An Efficient Variable-Consistency Replicated File Service

Carl D. Tait and Dan Duchamp

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

ABSTRACT

In a previous paper, we introduced a new method of file replica management designed to address the problems faced by mobile clients. Two goals shaped our design: minimizing synchronous operations, and letting clients determine the level of consistency desired. The previous paper was a high-level view of the most fundamental issues; here, we refine our previous ideas and flesh out the rest of the design.

1 Introduction

This work investigates how to design a replicated file system that will serve mobile clients well. The idea that file service clients might be mobile is a natural one, given the likely marriage between two exploding trends: portable computers and wireless networks.

In a previous paper [14] we have argued that client mobility is a major new development that requires re-thinking file system design, and that existing approaches to replica management (e.g., [7, 13]) would not cope well with mobile clients. In that paper we proposed an alternative: a lazy "server-based" update scheme and a new service interface that allows applications to select strong or weak consistency semantics on each particular read call. Our earlier work addressed only some of the aspects we see in the complete problem, and developed our solution in only a certain amount of depth. This paper finishes our design.

We make several assumptions about operating conditions:

1. Client movements cannot be constrained, although patterns of movement may exist.
2. Latency of remote operations degrades as the distance between hosts increases.
3. The presented load is what Ousterhout has called "engineering/office applications" [10]. In this model workload, sequential sharing is not uncommon, but simultaneous sharing (other than read-read) is rare.
4. A file cache of modest size is maintained by each client.

5. File service sites can synchronize their clocks. This assumption, once controversial, can now be satisfied by several clock synchronization protocols (e.g., [8]). We do not make any assumptions about client clocks.

Flowing from these assumptions is our conclusion that three design goals are of paramount importance in handling mobile clients: we must minimize synchronous multi-server operations, ease the addition and deletion of server sites, and allow for incomplete replication at servers. We discuss our reasoning for each in turn.

Minimize synchronous multi-server operations. Our first two assumptions lead us to the conclusion that file systems that frequently use "global communication"[1] will not perform well if clients move over a wide area while they remain mapped to a fixed set of servers. The reason is that the latency of contacting the most distant server is a lower bound on the latency of the entire multi-server operation. File service operations that involve global communication will slow down when a client moves away from its current set of servers.

Ease the addition and deletion of server sites. In order to avoid global communication, two features seem desirable. First, a primary-secondary server organization, in which the client communicates only with the primary and the primary communicates asynchronously with the other servers. Second, ensuring that the primary is always located near the client. Since the client moves unpredictably, the service must be ready to regularly add a new replication site, *typically as the new primary.* The model of mobile operation we envision is that a client moves over a wide area, perhaps to areas it has never visited before. Once in a new milieu, it will negotiate with some local machine to become its new primary. There is also presumably a need for regular replica deletion, in order to limit the degree of replication to a sensible number in case of a highly mobile client. Regular addition and deletion of replicas, especially the primary, marks our design as unusual.

Allow for incomplete replication. If identities of the servers — specifically, the primary — are changing frequently and unpredictably, replication sites should not be expected to always store complete copies of the client's file set. We provide for incomplete replication, in which a client's files need be replicated at only a subset of the server sites.

Our previous paper addressed only the first of these concerns, and did so in limited detail. In this paper, we address the goals of replica site addition/deletion and incomplete replication, while refining and expanding our previous ideas. In particular, we describe how our design manages the changing relationships between clients, servers, and files, and provide a careful analysis of failure cases.

2 Operation in the Absence of Failures

As suggested in the introduction, our design reduces global communication by using a primary-secondary server hierarchy. The client communicates synchronously with the primary only. We employ write-back caching with the primary, not the client, choosing when updates are copied from the client's cache. The primary makes periodic *pickups* from the clients it is servicing, and propagates updates back to the secondaries asynchronously. This allows the client's `write()` operation to return immediately after placing the new value

[1] We define a global communication as an operation in which several servers must be contacted synchronously. Coda's `close()` operation is an example in this category.

in its cache.

Once some number, N, of secondaries have acknowledged receipt of an update, the primary informs the client that the associated cached update can be discarded. This notification — which might be piggybacked onto the next pickup request — is called a *purge notice*. Because a client must retain an update in its cache until receiving a purge notice, the service has some latitude (constrained by the size of the client's cache) to "wait out" secondary server failures without any disruption in accepting `write()` calls.

During periods of heavy update activity, a client's cache may fill between primary server pickups. For this case, we provide a synchronous, blocking *forced write* operation that causes an immediate pickup.

Asynchronous update propagation has well-known benefits and dangers, and it has been used in the design of other replicated file systems (e.g., [13]). Our work is significantly different in two ways. First, by requiring that the client retain an update until a sufficient number of secondaries have the new value, we trade cache space for a low-latency, yet reliable write operation. Other systems typically block while some number of replicas are written (the remainder being written asynchronously), or return immediately with no guarantee that the update will be safely propagated. We push back the blocking point so that the client typically is not involved in the propagation process.

The second feature that sets our work apart is how we couple asynchronous update propagation with a read interface that allows an application to choose either "lazy" or "UNIX-like" semantics on a per-read basis, as described below. When there is little sequential sharing, this design achieves high fault tolerance and low latency for both reads and writes. Experimental data presented in our earlier paper supports the feasibility and potential value of this dual-read-call interface.

Given such a design, the key issues are:

- How does this interface work?
- When, why, and how do clients and primaries bind to each other?
- How are files replicated on secondaries?

We address these issues in the next three subsections.

2.1 Dual-Read-Call Interface

We split the traditional `read()` interface into `strict_read()` and `loose_read()`. There is no guarantee concerning the value returned by the loose form, whose implementation is shown in Figure 1. In principle, the strict form returns the "most recent consistent" value; this will be defined more precisely after we have explained the operation of `strict_read()`. If `strict_read()` and `write()` are used exclusively, the system provides a replicated analogue of "one-copy UNIX semantics" (1USR): the semantics found in a centralized UNIX system. 1USR is not equivalent to one-copy serializability (1SR): due to caching, 1USR clients can make conflicting updates to a file without knowing that they are doing so. For compatibility, a generic `read()` library call could simply call `strict_read()`. For the convenience of some applications, another library routine could be defined with

```
Look in the client's cache

If there is no copy in the cache, then check the primary server

If there is no copy at the primary, then check any of the secondaries,
   in any order
```

Figure 1: Loose Read Algorithm

the following semantics: strictly read the requested file if possible; otherwise, return any available copy.

While `loose_read()` is allowed to return any convenient value, analysis of file-trace data presented in our previous paper indicates that a "best effort" implementation will almost always return the most up-to-date value. This conclusion is supported by the recent study by Baker et al. [1], which found that only one third of one percent of `open()` calls read data that was written by another client less than 60 seconds previously. This study concluded that (automatic) cache consistency is desirable to prevent clients from using stale data; in our system, clients use the `strict_read()` operation to enforce cache consistency.

2.1.1 Currency Tokens

A naive implementation of `strict_read()` would contact all servers and all clients that had read the file and retrieve the most up-to-date copy it finds. *Currency tokens* (CTs) are used to avoid this naive approach.

CTs rely on the idea of a *potential consistent writer*, or PCW. A PCW is a client site with a process that has strictly read a file, and that has write permission for that file. In other words, a client is declared a PCW after demonstrating both desire (strict read) and ability (write permission) to make an update in a consistent fashion. A CT is given in response to a `strict_read()` if there are no PCWs for that file, or if the client is the only PCW. A client can receive a CT only from a strict read, never from a loose read. Holding a CT allows the client to know that either there are no PCWs for the file in question, or that all potential updates are localized to itself and its primary-secondary hierarchy.

A client that performs a strict read without a CT initiates a relatively complex series of actions. First, recall that an update must be replicated on at least N secondaries before a client is allowed to purge the update from its cache. Therefore, assuming that there are a total of T secondaries, at least T-N+1 secondaries must be contacted — as well as any PCWs that have the file in their caches — to ensure that the most recent value is located. This condition is simply that of quorum consensus, wherein the read and write quorums must overlap [4].

Files are tagged with timestamps so that the copies at different replicas can be compared for recency. After reading T-N+1 copies, the primary compares the timestamps and gives the client the most recent copy of the file plus an indication of whether it has a CT. To ensure consistency, lists of PCWs and clients who hold CTs must be maintained in non-

```
Client sends strict_read() to primary

Phase 1:
   Primary multicasts strict_read() to all secondaries

   Each secondary evaluates whether client is a PCW, and,
      if so, records the fact in non-volatile storage

   Each secondary returns a timestamped file to the primary, along with
      timestamped lists of PCWs and CTs for this file (if there are any)

Phase 2:
   Block until at least max{T-N+1, majority} secondaries reply

   Using the most up-to-date PCW list it has received,
      the primary determines that client should be given a CT
      iff there are no PCWs, or client is the only PCW

   If a new CT is being granted, this fact is recorded in
      non-volatile storage on at least a majority of secondaries

   If old CTs must be revoked due to the arrival of a PCW,
      the primary performs the revocation

Phase 3:
   Primary reads cached copies from all PCWs

   Primary returns most up-to-date replica gathered from
      secondaries and PCWs, along with a CT if appropriate
```

Figure 2: Strict Read Without a CT

volatile storage on at least a majority of secondaries so that this information will be found during any initial `strict_read()`. The complete algorithm is sketched in Figure 2.

Subsequent strict reads with a currency token are considerably simpler and faster. The algorithm for this case is shown in Figure 3; note the similarity to the loose read algorithm in Figure 1. CTs have the effect of allowing most strict reads to be implemented by executing almost the same sequence of actions that is performed for a loose read. The only difference is at the level of secondaries: if the search gets that far, a strict reader with a CT must contact enough secondaries to ensure that the most recent copy is located. The initial strict read is the only point in our design at which a synchronous operation is required: several secondaries are contacted, as well as all PCWs.

PCW and CT lists are updated at secondaries by a simple process that can do only one update at a time. It is necessary to wait in line to have a PCW or CT added to (or

```
Look in the client's cache

If there is no copy in the cache, then check the primary server

If there is no copy at the primary, then check at least T-N+1
   secondaries and return the most recent value found
```

Figure 3: Strict Read With a CT

OPERATION	QUORUM REQUIREMENT
Initial strict read	read file at T-N+1 secondaries read/write PCW and CT lists at majority of secondaries
Strict read with CT	read file at T-N+1 secondaries if secondaries read at all

Table 1: Quorum Conditions for Strict Read

deleted from) the list. Furthermore, since we allow initial strict reads from different clients to execute simultaneously, it is necessary that read quorums intersect so that at least one client will learn about the other. Thus, initial strict reads must actually contact T-N+1 secondaries or a *majority* of secondaries, whichever is larger. A complete statement of the quorum conditions is given in Table 1.

Correctness of Simultaneous Operations. These mechanisms, combined with the fixed sequence of operations involved in gaining a CT, implicitly serialize initial strict reads. Overlapping strict reads of the same file are therefore assured of functional correctness. To see why this is true, assume that two clients start an initial strict read at the same time:

1. Two non-PCWs: No problem, of course.
2. Two PCWs: Because read quorums overlap, at least one client is guaranteed to learn about the other's existence. Whichever client finds out revokes all CTs.
3. A non-PCW and a PCW: The same reasoning as (2), with one additional tricky case. The non-PCW may update the CT list *after* the PCW has read it. Thus, the PCW must ensure that the CT list has not changed since it was read (using timestamp comparison). If the list has been modified, it is re-read and re-analyzed. This is repeated until the list stabilizes.

   (Proof that the CT list will stabilize quickly: (1) If no new strict readers arrive, the list stabilizes immediately. (2) If a new non-PCW arrives, it notices the PCW in the PCW list. This prevents a CT from being granted to the new non-PCW, so the CT list is not modified. (3) If another PCW shows up, all CTs are revoked and the list stabilizes to an empty list.)

A small complication arises because we allow *incomplete replication*; that is, there is no requirement that a file be replicated at every secondary. It is possible that some secondaries will not have the file at all, much less the most recent version. Nonetheless, by using the scheme described above, we ensure that an initial strict read will find the most recent replica, assuming that a sufficient number of secondaries are available. A secondary that does not have a replica of the file in question acts as a kind of *witness* [11]: a server that may be counted as part of a quorum even though it contains no explicit information about a file.

Reducing Initial Strict Reads. In order to reduce the number of initial strict reads, which require global communication, CTs should apply to groups of files rather than individual files. We expand the scope of a CT by letting it cover a file's entire directory if there are no other PCWs for any files in that directory. (We are assuming the existence of a standard hierarchical name space.) In addition, a file that has several names — through symbolic links, for example — is covered by a single CT; it is not necessary to gain a CT for each name.

We have rejected the seemingly attractive notion of expanding a CT beyond the confines of a single directory. There are two major problems with doing so. First, expansion can be very time-consuming: in the general case, the number of files covered increases exponentially as we move up or down in the name space. Second, after spending time in expansion, a single additional PCW can topple the carefully constructed CT edifice. On the other hand, single-directory expansion has several advantages:

- Expanding CTs is fast and easy.
- We don't lose much if another PCW shows up.
- Directories generally contain enough files to make this approach dramatically superior to single-file CTs.

Currency Token Revocation. CTs can be revoked for several reasons. The guiding principle is that the presence of even one PCW for a file requires all other clients to give up their CTs on that file. However, we provide a "short-circuit" optimization for the single-writer, multi-reader case: reads are directed to the unique PCW. If there are two or more PCWs, no clients can hold CTs, and the short-circuit scheme is ruled out.

What happens if the service needs to contact a currently unreachable client, either to revoke a CT or to read a file from a PCW's cache? A straightforward solution involves timeouts: a client who cannot be contacted for a certain length of time is deemed to have become permanently disconnected. Record of the client's CTs and PCW status is discarded, and other users' strict reads are allowed to proceed. But this approach cannot completely close the window of error: a client who has been disconnected might suddenly reappear with a cache full of updates that were missed by strict readers in the interim.

Although we cannot eliminate errors entirely, we can at least eliminate the inelegance and delay of timeouts; we have concluded that demand-based disconnection is a convenient simplification. Whenever the service needs to contact a client, but the client is unreachable for some reason, we give up immediately, automatically revoking any CTs that the wayward client is holding. This applies with equal force whether the system is making server-based pickups, attempting to contact PCWs, or explicitly revoking CTs.

It is important to realize that CTs are not equivalent to read and write tokens used by other systems. They are simply *hints* used to improve the performance of strict reads. In other systems, tokens are prerequisites for performing operations; in our design, strict reads can take place without using currency tokens, though performance will be less than optimal. CTs are somewhat akin to *callbacks* in Coda [12], though the pessimistic nature of CTs provides a stronger consistency guarantee: a CT will not be granted if any other client has even the *potential* to make a consistent update.

2.2 Client-Primary Attachment

Client-to-primary assignments are made by a special module called the *matchmaker*, which is most conveniently implemented as a process on any service machine. When a client wishes to obtain a primary server, it sends a request to any convenient matchmaker, which then selects a primary based on criteria of its choice. For coping with mobile clients, the criterion might be physical proximity.

After making its choice, the matchmaker sends the client's address to the selected primary and forgets about the transaction. The new primary server then performs a pickup so that the client will learn the identity of its new primary. The client saves the primary's address in (volatile) storage for use in subsequent read operations.

A newly-selected primary always starts with an empty cache for the client it is servicing. This is necessary to prevent consistency problems that might occur due to currency token expansion. For example, if a client strictly reads `/usr/local/bin/emacs`, the client will typically be given a CT that covers all of `/usr/local/bin`. The primary, however, is only guaranteed to have the most recent version of `emacs`, not every file in the directory. Therefore, we must ensure that a down-level version of (for example) `gprof` is not returned during a strict read just because the primary happens to have a copy lying around.

The only non-volatile state retained by the matchmaker is a list of servers that can act as primaries. It is not necessary to modify this list when a potential primary crashes. The matchmaker can "ping" the machine it has chosen for a given client, and if there is no response, the matchmaker will simply choose a different server. Modifying the list of primaries is trivial. Adding a new server poses no problems, and deletion is not difficult. If a client attempts to use a primary that has been deleted from the list, either there will be no response, or the message will be rejected by the disgruntled ex-primary. In either case, the client need only apply to the matchmaker to obtain a new primary.

Given the relative infrequency of calls to the matchmaker, we do not anticipate any scalability problems. For availability, however, it is desirable to run the matchmaker on several different machines. Since the matchmaker is stateless except for the list of potential primaries, multiple executions on different machines can operate independently if necessary.

A client may ask the matchmaker for a new primary at any time, for any reason. Because the client does not discard cached updates until receiving confirmation that they have been stored on the required number of secondaries, switching among primaries in any arbitrary way is no threat to correctness. Similarly, a primary may safely pull out of a client-primary relationship at any time. In both cases, the client will have to obtain a new primary and pickups will resume normally.

Good reasons for requesting a new primary include failure of the old primary and movement (leading to reduced performance) by the client. Letting the client drive the selection process substantially simplifies our design, especially in the face of failures (see Section 3.2 for details). There is a cost for changing primaries: the matchmaking protocol must be run, CTs must be revalidated, and updates must be picked up once again by the new primary, so the client should not be capricious about requesting a new primary.

When a switch is made, both the new primary and the client will learn about it: the new primary's first job is to inform the client of its identity. In order to handle primary changeover gracefully without resorting to explicit disconnection messages, old primaries *figure out* that their services are no longer needed. This is straightforward because the client always knows the identity of its current primary. Whenever a putative primary goes to a client to make a pickup, it first verifies that it is still the primary server for that client. If the client is now being serviced by a new primary, the old one goes away and leaves the client alone. For example, suppose that client C has moved away from primary P1, and has called the matchmaker to obtain a new primary P2. During P1's next pickup, it learns that P2 is the new primary. P1 immediately stops making pickups from C.

It is tempting to blur the line between clients and primaries: why not run the primary server code on the client machine? This would reduce the number of messages required during an initial `strict_read()`. Unfortunately, clients are not trusted, so we cannot allow them to access secondaries directly. A malicious client could invalidate our consistency guarantees by ignoring our protocols and writing inconsistent values to secondaries.

2.3 Filesystem-Secondary Attachment

Primary servers are used principally as intermediaries between clients and secondaries. Primaries do not function as replication sites *per se*. Instead, we use groups of secondary servers to replicate file systems. For every file system, there is a corresponding group of secondary servers that handles the replication of files in that file system. Because we support incomplete replication, it is not necessary that every file be replicated on every server, however.

In practice, a server can simultaneously act as a primary and a secondary. A group of machines may be used both to replicate files and to act as primary servers for clients. Logically, however, it is necessary to maintain the functional distinction between primaries and secondaries.

A primary server learns about filesystem-to-secondary mappings by calling a *mapping server* specified by the client. Ordinarily, this `map()` function will be called only when a file system is mounted. This is a comparatively infrequent operation, so scalability should not be adversely affected.

We assume that secondary server mappings will rarely be changed, but we do provide a modification method based on Gray's idea of *leases* [5]. Mappings returned by `map()` are valid for only a limited period of time: relatively long (say, 10 to 30 minutes), but decidedly non-infinite. At the end of the lease period, `map()` must be called again to revalidate the mapping.

To modify a filesystem-to-secondary mapping, we must wait until all leases on the mapping expire. At that point, we can update the mapping and let `map()` start handing out the new information. In order to prevent starvation while waiting for leases to expire,

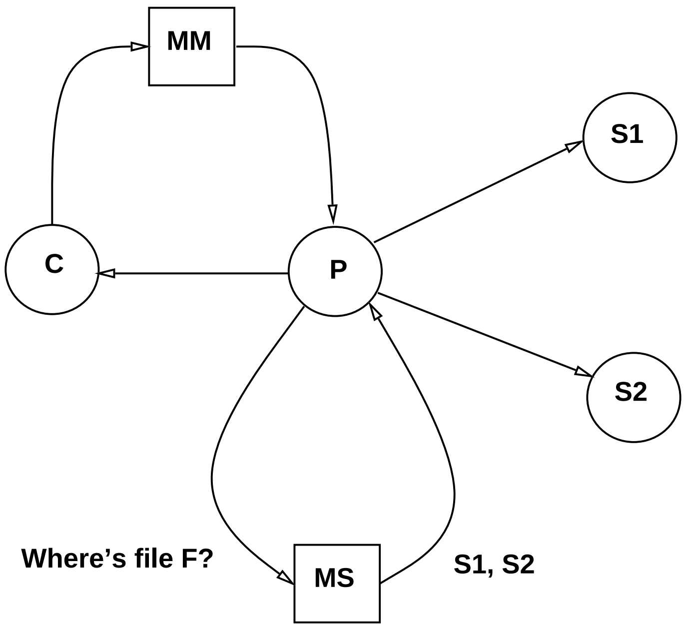

Figure 4: System Organization

the mapping server can hand out shorter leases after there has been a request to re-map. For example, if we know that all leases will expire in at most 30 minutes, we can still provide 29-minute leases on the same information.

The overall organization of our system is shown in Figure 4. There, C is the client, P is the primary, S1 and S2 are secondaries, MM is the matchmaker, and MS is the mapping server. Table 2 shows the state information that must be held by each component in the system, and whether the state needs to be held in non-volatile or volatile storage. In that table, the notation "File+timestamp cache?" refers to the fact that the client may store its updates (and their associated timestamp information) in either volatile or non-volatile storage; clients that store updates in non-volatile storage will not lose data in a crash.

2.4 Connection at a Distant Site

A desirable feature of our design is that a client can move to any location where network access is available and still continue to access the files at its "home site" in the same way that it would access them locally. For example, someone from New York who is visiting Berkeley would call the Berkeley matchmaker to obtain a local primary server. (We assume that some form of negotiation is made for the use of local resources.) The locally-chosen primary then calls the mapping server in New York to locate files that the client wishes to access. The client need remember only its own address and the address of the mapping server at its home site. These addresses need to be retained in non-volatile storage. Previous work on mobile internetworking by Ioannidis et al. [6] has demonstrated the feasibility of such a scheme from a networking perspective.

Under normal circumstances, we do not expect the primary server to play an im-

COMPONENT	NON-VOLATILE STORAGE	VOLATILE STORAGE
Client	Own address Mapping server address File+timestamp cache?	Current primary CTs held File+timestamp cache?
Primary	—	List of clients Filesystem-to-S mappings Files being propagated
Secondary	List of PCWs List of CTs	—
Matchmaker	List of primaries	—
Mapping server	Filesystem-to-S mappings	—

Table 2: State Held by Each Component

portant role as an intermediate cache. Muntz and Honeyman [9] report that intermediate caches experience a surprisingly low hit rate: less than 19% when the client itself has a 20 megabyte cache. Blaze and Alonso [2] achieve better results by using a dynamic, hierarchical caching scheme. In their model, however, clients are assumed to have minimally-shared hierarchies such as home directories and temporary files stored on local disks, which is not an assumption we wish to make in our system. When the primary is far from the secondaries, however, even a relatively low intermediate cache hit rate is a significant advantage for a client whose files would otherwise have to be shipped from a distant site.

3 Failure Recovery

A major advantage of our design is that it makes the algorithms for failure recovery extremely simple. In this section, we describe how crashes and partitions are dealt with.

3.1 Secondary Server Failure

Failure of a secondary server will not be noticed unless so many secondaries fail that an initial `strict_read()` can no longer guarantee consistency.

Recovery is easy: the recovering secondary is not required to do anything when it comes back up because an initial strict read will always contact enough secondaries to ensure correctness. However, finding and copying over the most recent versions of files is a good idea because it will increase the likelihood that `loose_read()` will return up-to-date values. The process of getting up to date can be done simultaneously with accepting updates — there need not be a synchronous "update phase." This is a consequence of the quorum requirement: there are no assumptions made about which secondaries have the most recent values.

3.2 Primary Server Failure

It is the client's responsibility to detect when its primary server has crashed. When its requests go unanswered, the client asks the matchmaker for a new primary. By letting the client drive the process, and by recognizing that the new primary is not required to have up-to-date versions of any files, we avoid the complexity of a traditional election scheme [3].

Furthermore, a primary can crash (or just drop out) at any time without causing updates to be lost. Because a client is required to hold an update in its cache until receiving a purge notice — at which point the update is replicated on N or more secondaries — it is impossible for a primary to hold the only copy of an update. The newly-chosen primary will automatically restart the update propagation process for any updates remaining in the client's cache. No special logic is required; the new primary treats this case in exactly the same way as regular update propagation. The only minor inconvenience is that a client's currency tokens must be revalidated immediately after attaching to a new primary; the client-primary-secondary hierarchy upon which CTs rely may have been temporarily disrupted by the crash.

Recovery is trivial, again because all the state the primary held can be regenerated elsewhere. A primary that has come back up simply waits for the matchmaker to pair it up with clients. The recovering primary does not retain any state pertaining to its interactions with previous clients.

3.3 Client Failure

As in any write-back caching system, it is possible to lose data held at the client when the client crashes. Due to the use of asynchronous update propagation, our window of danger is somewhat wider than in other systems. However, if the client's cache is non-volatile, updates need not be lost, even if they are unavailable when a client is disconnected.

When a primary server is unable to make a pickup from a client, it assumes that the client has crashed and so it stops making pickups. If the client has not crashed, no harm is done: the client will eventually ask the matchmaker for a new primary. This is exactly what the client will do during recovery if it actually has crashed.

3.4 Reaction to Partition

During a network partition, an initial `strict_read()` cannot be performed in a partition containing fewer than T-N+1 or a majority of secondaries, whichever is greater. T-N+1 secondaries are needed to locate the most recent version of the file, while a majority is needed to read and update the PCW and CT lists.

Similarly, when fewer than N secondaries are reachable, the `write()` operation will block if the client's cache fills due to the primary's inability to propagate updates to a sufficient number of secondaries.

3.5 Resolution of Conflicting Updates

In general, the most recent version of a file, as determined by timestamp, is the one that will be retained. Timestamps are assigned by clients, but since clients are not trusted, timestamps are validated by primaries during each pickup. If the client-assigned timestamp falls outside the range of plausibility, the primary assigns its own timestamp and informs the client of the correct time for future use.

Even when `strict_read()` is used exclusively, conflicts can still occur. For example, two clients may strictly read the same file, and then make conflicting updates into their caches. Ideally, if client A updates a file while client B is modifying its cached copy, client B should eventually learn that its updates were based on stale data. Client B's updates should not be written to secondaries without explicit authorization from B when such a conflict occurs. Conflict detection is particularly important after a network partition heals: we do not want updates to be overwritten with inconsistent versions that happen to have later timestamps.

We detect conflicts in our system through the use of timestamps augmented with a unique identifier (such as an IP address) that indicates which client last modified the replica in question. [2] After a read or a successful write (signified by receipt of a purge notice), the timestamp/client pair associated with that version of the file is retained by the client. On future writes, if the service determines that a newer version *written by a different client* has superseded the cached copy, a message concerning the conflict is sent to the client currently attempting to write.

There is no guarantee that the conflict message will arrive at the client, but this is not a problem. In the worst case, another message will be sent during the next round of pickup and propagation.

Because of our asynchronous update propagation, a client must maintain *two* timestamp/client pairs for each file in its cache: a *base* timestamp, indicating the version upon which updates are based, and an *eventual* timestamp, which is the ID that will be assigned to the current version after propagation is complete. If a client is not a PCW for a given file, the base and eventual timestamps are identical. The eventual timestamp is given out to clients who read a file from a PCW's cache during an initial strict read: this is the version on which other clients' updates will be based, whether or not that version has been fully propagated.

The client ID is necessary to detect conflicts because no order is mandated for update propagation. Without knowing which client is responsible for an update, it would not be possible to tell the difference between out-of-order updates from a single client (which is fine), and out-of-order conflicting updates from multiple clients (which must be flagged).

It is possible for an irresponsible client to *loosely* read a file, modify it, and write it back. However, one of the central themes of our work is that the cost of consistency should be borne by the users who demand it. If a client with write access to a file refuses to use `strict_read()` before modifying the file, that's the client's problem. A user who refuses to bear the cost of consistency is not guaranteed to receive it. Since write access is (presumably) granted only to clients who are willing to take responsibility for modifying a file properly, conflicts resulting from this problem should be very rare in practice.

3.6 Accommodation of Wandering Users

Finally, there is a case — unique to mobile operation — that we call the *wandering user*. Suppose a client strictly reads a file just before a network partition occurs. This person

[2] Although version vectors are a more powerful conflict detection mechanism than timestamps, the added power would not provide a significant advantage in our system. Version vectors are useful for detecting conflicts in an *optimistic* system such as Coda [12], which always returns the best value it can find. In contrast, our `strict_read()` operation will not return any value unless a certain level of consistency can be assured, thereby preventing many potential conflicts from occurring. Since the use of version vectors requires synchronously executing a protocol to exchange and update vectors, we ruled out their use.

then makes several updates to the file, which are propagated to all reachable secondaries. Before the partition heals, the client wanders into a different partition and makes more updates to the same file, which are propagated to a different subset of secondaries.

This appears to be potentially chaotic, but actually works out quite well. The client can make as many updates as desired during the partition because unavailable replicas do not cause write operations to block. When the partition heals, timestamps make it easy to decide which versions the secondaries should retain. If several wandering users make updates to the same file during a partition, the conflict detection scheme described above will sort out the divergent versions.

3.7 Semantics

Having reviewed the operation of our design during both normal and failure conditions, we now summarize the semantics of the three operations.

- The value returned by `loose_read()` is unpredictable, as is the fate of a write following a loose read.
- When a `write()` operation following a strict read returns, the client is assured that, if it remains reachable long enough, then its update will be stored at N secondaries, and — if it has not been superseded by a more recent conflicting update — will be installed as the latest value.
- In the presence of an arbitrary combination of writes by both loose readers and strict readers, the value returned by `strict_read()` is unpredictable.
- In the presence of only writes and strict reads, a strict read operation will return the value which is the latest among:
  1. The latest value written by a strict reader and present at any secondary server. The propagation of values to secondaries is subject to variation depending on the pickup schedules of primaries.
  2. The latest value in the cache of any reachable client which is a PCW.

  When there are no failures, `strict_read()` returns the "latest" value; when there are failures, this fact can be detected and returned to the user.

4 Conclusion

We have presented the details of a variable-consistency file service, emphasizing the dynamic relationships of clients, servers, and files. In addition, we have described and analyzed the recovery procedures for various types of failures, including the detection of conflicting updates.

Although mobile operation served as the initial motivation of our thinking, we think our proposal contains profound advantages for "regular" operation as well. The major novel features of our design are:

1. An interface in which the required consistency must be declared by the reader.

2. Currency tokens, which can drastically reduce the cost of strict reads after the overhead of the initial strict read.

3. An asynchronous update propagation scheme driven by the primary server, in which client cache space is traded for a low-latency, reliable write operation.

The extra information provided to the service through the interface allows the implementation of both `write()` and `loose_read()` to be made completely lazy, and hence fast and scalable. Another major benefit is considerable simplification of the internal algorithms, especially those for recovery; very little state need be kept in non-volatile storage. In essence, our design heaps the entire burden of implementing consistency onto the initial strict read. Thanks to the currency token — which persists until explicitly revoked, independent of cache occupancy — an initial strict read need be done very infrequently.

In our design a "primary server" is not a true replication site, but rather a bigger cache for reading and a coordinator for the asynchronous update propagation. By using the primary in this way, a client can be ignorant about the actual location of file replication sites. More important, though, is the elimination of blocking during typical write operations, courtesy of the beneficent primary.

We see two major drawbacks in this design: the need to program according to a new interface (although this can be mitigated by using one of the generic read calls suggested in Section 2.1), and the mediocre performance of the initial strict read. Evaluation of this file service is a usability question; to clear it up, we are at work on a prototype implementation.

5 Acknowledgements

This work was supported in part by National Science Foundation grant CDA-9022123, IBM Corporation, and the Center for Telecommunications Research, an NSF Engineering Research Center funded by grant ECD-88-11111.

References

[1] M. Baker et al. Measurements of a Distributed File System. In *Proc. Thirteenth ACM Symp. on Operating System Principles*, pages 198–212, October 1991.

[2] M. Blaze and R. Alonso. Dynamic Hierarchical Caching in Large-Scale Distributed File Systems. In *Proc. Twelfth Intl. Conf. on Distributed Computing Systems*, June 1992.

[3] H. Garcia-Molina. Elections in a Distributed Computing System. *IEEE Trans. Computers*, C-31(1):48–59, January 1982.

[4] D. K. Gifford. Weighted Voting for Replicated Data. In *Proc. Seventh ACM Symp. on Operating System Principles*, pages 150–162, December 1979.

[5] C. G. Gray and D. R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proc. Twelfth ACM Symp. on Operating System Principles*, pages 202–210, December 1989.

[6] J. Ioannidis, D. Duchamp, and G. Q. Maguire, Jr. IP-based Protocols for Mobile Internetworking. In *Proc. SIGCOMM '91*, pages 235–245, September 1991.

[7] T. Mann, A. Hisgen, and G. Swart. An Algorithm for Data Replication. Technical Report 46, Digital Systems Research Center, June 1989.

[8] D. L. Mills. Network Time Protocol (Version 2) Specification and Implementation. Network Working Group RFC 1119, September 1989.

[9] D. Muntz and P. Honeyman. Multi-level Caching in Distributed File Systems. Technical Report CITI TR 91-3, University of Michigan, August 1991.

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

[11] J.-F. Paris. Voting with Witnesses: A Consistency Scheme for Replicated Files. In *Proc. Sixth Intl. Conf. on Distributed Computing Systems*, pages 606–612, 1986.

[12] M. Satyanarayanan et al. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Trans. Computers*, 39(4):447–459, April 1990.

[13] A. Siegel, K. Birman, and K. Marzullo. Deceit: A Flexible Distributed File System. In *Proc. 1990 Summer USENIX Technical Conf.*, pages 51–62. USENIX, June 1990.

[14] C. Tait and D. Duchamp. Service Interface and Replica Management Algorithm for Mobile File System Clients. In *Proc. First Intl. Conf. on Parallel and Distributed Information Systems*, pages 190–197, IEEE, December 1991.