than the version at the primary.) If the client is a PCW, this fact is recorded by each server. Furthermore, each PCW holding the file must be contacted to ensure that no cached consistent updates are overlooked. Files are tagged with _version vectors_ [5] so that the copies at different replicas can be compared for recency. The primary compares the version vectors and gives the client the most recent copy of the file plus an indication of whether it has a CT. Record of a CT is kept in (volatile) memory at the client and the primary, and nowhere else.

In order to reduce the number of these global operations, CTs should apply to sets of files rather than individual files. Furthermore, these sets should be reasonably large, since obtaining a CT is an expensive and non-fault-tolerant operation. We expand the jurisdiction of a CT by letting it cover the directory in which a file is located if there are no other PCWs for any files in that directory. (Note that only the current directory is covered; CTs never apply to nested subdirectories.) This substantially increases the number of files covered by a CT without significantly increasing the time required to grant the token, since information about PCWs is replicated at all servers.

For example, suppose user _elmer_ does a strict read on the file _/u/elmer/paper/intro.mss_. If there are no PCWs other than _elmer_ for any files in this directory, then we issue a CT covering all the files in _/u/elmer/paper_. This CT does not apply to any files in subdirectories below that point, however. By enlarging the domain of the CT in this way, we allow _elmer_ to read and modify many files with strict consistency and low latency (after the initial strict read).

It is possible for several clients to hold a CT on the same directory simultaneously. Consider an example: one user does a strict read on _/usr/bin/cc_, and a different user then strictly reads _/usr/bin/grep_. Because files in _/usr/bin_ are typically written infrequently by a specially privileged user, there will normally be no PCWs for any files in that directory. Hence, both of our clients will be given CTs that apply to all of _/usr/bin_.

### 2.4 Revocation of CTs

Currency tokens are revoked when one of the following occurs:

---

\(^5\)See Section 3 for empirical support for this contention.
1. One or more clients have strictly read a file to which they do not have write access, and the first PCW for that file arrives. In this case, existing tokens are revoked, and the unique PCW is given the only CT.

2. Exactly one PCW has the file and a second PCW strictly reads the same file. Neither client is allowed to hold a CT in this case.

3. The primary is unable to contact the client for an extended period. In this case, the primary discards its record of the client’s CTs. This rule protects the service from being hamstrung by the disappearance of a client.

When a CT applies to a directory rather than a single file, it must be revoked when any files in that directory are strictly read by a new PCW. The affected client may still attempt to gain CTs that cover individual files within the directory.

A client should not give up the CT when a file is closed, or even when the file is flushed from the cache. A CT makes this guarantee to a client: “either there are no PCWs, or all potential updates are localized to you and your primary-secondary server organization.” There is no need to relinquish this advantage unless the premises on which it is based are invalidated.

On every read the client indicates to the primary whether it thinks it has a CT covering the requested file. If in its reply the primary disagrees because the token has been revoked for one of the reasons given above, then the client drops its CT. CT information is stored only at the client and its primary. However, the list of PCWs for a file must be replicated at each server storing a copy of that file. This requirement is necessary to continue operations during crashes and partitions.

Note that a currency token is not analogous to a “write token” used by other systems [18, 1]. Instead, it is a hint [19] that substantially improves performance in an environment with little sharing.

2.5 Further Details

2.5.1 Interference Between Strict and Loose Reads

It is the responsibility of clients to use strict reads when a file is expected to be shared. However, a server can optionally retain memory of all clients that have read files from it, whether these reads were performed strictly or loosely. In this case, the service can detect potential interference between loose_read and strict_read.

2.5.2 Optimization for Write-Read Sharing

By employing a “short-circuit” optimization, it is easy to ensure that both strict and loose reads will return the latest value during simultaneous write-read sharing when there is only one writer. In addition, both operations will be very efficient because only the client and the potential writer are involved—no servers need be contacted after the initial read.

Because we require that the list of PCWs be replicated at all servers, a client performing even a loose read will learn of PCWs when it contacts its primary server. If there is only one pre-existing PCW at the time of the read operation, and the reader is not itself a PCW, then the reader can request the file from the unique PCW. If the PCW still holds a CT on that file, and the file is in its cache, it delivers its value (guaranteed to be the latest) to the reader. Thereafter, strict read can be performed by direct client-to-client communication, reverting to the usual method only when the PCW loses its CT.

2.5.3 Other Operations

Create and Delete. Programs such as compilers and text formatters often create intermediate files as they run. Since these temporary files are usually short-lived, there is no reason to write them to servers. Therefore, newly created files at the client are not accessible to other clients by any form of read until they are propagated to servers—if they live long enough to be picked up. When long-lived files are deleted, the server propagates record of the deletion in the same manner as any other update.

Control Operations. Because of the need to keep track of PCWs, control operations that change the protection attributes of a file must be done in the manner of an initial strict read: all copies must be updated. However, by enforcing strictness on updates, operations such as stat that get file attributes are quite efficient. We can read these attributes from any available copy of a file.

2.6 Fault Tolerance

2.6.1 Conditions for Blocking

Fault tolerance is a concern with only two of the three main operations; because of its weak semantics, loose_read is oblivious to anything less than total failure.

The write call can block, but will do so only if the client’s cache fills because purge notices have not been received for the updates in the cache. Unusually heavy update activity may cause this situation to occur between pickups, and a forced write will help to free cache space in this case. However, the lack of purge notices may also be due to failures: the updates cannot be replicated on a sufficient number of secondaries to guarantee N-resilience, so the client must retain the updates until it is safe to purge them.

A client holding CTs is prevented from performing strict reads only if all of the following are true:

1. The desired file is not in the local cache.
2. The client’s primary is unreachable or does not have the file.
3. A total of N or more sites are unreachable, where N denotes the number of secondaries to which updates must be propagated before purge notices are issued.
In this case, the service cannot provide the client with a copy of the file that is guaranteed to be up-to-date, and so \textit{strict read} will block.

A client trying to obtain a CT via a strict read is blocked if any server is unreachable, or if any PCW for the requested file is inaccessible. Therefore, it is important that a CT apply to multiple files and that the CT be retained by the client for as long as possible. The fewer the number of CTs to be obtained, the fewer the opportunities for blocking.

\subsection{2.6.2 Reaction to Failure}

We consider two cases: loss of a primary and loss of a secondary. (Loss may mean either site crash or network partition.) Up to the point at which purge notices cannot be issued due to insufficient propagation of replicas, loss of a secondary has no effect. The loss will not be noticed until the secondary returns and is reintegrated with the other servers.

Failure of a primary is the difficult case. It is detected in two ways. First, a client may find that its read requests go unanswered. At this point, it must block and wait for a new primary to be provided. If a sufficient number of servers are up and in communication, a new primary will be generated by the service thanks to the second way in which loss of a primary becomes evident: namely, when the secondaries fail to receive their periodic updates and are unable to cause a retransmission.

Once a secondary has decided the primary is gone, it starts an election [12] to choose a new primary. The only requirement on the election is that few than \( N \) servers are unreachable, where \( N \) is the number of replicas that must be propagated to secondaries before a purge notice can be issued. This guarantees that the most recent version of any file is either (1) in one client's cache pending purge notification, or (2) stored on at least one of the \( N \) or more surviving servers. In other words, if fewer than \( N \) servers have been lost, they have not taken all the copies of the most recent version of some file with them.

Assuming that the requisite number of servers are available, any one of them can be elected as the new primary. For a system servicing mobile clients, however, we believe it is advisable to choose the server that is nearest to the client, as measured by network hops.

After being elected, the new primary attempts to revalidate the CTs held by its clients. It does this by searching out the most recent versions of the files covered by those CTs and copying them to itself. Version vector comparison is used to determine which replica is the most recent; if any conflicts are detected, they are reported to the appropriate client. Note that other clients are never involved in this process; by definition, no client serviced by this primary could hold a CT on a file if some other client were a PCW for that file. Once the new primary is brought up to date, it immediately makes a pickup from all of its clients so that they will learn of their new primary. Both strict and loose reads can now be performed as usual, with CTs covering the same domains as they did before the election.

So in summary, propagating updates to \( N \) secondaries yields \( N \)-resiliency and still lets a client retain its CTs and perform strict reads.

\section{3 Evaluation}

Our evaluation consists of experiments to measure the constraints imposed by operating circumstances, and critical comparison with other work.

\subsection{3.1 Experiments}

Our motivation is to reduce or eliminate synchronous global communication from the typical case of common file service operations. Our idea of lazy propagation of updates and the corresponding "dual read call" interface leads to three hypotheses.

First, we hypothesize that few applications need what is provably the most recent version of a file. If few applications require \textit{strict read}, then most reads and all writes will be lazy and hence fast and scalable. The second hypothesis is that it is not hard for application programmers to know they need the latest value and hence must use the \textit{strict read} call. A third hypothesis is that lazy update can take place quickly enough so that those applications that perform a loose read will receive the most recent value anyway.

We have performed experiments to test our hypotheses:

- \textbf{Experiment 1:} Measure use of existing file systems to determine the distribution of time intervals between write and subsequent read.

This information tells how fast update propagation should be. It also obliquely hints at the extent to which strict read is needed.

- \textbf{Experiment 2:} Determine what fraction of applications can be claimed to "obviously" require strict read.

Evaluating the \textit{loose read/strict read} interface is an ease of use question, best answered by a prototype file system. In absence of a prototype, this experiment gives some information about how difficult to use the dual call interface might be.

- \textbf{Experiment 3:} Determine client update activity.

This information helps describe what size the cache should be and what pickup interval is appropriate.

\subsection{3.2 Results}

Our results were obtained by examining file traces gathered at our institution. Traces were gathered on a Sun 3/80 running SunOS 4.0.3c. This version of SunOS offers a “C2 secure computing facility” that includes the ability to produce a system call audit trail. Using this feature, we gathered three large traces of a single user’s file access activity. The traces report the file access activity of a volunteer user performing his normal work.
activity over a period of two weeks. The first trace contains 10,182 events captured over 33 hours. The
second trace contains 12,472 events captured over 72 hours, while the numbers for the third trace are 25,440
and 86, respectively. During these hours activity varied widely and included compilations, document
production, data analysis and display, large searches for certain files (i.e., UNIX “find” commands), USENET
news reading, printing, and other operations.

3.2.1 Experiment 1: Write-Read Separation

When files are write-shared—or written by one client
and read by others—it is important to know how closely
a file access follows an update to that file. If updates
to a shared file are widely spaced, a client will get the
most recent version of a file even with a loose read. We
analyzed our traces to determine the distribution of time
between an update and the next open of the same file.
Most files had long intervals between update and open,
as indicated in the first three columns of Table 1.

It would be enlightening to know what fraction of
writes and reads come from the same site. Unfortunately,
we cannot use our data for such an experiment
because it was garnered from the activity of a single
user. Thanks to currency tokens, update propagation
would in many cases not be required to catch write-
read dependencies between applications at the same
site. Therefore, the figures in the table overestimate
the need for timely update propagation.

3.2.2 Experiment 2: Choosing Between Strict and Loose Read

The fourth column of Table 1 shows the percent-
age of those accesses made by applications that “ob-
viously require” 1USR. This estimate was made by us
based on our limited knowledge of a handful of applica-
tions appearing in the traces, and therefore constitutes
an underestimate of the ease with which the need for
strict_read can be judged. Files we counted as obvi-
ously being used in a serial fashion included the inter-
mediate files of a document processor and object files
produced by the C compiler and passed onto the linker.
While this eyeball technique is hardly scientific, it mim-
ics the decision programmers would have to make.

The conclusion is that files with short update-open
intervals were used by programs that could easily know
that serializability is required. In other words, files ei-
er had “safe” update-open intervals, or were being
used by programs that understood the serial nature of
what they were doing.

3.2.3 Experiment 3: Update Activity and File Lifetimes

Update Activity. Due to the limitations of our trace
data, it was difficult to estimate the amount of update
activity performed by a user. Ousterhout’s 1985 study
[7] found that each user read or wrote an average of
300 to 600 bytes of file data per second. This study
was conducted on VAX/11-780s and is now outdated.
Our own experiments, conducted on a Sun-3, suggest
a higher figure—perhaps 1000 bytes per second. Both
figures represent the total volume of reads and writes
together, and therefore constitute a very conservative
upper bound on the amount of update activity.

A rule of thumb is that reads outnumber writes by
2:1, so update activity can be estimated as perhaps 350
bytes/second. But even 1000 bytes/second of updates
would be acceptable: this amounts to 60KB/minute,
which suggests that even the small caches of portable
workstations would be capable of storing several min-
utes of recent updates. Furthermore, a file may be up-
dated several times between pickups, but only a single
version need be maintained in the cache. This further
reduces the actual amount of client cache space required
to store updates between pickups.

File Lifetimes. Several studies have already been per-
formed on file lifetimes [7, 11, 17], and our own data
confirms one of the earlier results: if a file is short-lived,
it is usually very short-lived. The majority of files that
were both created and destroyed during our traces have
a lifetime of less than five minutes. Furthermore, the
great majority of these files live for less than three sec-

<table>
<thead>
<tr>
<th>Trace</th>
<th>Interval</th>
<th>Pct Accesses</th>
<th>Pct Known 1USR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0-3</td>
<td>1.4</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td>0-10</td>
<td>3.5</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td>0-60</td>
<td>6.8</td>
<td>98.5</td>
</tr>
<tr>
<td></td>
<td>0-300</td>
<td>9.0</td>
<td>96.3</td>
</tr>
<tr>
<td></td>
<td>0-600</td>
<td>11.4</td>
<td>94.3</td>
</tr>
<tr>
<td></td>
<td>0-3600</td>
<td>17.4</td>
<td>87.6</td>
</tr>
<tr>
<td></td>
<td>0-∞</td>
<td>100.0</td>
<td>38.1</td>
</tr>
<tr>
<td>2</td>
<td>0-3</td>
<td>2.0</td>
<td>97.6</td>
</tr>
<tr>
<td></td>
<td>0-10</td>
<td>5.2</td>
<td>96.0</td>
</tr>
<tr>
<td></td>
<td>0-60</td>
<td>9.7</td>
<td>95.7</td>
</tr>
<tr>
<td></td>
<td>0-300</td>
<td>13.0</td>
<td>95.7</td>
</tr>
<tr>
<td></td>
<td>0-600</td>
<td>16.5</td>
<td>95.6</td>
</tr>
<tr>
<td></td>
<td>0-3600</td>
<td>23.9</td>
<td>95.1</td>
</tr>
<tr>
<td></td>
<td>0-∞</td>
<td>100.0</td>
<td>47.1</td>
</tr>
<tr>
<td>3</td>
<td>0-3</td>
<td>2.0</td>
<td>98.2</td>
</tr>
<tr>
<td></td>
<td>0-10</td>
<td>5.6</td>
<td>96.4</td>
</tr>
<tr>
<td></td>
<td>0-60</td>
<td>10.8</td>
<td>90.1</td>
</tr>
<tr>
<td></td>
<td>0-300</td>
<td>14.3</td>
<td>85.8</td>
</tr>
<tr>
<td></td>
<td>0-600</td>
<td>17.9</td>
<td>82.6</td>
</tr>
<tr>
<td></td>
<td>0-3600</td>
<td>26.0</td>
<td>77.2</td>
</tr>
<tr>
<td></td>
<td>0-∞</td>
<td>100.0</td>
<td>39.5</td>
</tr>
</tbody>
</table>

Table 1: Update-Open Intervals

The interval is measured in seconds. The Pct
accesses column indicates what percentage of
the write-read dependencies occurred during
the given time interval. The numbers in the
"known 1USR" column indicate what percentage
of dependencies during each time period could
easily be seen to require strict_read.
This table lists the number of files that were both created and deleted within the trace. In all three traces, more than 75% of those files lived 3 seconds or less.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Files</th>
<th>Pct 0-3</th>
<th>Pct 3-300</th>
<th>Pct &gt;300</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>67</td>
<td>88.1</td>
<td>10.4</td>
<td>1.5</td>
</tr>
<tr>
<td>2</td>
<td>39</td>
<td>87.2</td>
<td>5.1</td>
<td>7.7</td>
</tr>
<tr>
<td>3</td>
<td>53</td>
<td>75.8</td>
<td>16.4</td>
<td>7.8</td>
</tr>
</tbody>
</table>

Table 2: File Lifetimes

This imposes additional overhead, but prevents divergent writes to a file. In addition, Deceit supports automatic file migration to a nearby server to improve performance. There is very little service disruption in Deceit, although a high “write safety level” requires the client to wait while multiple copies of a file are written. The default is to wait for just one copy to be written. Deceit’s write tokens could cause scalability problems, but assuming a low degree of sharing—and a consequent lack of contention for tokens—this system should also scale well.

Echo takes the position that all writes should be visible immediately. This necessitates the use of both write and read tokens. Since sharing is infrequent, this is usually not a problem, but during periods of sharing, Echo’s performance suffers for its clean semantics; tokens must be shuttled back and forth between all clients who are accessing the file. Echo’s clients must wait for writes to propagate—another cost of the nice semantics.

Our lazy tree method can theoretically provide better write performance than any of the above systems, in the typical case. Ordinarily, a user’s updates are made only to the local cache, and are picked up asynchronously by that user’s primary server. In general, the lazy tree’s clients wait only during open, since write and close operations are done locally and picked up later. In the worst case, a read operation will also require the client to wait. The lazy tree scheme has the potential to scale very well. Because many files live for only a short time, they will never even be picked up by the primary server, and this will reduce network traffic.

### 3.3.2 Resiliency

With Coda, a client may both read and write so long as any replica is available. Coda takes the position that conflicts are sufficiently rare so that detection is preferable to prevention.

In Deceit, reads are always allowed. Writes may or may not be possible, depending on the setting of a parameter that allows generation of a new write token when the old one is unavailable.

In Echo, the goal of one-copy serializability prevents operation in a minority partition. Even a read operation requires that a majority of the replicas be up and in communication. Otherwise, a user might read stale data, which violates one-copy serializability.

Our design lies somewhere between Coda and Deceit. Of course loose_read is always possible when any replica is accessible, as in Coda. Writes are blocked only when the client’s cache is full. The cache will fill up only if purge notices are not forthcoming. And purge notices will not be delivered if failures prevent an update from being sufficiently widely propagated. However, it is important to note that once service is re-established after a failure, the primary server can re-read any insufficiently propagated value from the client’s cache. The client need not even be aware that any failure and recovery algorithms have been executed by the service.

The fault tolerance of strict_read is dependent on
usage patterns. In the worst case, the operation will block during any failure. But since CTs will usually apply to entire directories rather than single files, the typical case is very far from the worst case.

4 Summary

We have presented a replica management algorithm that extends lazy semantics to its logical extreme, introducing both lazy read and lazy write operations. Lazy operation decreases overhead and thereby increases scalability. To make the resulting weak semantics palatable to applications that require consistency guarantees, we have proposed a different file system interface in which applications must explicitly declare that they need what is provably the latest value.

To discover the latest value in a lazy-update environment requires global communication, and so the naive implementation of strict read is both slow and not fault-tolerant. To address this problem, we have invented the notion of a currency token. A client holding a currency token need contact few (if any) servers to perform strict reads. By performing strict read only when necessary and by holding a currency token over multiple files for as long as possible, an application can greatly lessen its chance of blocking.

5 Acknowledgements

This work was supported in part by National Science Foundation grant CDA-9022123, IBM Corporation, the New York State Science and Technology Foundation’s Center for Advanced Technology in Computer and Information Systems, and the AT&T Foundation.

References