Safety, Visibility, and Performance in a Wide-Area File System

Minkyong Kim, Landon P. Cox, and Brian D. Noble
Electrical Engineering and Computer Science
University of Michigan
{minkyong,lpcox,bnoble}@umich.edu

Abstract

As mobile clients travel, their costs to reach home filing services change, with serious performance implications. Current file systems mask these performance problems by reducing the *safety* of updates, their *visibility*, or both. This is the result of combining the propagation and notification of updates from clients to servers.

Fluid Replication separates these mechanisms. Client updates are shipped to nearby replicas, called *WayStations*, rather than remote servers, providing inexpensive safety. WayStations and servers periodically exchange knowledge of updates through *reconciliation*, providing a tight bound on the time until updates are visible. Reconciliation is non-blocking, and update contents are not propagated immediately; propagation is deferred to take advantage of the low incidence of sharing in file systems.

Our measurements of a Fluid Replication prototype show that update performance is completely independent of wide-area networking costs, at the expense of increased sharing costs. This places the costs of sharing on those who require it, preserving common case performance. Furthermore, the benefits of independent update outweigh the costs of sharing for a workload with substantial sharing. A trace-based simulation shows that a modest reconciliation interval of 15 seconds can eliminate 98% of all stale accesses. Furthermore, our traced clients could collectively expect availability of five nines, even with deferred propagation of updates.

1 Introduction

Mobile devices have become an indispensable part of the computing infrastructure. However, networking costs continue to render them second class citizens in a distributed file system. Limits imposed by networking costs are not new, but they are no longer due to "last mile" constraints. With the widespread deployment of broadband connectivity, mobile users often find themselves in a neighborhood of good network performance. Unfortunately, they must reach back across the wide area to interact with their file servers; the latency and congestion along such paths impose a substantial performance penalty.

To cope with increased costs, wide-area and mobile file systems employ two techniques to limit their use of the remote server. The first technique is *caching*. When workloads have good locality and infrequent sharing, caching can avoid most file fetches from the server. The second technique is *optimistic concurrency control*. Clients defer shipping updated files until time and resources permit. Ideally, most updates will be overwritten at the client, and need never be sent to the server.

Deferring updates improves performance, but harms safety and visibility. An update is *safe* if it survives the theft, loss, or destruction of the mobile client that created it. Safety requires that the contents of an update reside on at least one other host. An update is *visible* if every other client in the system knows it exists. Visibility requires that notification of an update reaches all replicas in the system.

Current file systems needlessly combine safety and visibility by propagating the contents of an update, implicitly notifying the destination that the update exists. *Fluid Replication* separates the concerns of safety and visibility through the addition of secondary replica sites. These sites, called *WayStations*, act as server replicas to nearby clients, servicing uncached reads and writes. Client writes are propagated to a WayStation immediately, providing safety. WayStations periodically *reconcile* their updates with the servers for which they act as replicas; this exchanges update notifications, but not the contents of those updates. Since reconciliation involves only meta-data, it can be done frequently, providing bounded visibility. Put another way, Fluid Replication aggressively writes updates back to the WayStation, but periodically invalidates updates between WayStation and server.

To maintain a simple consistency model, Fluid Replication provides *copy semantics*; each replica behaves as if a reconciliation actually copied all new versions accepted by the other replica. To support this, each WayStation retains file versions in *escrow* in the event that they are referenced at other replicas. Escrow also allows us to provide wait-free reconciliations with no additional mechanism.

We have built a prototype of Fluid Replication, and evaluated it with a variety of benchmarks. Updates are isolated from wide-area networking costs in the absence of sharing; clients pay wide-area costs only for accesses to shared files. A trace-based simulation of Fluid Replication shows that a modest reconciliation interval of fifteen seconds provides a stale access rate of only 0.01%, compared to a sharing rate of 0.6%. Space devoted to escrow storage is modest but bursty, with a high water mark of less than 10 MB. The impact of deferred propagation on availability is also small; our traced client population would expect to see one failed access per year, collectively.

2 Related Work

A number of projects have explored file systems for mobile, wide-area clients. Many of the ideas in Fluid Replication stem from the Coda file system. Coda provides high availability through disconnected operation [13] for clients and server replication [16] between well-connected servers. Fluid Replication is orthogonal to both; we do not expect clients to avoid disconnection or server failure. Coda also supports weakly-connected operation [18] to utilize low bandwidth between a client and its servers. During weakly-connected operation, Coda clients defer updates, and ship them later through a process called *trickle reintegration*. This sends both the knowledge of an update and its contents to the server. Since this is expensive, Coda defers reintegration in the hopes that some updates will be canceled by overwrites or deletions. This *aging window* is set to ten minutes to capture an acceptable fraction of possible optimizations, trading network bandwidth for more aggressive propagation.

Ficus [9, 23] shares Coda's goal of providing optimistic file access, but uses a peer-to-peer architecture. Each update is accepted by a single replica and asynchronously propagated to other sites, but no effort is made to ensure that update messages arrive. To propagate missed updates, Ficus provides copy semantics through the actual exchange of updates between two replicas. Updates are discovered by a disk scan at each replica, a heavyweight process. Reconciliation exchanges both knowledge and content of updates. Since this is expensive, it is intended only for well-connected peers, making visibility dependent on the mobility pattern of clients.

Bayou [26] provides optimistic concurrency control for database applications, where distributed updates are eventually committed by a primary replica. Deno [11] extends the Bayou model to provide distributed commit. These systems exchange full update sets whenever convenient. Unlike Ficus, they log updates; unlike Coda, updates are represented by-operation rather than by-value. This inherently combines notification of an update and its contents. However, since databases exhibit many fine-grained, partial updates, this may be less of an issue than for file systems. Exchanges between peers rely on client mobility and communication patterns. So, these systems can offer eventual visibility, but cannot bound the time required to do so.

OceanStore [15] advocates an architecture for global storage. Like Fluid Replication, it envisions storage supported by a loose confederation of independent servers. A primary focus of OceanStore is safely using untrusted nodes in the infrastructure. This is also an important problem for Fluid Replication, which we address elsewhere [20]. OceanStore's consistency mechanism, like that of Ficus and Bayou, is based on epidemic algorithms, and so cannot bound visibility. OceanStore represents updates by operation rather than by value, combining notification of updates with the shipment of their contents.

TACT [33] provides a middleware service to coordinate the consistency of data used by wide-area applications. Unlike Fluid Replication, it combines visibility and safety of updates. However, it does provide strong bounds on visibility. In addition to temporal bounds—which are offered by Fluid Replication—it can also bound the number of unseen writes or the *weight* of those writes. Combining Fluid Replication's separation of safety and visibility with TACT's mechanisms to trigger reconciliation can offer applications a tunable consistency mechanism with isolation from wide-area networking costs.

xFS is a file system designed for tightly coupled clusters of workstations [1]. However, early designs focused on the wide area [31]. In this model, clients were aggregated based on physical proximity, and served by a *consistency server*. On update, a client would retain the contents of the update—reducing safety—and notify the consistency server that the update occurred. Consistency between these second-level replicas and the main storage server was to be managed conservatively, imposing wide-area networking costs on normal operation.

Finally, JetFile [8] is motivated by the same concerns as the early xFS design, though it provides a number of additional features. It uses a central version server that provides serialization, but does not hold data; objects are stored primarily on the client that creates them, reducing safety. Clients locate files with scalable reliable multicast [7]. As in Ficus, invalidations are sent best-effort. However, in JetFile, the central version server broadcasts invalidations periodically to ensure bounded consistency; this period provides consistency equivalent to Fluid Replication. JetFile's most serious drawback is that it depends on ubiquitous IP multicast.

3 Design

This section presents the design of Fluid Replication with an eye towards separating the concerns of safety and visibility. Clients treat WayStations, wide-area replica sites, exactly as they would a server. Each WayStation reconciles its updates with the server periodically; reconciliation exchanges notification of updates, but not their contents. In order to provide copy semantics without actually copying during reconciliation, replicas hold reconciled updates in *escrow*. Due to the escrow mechanism, reconciliation does not require any node to hold a lock during a network round trip.

3.1 Context

Fluid Replication is designed to complement a client-server architecture, such as that of AFS [10]. In the AFS model, servers provide the centralized point of administration. They are expected to be carefully maintained to provide the best possible availability to their clients. Clients consist of workstations and mobile devices. Clients are considered the property of their users; they are not carefully administered and are much more likely to fail. This is particularly true of mobile clients.

The AFS model uses *callbacks* for consistency. Before using an uncached file, a client must first fetch it from the server. As a side effect of the fetch, the server establishes a callback on that file. The client can then use the now-cached copy. If a client modifies a file, the new version is sent to the server on `close`; this is called a *store* event. The server *breaks callback* on any other clients caching this file, invalidating their copies, before accepting the store. Unix semantics forbids invalidation of an open file; such files are invalidated after they are closed. If a file subject to pending invalidation is modified, the close returns an error.

3.2 WayStations: Hosting Replicas

The success of Fluid Replication depends on *WayStations*, which act as replicas for servers. They provide caching and immediate safety to nearby clients without penalizing performance, and act as reconciliation partners with remote servers. A WayStation is able to host replicas of any service for any client, regardless of administrative domain, and can host replicas of multiple servers concurrently.

In our architecture, WayStations play a role similar to that of servers; one model for a WayStation is as a fee-based service, much as broadband connectivity is for travelers today. In contrast to clients, they are carefully administered machines, and are relatively permanent members of the infrastructure. While they can be independently created, they are not expected to be transient. For example, a file server could also provide WayStation services to foreign clients. However, an end-user machine is not an appropriate WayStation because it may be turned off or disconnected at any time.

The decision of whether or not to use a WayStation is client-driven. Clients estimate the quality of the network between themselves and their current replica site [12]. If the network quality becomes poor, the client looks for a WayStation close enough to improve matters, and initiates a replica on it. Such searches are carried out in a neighborhood near the client through a process called *distance-based discovery* [19]. The WayStation informs the server of initiation. This notification is the point at which the invalidation semantics on a client change. It is an expensive operation, but it enables the server to drop any outstanding callbacks for this client. Requiring callback breaks for wide-area clients would substantially penalize local-area clients.

The WayStation fetches objects on demand. We considered prefetching, but rejected it. It is not clear that inferred prefetching [17] can provide requests in time to be useful given wide-area delays. We are also unwilling to require that applications disclose prefetch requests [24].

Foreign clients must trust WayStations to hold cached files securely and reconcile updates promptly. While one cannot prevent WayStation misbehavior completely, we plan to provide mechanisms to prevent exposure and reliably detect data modification or repudiation of accepted updates. This allows WayStation/client relationships to be governed by a contract. Parties to a contract are not prevented from breaching it; rather, the contract specifies penalties and remedies in the event of a breach. The details of our security architecture are beyond the scope of this paper [20]. We comment further on it in the context of future work in Section 6.

3.3 Reconciliation: Managing Replicas

Once a client has selected a WayStation, that WayStation is treated as the client's file server. Clients cache files from WayStations, which manage callbacks for those clients. Dirty files are written back to WayStations on `close`. WayStations maintain update logs using a mechanism similar to Coda's *client modify log* [13]. It contains file stores, directory operations, and meta-data updates. Redundant log records are removed via cancellation optimizations.

The server maintains an update log whenever one or more WayStations hold replicas of that server. The server also tracks which files are cached at the WayStation, called the *interest set*. The bookkeeping for interest sets is similar to that for callbacks, but the information is used only during reconciliation.

Periodically, each WayStation reconciles its update log with that of the server, exposing the updates made at each replica to the other one. To initiate a reconciliation, the WayStation sends the server its update log plus a list of files the WayStation has evicted from its cache. The server removes the latter from the WayStation's interest set, and checks each WayStation log record to see if it is serializable. If it is, the server invalidates the modified object, and records the WayStation as the replica holding that version.

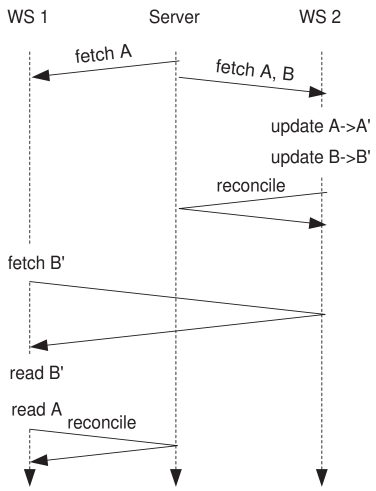
The server responds with a set of files to invalidate, and a set of files that are now in conflict, if any. The invalidation set consists of any updates accepted by the server—whether from a client or through another WayStation—that also reside in the WayStation's interest set. Since the WayStation will invalidate them, the server can remove them from the interest set as well.

WayStations truncate their update logs on successful reconciliation. However, since the server maintains a merged log of all replica sites, it cannot truncate its log immediately. The server maintains the last reconciliation time for each WayStation holding a replica; let t_{oldest} be the earliest such time. The server need only hold log records from t_{oldest} forward, since all older records have already been seen by all WayStations. Slow WayStations can force the server to keep an arbitrarily large log. This policy is in contrast to that taken by Coda's implementation of optimistic server replication. In Coda, only the tails of large update logs are retained; objects with discarded log records are marked in conflict and will need manual repair. Given Coda's presumption of tightly-coupled and jointly-administered replicas, this is an appropriate design choice. However, since WayStations are administered by independent entities, it would be unwise to allow a WayStation's absence to necessitate many manual repairs.

If each WayStation reconciles at a constant rate, all updates are globally visible within twice the longest reconciliation period. The first reconciliation invalidates the server copy, and all other WayStation copies are invalidated during the next round. In order to provide clients with well-bounded visibility, reconciliation must be a lightweight operation. This is why reconciliations exchange only notification of updates, not their contents. Because sharing is rare, aggressively exchanging file contents increases reconciliation time without improving client access time. Leaving shared accesses to pay the full cost of wide-area network delays preserves performance, safety, and visibility for common-case operations.

3.4 Escrow: Providing Copy Semantics

When a replica site needs the contents of an invalidated file, what version does it obtain? The simplest approach would be to provide the most recent version at the time



This figure shows how fetching the most recent version of a missing file can result in the propagation of out-of-order updates. WayStation 2 updates A then B. WayStation 1 sees the update to B first, because A had already been cached without an intervening reconciliation.

Figure 1: Propagating Updates Out-Of-Order

it is requested; we call this *last-version semantics*. Unfortunately, last-version semantics allow WayStations to see updates out of order. Instead, Fluid Replication's copy semantics guarantees that updates are seen in the order they are made. We believe this is important, as other studies have shown that users do not always understand the implications of complicated consistency mechanisms [6].

Figure 1 illustrates how out-of-order updates can arise under last-version semantics. There are two files, A and B, shared by clients at two WayStations, WS_1 and WS_2. Assume that WS_1 caches A, and WS_2 caches both A and B. Suppose that a client at WS_2 updates A then B, and then reconciles. The server now knows about both new versions held at WS_2. If a client at WS_1 were to reference both files A and B without an intervening reconciliation, it would see the old version of A but the new version of B. When WS_1 eventually reconciles, the update of A would be known, but it would have been seen out of order.

One solution would be to require clients to fetch all updated objects of interest whenever fetching any. Fluid Replication eschews this approach, as it would result in more work across the wide-area path between WayStation and server. Instead, we appeal to copy semantics. Under copy semantics, WS_1 would have read B rather than B', because B was the current version when WS_1 last reconciled. This is despite the fact that B was not in WS_1's cache and the update, B', had already been reconciled by WS_2. Copy semantics guarantees that each

WayStation see a complete prefix of all activity at each WayStation, and therefore provides a consistent view of the system. It is important to note that providing copy semantics does not group related updates into atomic actions. In other words, copy semantics cannot prevent seeing one update but not a related one. However, all such mis-matches will be explicable in reference to “real time”. In Bayou’s terminology, this provides *monotonic writes* [30].

To provide copy semantics to a WayStation requesting a file, Fluid Replication must supply the version known to the server at the WayStation’s last reconciliation. This version may reside on the server directly, or it may reside on the WayStation that had accepted it. Therefore, a replica site that makes an update visible to another via reconciliation must retain a copy of the update for as long as the other replica might refer to it. We say that such copies are held in *escrow*.

If a client at the server refers to a version escrowed at some WayStation, the server *back-fetches* it; responsibility for the escrowed version passes from WayStation to server. If a client at another WayStation references it, the fetch request is first sent to the server. The server obtains it from the escrowing WayStation—back-fetching as before—and then forwards it on to the requesting WayStation. Responsibility for escrow always migrates from a WayStation to the server, never the reverse.

For escrow to be practical, we must prevent unbounded storage growth. To see how this might occur, consider a WayStation at which a file is updated between each reconciliation. Without knowing which versions are visible to other replica sites, the WayStation would be forced to hold all of them. The key to managing this growth is to observe that only the *most recent* version visible to each replica need be held in escrow. Any other versions are said to be *irrelevant*, and can be safely pruned.

A simple way to prune versions is to track which are old enough to be globally irrelevant. Recall that t_{oldest} is the latest time after which all WayStations have reconciled successfully. Sending this time to a WayStation during a reconciliation allows the WayStation to prune old versions. Let $f_1 \ldots f_n$ be the sequence of updates to some file f at a WayStation that were reconciled with the server. Let f_i be the most recent of those updates performed before t_{oldest}. Clearly, all versions $f_1 \ldots f_{i-1}$ are irrelevant; each WayStation would request either f_i or some later version.

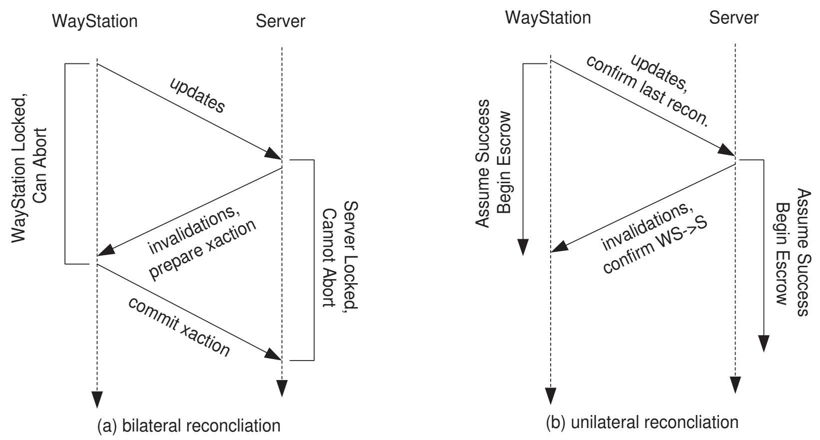
If all WayStations reconcile at the same rate, each WayStation must keep at most one additional version of a file in escrow. This is because each new reconciliation would find that t_{oldest} had advanced past the time of that WayStation’s last reconciliation. A version enters escrow only when a more recent version is created; after reconciliation, t_{oldest} advances past the new version, rendering the old one irrelevant. Note that this scheme allows some versions of a file stored at WayStations to be discarded without ever being sent to the server. However, the t_{oldest} mechanism guarantees that no replicas still hold an active reference to that version.

While this pruning is helpful, it does not prevent the possibility of unbounded escrow space. Consider what happens if one WayStation is slow to reconcile. This single WayStation prevents the advancement of t_{oldest}, requiring the retention of all future versions at each WayStation. In effect, a single WayStation can hold all others hostage. So, instead of sending only t_{oldest}, the server sends a sorted list of timestamps $t_{oldest} \ldots t_{newest}$, where each entry lists the time of last reconciliation by some WayStation. If two or more visible versions lie between adjacent timestamps, only the last one is needed. Under this scheme, a slow-to-reconcile WayStation requires only one additional copy in escrow.

The escrow mechanism depends on the fact that WayStations are closer in spirit to servers than they are to clients. Because they hold versions in escrow for other replicas, they must be reasonably available. An alternative design for Fluid Replication allows a client to act as its own WayStation. While this approach would be simpler, it can leave escrowed copies unavailable to other sites. End-user machines are often unavailable for days at a time, and may even disappear completely [4].

Under escrow, file updates can reside on WayStations indefinitely. One could argue that this is proper; if an update is never needed elsewhere, it should never be shipped from the WayStation. However, there are two reasons to ship versions in advance of need. First, administrative tasks such as backup are greatly simplified if files eventually migrate to the server. Second, migration limits the risk of unavailable escrowed copies and data loss. Therefore, any version that has not been updated for one hour is sent to the server. We chose one hour to take advantage of most cancellation opportunities [18], while balancing propagation overhead with the need for availability.

It is important to note that copy semantics cannot guarantee in-order updates if those updates are destined for more than one server. This is because there is no common synchronization point. A WayStation could provide this guarantee through synchronous, atomic reconciliations across all of its servers. However, such steps introduce wide-area latencies into the critical path of operations, counter to Fluid Replication’s philosophy. Alternatively, the servers could synchronize amongst themselves, but with similar drawbacks. We expect that most users will have one ”home” file system, obtaining the



This figure illustrates the difference between bilateral and unilateral reconciliations. Bilateral reconciliations require three messages. Each replica must be locked during an expensive round trip, and WayStation failures can hold the server hostage. In contrast, unilateral reconciliations optimistically assume that exposures are successful, and begin escrow immediately. They require two messages rather than three, employ only short-term, local locks, and hold no replicas hostage to failure.

Figure 2: The Advantage of Unilateral Reconciliations

benefit of in-order updates while paying only the small cost of local escrow.

3.5 Fault Tolerance in Reconciliation

The simplest model for reconciliation is *bilateral*: the atomic exchange of all update log records between a WayStation and a server. Unfortunately, this simple model is problematic in the face of node or network failures. Atomic exchanges require a two-phase commit protocol. One node must *prepare* the transaction, agreeing to either commit or abort until the other party confirms the result. In the meantime, the prepared node must block many operations until the transaction completes.

The difficulty is caused by store operations during a bilateral reconciliation. These stores cannot be serialized before the reconciliation. Doing so would require that they had been in the reconciled update log, which is impossible. The stores cannot be serialized after the reconciliation either, since they may refer to a file that the reconciliation will invalidate. Therefore, store operations issued during a bilateral reconciliation must block until it completes. In the presence of failures, stores may be blocked indefinitely. Put another way, bilateral reconciliation imposes wide-area networking costs on clients even in the absence of sharing; this runs counter to Fluid Replication's philosophy.

In light of these problems, we split a single, bilateral reconciliation into two *unilateral* ones. These alternatives are illustrated in Figure 2. WayStations initiate reconciliations, as before. However, as soon as the reconciliation message is sent, the WayStation assumes it will be received successfully. It can continue processing client requests immediately, placing versions in escrow as needed. The server likewise assumes success and begins escrow after sending its message. As a side effect, the server's message confirms the exposures assumed by the WayStation. The next WayStation request confirms the completion of the prior reconciliation.

There are several benefits to this approach. WayStations block updates only while computing a reconciliation message; no locks are held across expensive round trips. This appeals to the common case by placing potentially-visible versions in escrow immediately, rather than waiting for confirmation. Since the escrow mechanism is needed to provide copy semantics, no additional complexity is required. Finally, the third reconciliation message—required to keep bilateral locking times short—is implied by future messages. This *piggybacking* is similar to the process Coda uses to manage updates to well-connected, replicated servers [27]. Unlike Coda's COP2 messages, confirmation of reconciliations can be deferred indefinitely. The only penalty of an undetected failure is a larger escrow.

Unilateral reconciliations provide a good approximation to the desired effect of bilateral ones. The server sees a single, atomic event, while WayStations are able to allow clients to continue during the wide-area operation. In addition to a potential increase in escrow size, there is a slight widening of the conflict window, because concurrent writes are now allowed. Suppose that a WayStation initiates a reconciliation, then accepts an update to file f that is later invalidated by the server. This causes f to be in conflict. With bilateral reconciliation, the update to f would have been delayed until after invalidation and then rejected, avoiding a conflict. However, given the low incidence of write-shared files—particularly over such short time frames—it is unlikely that such spurious conflicts will occur in practice.

4 Implementation

Our Fluid Replication prototype consists of three components: a client cache manager, a server, and a WayStation. Each of these components is written primarily in Java. There are several reasons for this, foremost among them Java's clean combination of thread and remote procedure call abstractions and the benefits of writing code in a type-safe language.

The bulk of the Fluid Replication client is implemented as a user-level cache manager, supported by a small, in-kernel component called the *MiniCache* [29]. The MiniCache implements the vnode interface [14] for Fluid Replication. It services the most common operations for performance, and forwards file operations that it cannot satisfy to the user-level cache manager.

Calls are forwarded from the kernel to the cache manager across the Java Native Interface, JNI. The calls are then satisfied by one of a set of worker threads, either from the local disk cache or via Remote Method Invocation, RMI, to the appropriate replica. Fluid Replication uses write-back caching; a close on a dirty file completes in parallel with the store to the replica. The client supports dynamic rebinding of servers and WayStations for migration; currently, our prototype migrates only on user request.

WayStations and servers share much of the same code base, since their functionality overlaps. Data updates are written directly to the replica's local file system. Metadata is stored in memory, but is kept persistently. Updates and reconciliations are transactional, but we have not yet implemented the crash recovery code. We use Ivory [2] to provide transactional persistence in the Java heap; used in this way, it is similar to RVM [28].

The decision to write the client in Java cost us some performance, and we took several steps to regain ground. Our first major optimization was to hand-serialize RMI messages and Ivory commit records. The default RMI skeleton and stub generator produced inefficient serialization code, which we replaced with our own. This reduced the cost of a typical RMI message by 30%.

The second major optimization concerned the crossing of the C-Java boundary. Each method call across this boundary copies the method arguments onto the Java stack, and returned objects must be copied off of the Java stack. We were able to avoid making these copies by using *preserialized objects*, provided by the Jaguar package [32]. Jaguar allows objects to be created outside the Java VM, and still be visible from within. We used this to pass objects, copy-free, between our C and Java code.

5 Evaluation

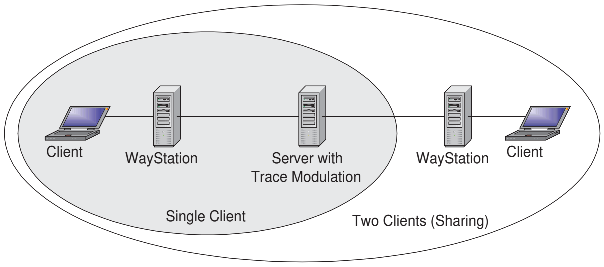
In evaluating Fluid Replication, we set out to answer the following questions:

- Can Fluid Replication insulate clients from wide-area networking costs?
- What is the impact of sharing on performance?
- How expensive is reconciliation?
- Can Fluid Replication provide the consistency expected by local-area clients to those in the wide area?
- How does escrow affect availability?
- What are the storage costs of escrow?

These questions concern the performance seen by individual clients and the behavior of the system as a whole. To measure client performance, we subjected our prototype to a set of controlled benchmarks. We explored system behavior through the use of trace-based simulation.

Our benchmarks ran on the testbed depicted in Figure 3. The WayStations are connected to the server via a *trace modulated* network. Trace modulation performs application-transparent emulation of a slower target network over a LAN [22]. We have created modulation traces that emulate the performance of a variety of different wide-area networking scenarios, listed in Table 1. Latency numbers in these traces are in *addition* to baseline latency, while bandwidth specifies the bottleneck capacity. All testbed machines run the Linux 2.2.10 kernel. The server and WayStations have 550 MHz Pentium III Xeon Processors, 256 MB of RAM, and 10K RPM SCSI Ultra Wide disks. The clients are IBM ThinkPad 570s; these machines have 366 MHz mobile Pentium IIs with 128 MB of memory.

For the trace-based studies, we collected traces comprising all activity on a production NFS server over one week. This server holds 188 users' home directories, plus various collections of shared data, occupying 48 GB. The users are graduate students, faculty, and staff spread throughout our department. They come from a variety of research and instructional groups, and have diverse storage needs. Generally speaking, the clients are not mobile, so they may not be wholly representative of our target domain. However, prior studies suggest that, at the operation level captured by our traces, mobile and desktop behavior are remarkably similar [21].



This figure illustrates our benchmarking topology. Each client is well connected to its WayStation, but traffic between a WayStation and the server is subject to trace modulation.

Figure 3: Benchmark Topology

Scenario	Latency (ms)	Bandwidth
local area	0.0	10 Mb/s
small distance	15.0	4 Mb/s
medium distance	33.5	3 Mb/s
large distance	45.0	1 Mb/s
intercontinental	55.0	1 Mb/s
low bandwidth	0.0	56 Kb/s
high latency	200.0	10 Mb/s

This table lists the parameters used in each of our trace modulation scenarios. The local area scenario is the baseline against which we compare. The next four were obtained by measuring small `ping` and large `ftp` performance to four different sites. Bandwidth numbers are increased over `ftp` throughput by 20% to account for the difference in metric. The last two are used only to determine trends as latency or bandwidth worsen, and modify parameters orthogonally. Latencies are one-way; bandwidths are symmetric.

Table 1: Trace Modulation Parameters

Traces were collected using `tcpdump` on the network segment to which the server was attached. These packet observations were then fed into the `nfstrace` tool [3], which distilled the traces into individual fetch and store operations. Note that this tool does not record operations satisfied by a client's cache. However, since NFS clients do not use disk caches, this will overstate the amount of read traffic a Fluid Replication client would generate. For the purposes of our analyses, we assume that each client host resides on a separate WayStation. The traces name 84 different machines, executing 7,980 read operations and 16,977 write operations. There are relatively few operations because most of our client population did not materially contribute to the total. Seven hosts account for 90% of all requests, and 19 hosts account for 99% of all requests.

5.1 Wide-Area Client Performance

How effectively does Fluid Replication isolate clients from wide-area networking costs? To answer this question, we compare the performance of Coda, AFS, and Fluid Replication in a variety of networking conditions. For Coda, we ran Coda 5.3.13 at the server and the client. For AFS, we used OpenAFS, a descendant of AFS 3.6, as the server, and Arla 0.35.3 as the client. To provide a fair comparison, we set up our Coda volume on a single server, rather than a replicated set.

Our benchmark is identical to the Andrew Benchmark [10] in form; the only difference is that we use the *gnuchess* source tree rather than the original source tree. Gnuchess is 483 KB in size; when compiled, the total tree occupies 866 KB. We pre-configure the source tree for the benchmark, since the configuration step does not involve appreciable traffic in the test file system. Since the Andrew Benchmark is not I/O-bound, it will tend to understate the difference between alternatives. In the face of this understatement, Fluid Replication still outperforms the alternatives substantially across wide-area networks.

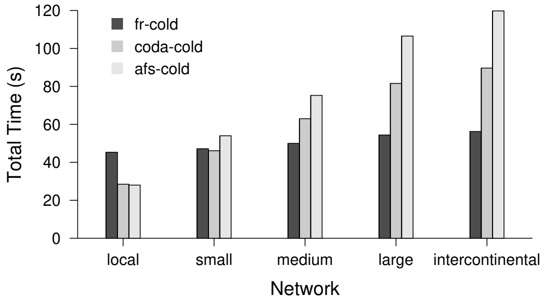
We tested each file system with both cold and warm caches. In the case of AFS and Coda, a "warm cache" means that the clients already hold valid copies of the *gnuchess* source tree. In the case of Fluid Replication, the source tree is cached on the WayStation.

Figures 4 compares the total running times of the Fluid Replication, Coda, and AFS clients under different network environments. Figure 4(a) gives the performance with a cold cache, and Figure 4(b) shows it with a warm cache. Each experiment comprises five trials, and the standard deviations are less than 2% of the mean in all cases.

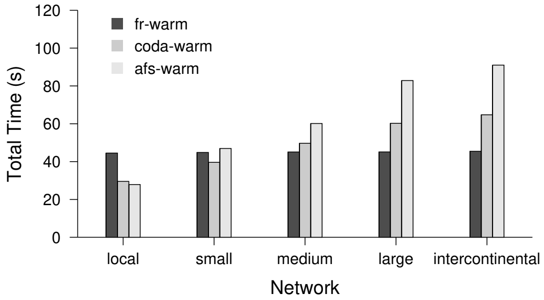
With a cold cache and a well-connected server, Coda and AFS outperform Fluid Replication. We believe this is due to our choice of Java as an implementation language and the general maturity level of our code. We see no fundamental reason why Fluid Replication's performance could not equal Coda's. They have identical client architectures, and the client/WayStation interactions in Fluid Replication are similar to those between client and server in Coda.

As the network conditions degrade, the cold-cache times of Coda and AFS rapidly increase while those of Fluid Replication increase slowly. All systems must fetch source tree objects from the server, and should pay similar costs to do so. The divergence is due to the systems' different handling of updates. In Fluid Replication, all updates go only to the nearby WayStation. In Coda and AFS, however, updates must go across the wide area to the server.

With a warm cache, the total running time of Fluid Replication remains nearly constant across all network environments. This is because the updates are propagated



(a) cold cache



(b) warm cache

This figure compares the total running times of Coda, AFS and Fluid Replication under a variety of network environments. The upper figure gives the results for a cold cache, and the lower figure shows them for a warm cache. With a cold cache, Fluid Replication is least affected by network costs among three systems. With a warm cache, the performance of Fluid Replication does not appreciably degrade as network costs increase.

Figure 4: Client Performance

only to the WayStation; invalidations are sent to the server asynchronously. The running time of AFS and Coda increases as the network degrades. They must propagate updates to the server during the *copy* phase and write object files back to the server during the *make* phase. The Coda client never entered weakly-connected mode, since the bandwidths in our sample traces were well above its threshold of 50 KB/s. Had Coda entered weakly-connected mode, its performance would be identical to Fluid Replication's, but its updates would be neither safe nor visible for many minutes.

Table 2 shows the normalized running time of Fluid Replication, with the local-area case serving as the baseline. While the running time increases by as much as 24.1% when the cache is cold, it remains nearly constant when the cache is warm. Looking only at the four scenarios generated by `ping` and `ftp` experiments, there appears to be a slight growth trend, but it is within observed variance. Only the results for Small and Intercontinental are not identical under the t-test; all other pairs are sta-

Trace	Cold	Warm
Small	1.040	1.007
Medium	1.103	1.012
Large	1.200	1.013
Intercontinental	1.241	1.020
Low Bandwidth		1.014
High Latency		1.007

This figure shows the normalized running time for Fluid Replication over each of the networking scenarios in Table 1. The local-area case served as the performance baseline in each case. When the WayStation cache is cold, performance decreases as wide-area networking costs increase. However, no consistent trend exists when the WayStation cache is warm.

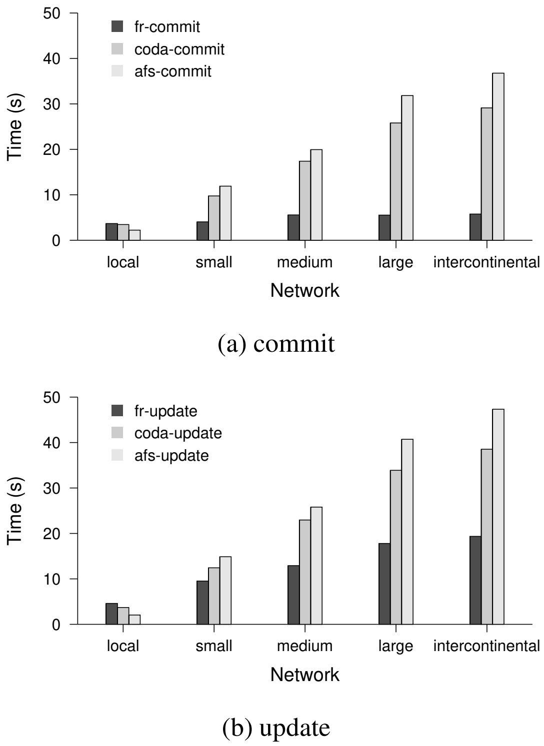
Table 2: Fluid Replication over Wide-Area Networks

tistically indistinguishable. To conclusively rule out a true trend, we also ran the warm-cache Fluid Replication experiment across two more-demanding networking scenarios. The first decreased bandwidths to 56 Kb/s, but added no additional latency. The second increased one-way latency to 200 ms, but placed no additional bandwidth constraints. In both cases, Fluid Replication's running time was less than that for the Intercontinental trace. Therefore, we conclude that that Fluid Replication's update performance does not depend on wide-area connectivity. Of course, this is due to deferring the work of update propagation and the final reconciliation — neither of which contribute to client-perceived performance. Sections 5.4 and 5.5 quantify any reduction in consistency or availability due to deferring work.

5.2 Costs of Sharing

Our next task is to assess the potential impact that deferred propagation has on sharing between wide-area clients. We devised a benchmark involving two clients sharing source code through a CVS repository. In the benchmark, clients C1 and C2 are attached to WayStations W1 and W2, respectively. In the first phase, C1 and C2 each check out a copy of the source tree from the repository. After changing several files, C1 commits those changes to the repository. Finally, C2 updates its source tree. Both the repository and working copies reside in the distributed file system. When the benchmark begins, both clients have the repository cached.

We used the Fluid Replication source tree for this benchmark. The changes made by C1 are the updates made to our source tree during the four of the busiest days recorded in our CVS log. At the beginning of the trial, the source tree consists of 333 files totaling 1.91 MB. The committed changes add four new files, totaling seven KB, and modify 13 others, which total 246 KB. We reran this activity over the two-client topology of Figure 3.



(a) commit

(b) update

Our sharing benchmark replayed sharing from the CVS log of our Fluid Replication source tree. The benchmark consisted of five phases: a checkout at C1, a checkout at C2, an edit at C1, a commit at C1, and an update at C2. This figure shows the time to complete the commit and update phases for AFS, Coda, and Fluid Replication.

Figure 5: Sharing Benchmark

Figure 5 shows our results for the commit and update phases, run over Fluid Replication, AFS, and Coda; each bar is the average of five trials. We do not report the results of the checkout and edit phases, because they exhibit very little sharing.

Unsurprisingly, the cost of committing to the repository was greater for AFS and Coda than for Fluid Replication. This is because Fluid Replication clients ship updates to a nearby WayStation. AFS and Coda must ship data and break callbacks over the the wide area.

What is surprising is that the update phase is also more costly for AFS and Coda. One would think that Fluid Replication would perform poorly, since file data has to traverse two wide area paths: from the first WayStation to the server, and then to the second WayStation. The unexpected cost incurred by AFS and Coda stems from the creation of temporary files. The latter are used to lock the repository and back up repository files in case of failure. WayStations are able to absorb these updates, and later optimize them away, since they are all subject to cancellation. AFS and Coda clients, on the other hand, send every update to the server. Furthermore, these updates break callbacks on the far client.

It is important to note that hiding the creation of temporary locking files may improve performance, but it renders the intended safety guarantees useless. The best that Fluid Replication can offer in the face of concurrent updates is to mark the shared file in conflict; the users must resolve it by hand. Coda would be faced with the same dilemma if it had entered weakly-connected mode during this benchmark. However, we believe that in practice, optimism is warranted. Even in the case of CVS repositories, true concurrent updates are rare. Our own logs show that commits by different users within one minute occurred only once in over 2,100 commits. A commit followed by an update by different users within one minute happened twice.

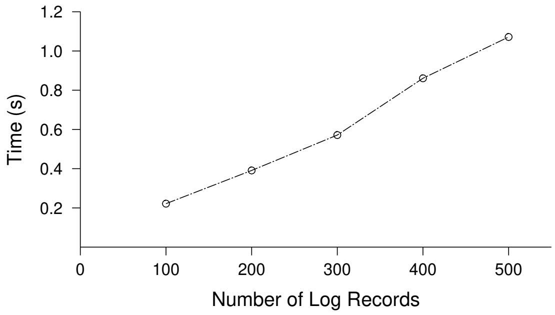
This benchmark illustrates the cost of obtaining deferred updates. However, in practice, these files are likely to have migrated to the server. Our logs show that the median time between commits by different users was 2.9 hours, and that the median time between a commit and an update by different users was 1.9 hours. This would provide ample opportunity for a WayStation to asynchronously propagate shared data back to the server before it is needed.

5.3 Reconciliation Costs

To be successful, Fluid Replication must impose only modest reconciliation costs. If reconciliations are expensive, WayStations would be able to reconcile only infrequently, and servers could support only a handful of WayStations. To quantify these costs, we measured reconciliations, varying the number of log records from 100 to 500. To put these sizes in context, our modified Andrew Benchmark reconciled as many as 148 log records in a single reconciliation.

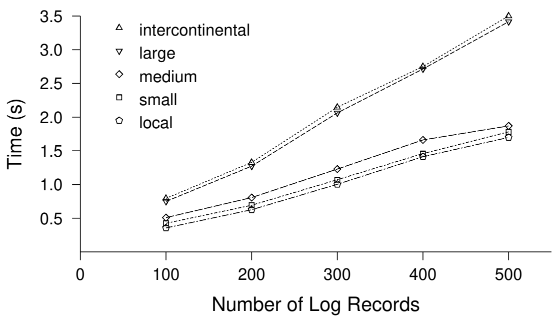
Figure 6 shows the reconciliation time spent at the server as the number of log records varies; each point is the average of five trials. This time determines the number of WayStations a server can handle in the worst case. Server-side time increases to just under 1.1 seconds for 500 log records. In practice we expect the costs to be much smaller. The week-long NFS trace would never have generated a reconciliation with more than 64 log records.

Figure 7 shows the total time for a reconciliation measured at a WayStation as the number of log records varies. This includes the server-side time, RMI overheads, and network costs. The total reconciliation times for the local area, small distance, and medium distance traces do not vary significantly. This means that, at these speeds, bandwidth is not the limiting factor. Profiling of the reconciliation process suggests that RMI—even with



This figure shows the reconciliation times spent at a server as the number of log records varies. These times determine the number of WayStations a server can handle.

Figure 6: Reconciliation Time at Sever



This figure shows the total reconciliation times in seconds as the number of log records varies. These times determine the upper limit on how frequently a WayStation can reconcile with the server. For all sizes, they are much shorter than our default reconciliation period of 15 seconds.

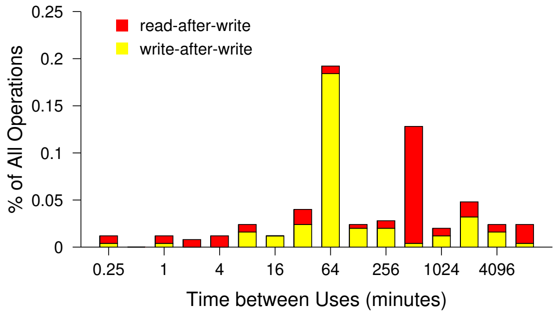
Figure 7: Reconciliation Time at WayStation

our hand-optimized signatures—is the rate-limiting step. However, at 1 Mb/s, the bandwidth of the large distance and intercontinental traces, the bottleneck shifts to networking costs. In any event, the reconciliation times are much shorter than our default reconciliation period of 15 seconds, allowing the WayStation to reconcile more frequently if sharing patterns warrant.

5.4 Consistency

The consistency offered by Fluid Replication depends on two factors: the frequency of reconciliation and the time between uses of a shared object. Section 5.3 quantified the former. In this section, we address the incidence of sharing observed in a real workload.

To determine how often sharing happens and the time between shared references, we examined our week-long NFS client traces. For this analysis, we removed references to user mail spools from our traces. A popular mail



This figure shows the percentage of operations that caused sharing. The x-axis shows the time between uses of a shared objects in minutes, and the y-axis shows the percentage of total operations that exhibited sharing. The top part of bar shows the read-after-write sharing; the bottom part shows the write-after-write. Only 0.01% of all operations caused sharing within 15 seconds. Just over 0.6% of all operations exhibited any form of sharing.

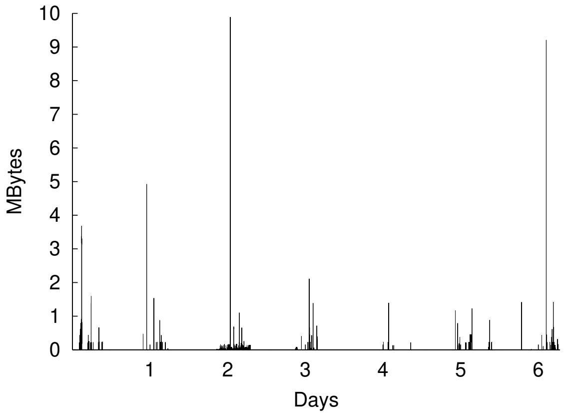
Figure 8: Sharing

client in our environment uses NFS rather than IMAP for mail manipulation. Many users run these clients on more than one machine, despite NFS's lack of consistency guarantees [25], generating spurious shared references. Since we would expect mobile users to use IMAP instead, we excluded these references.

Figure 8 shows the percentage of all references to objects written previously at another replica site. The top part of each bar shows read-after-write sharing, and the bottom part shows write-after-write. As expected, sharing is not common, especially over short periods of time. Only 0.01% of all operations caused sharing within 15 seconds. The total fraction of references that exhibited sharing during the week was just over 0.6% of all operations. Note that these numbers are pessimistic, as we have assumed that each client uses a distinct WayStation. The graph shows some interesting periodic behavior; unfortunately, with network-level traces, we are unable to identify the processes causing it.

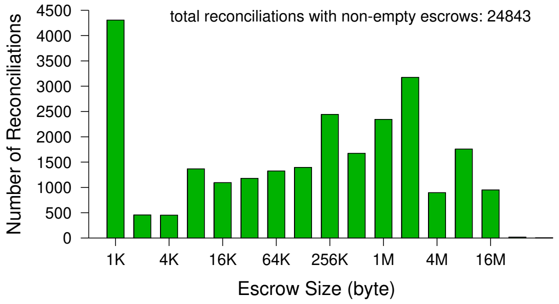
5.5 Availability

Because WayStations do not propagate file contents to the server immediately, a failed WayStation could keep a client from retrieving a needed update. To gauge how likely such a scenario might be, we fed our NFS traces into a Fluid Replication simulator. We augmented the trace with WayStation reconciliations and failure/recovery pairs. Reconciliations were scheduled every 15 seconds from the WayStation's first appearance in the trace. We assume a mean time to failure of 30 days and a mean time to repair of one hour, both exponentially distributed. These parameters were chosen to rep-



This figure shows the size of global escrow. The x-axis shows time in days from the beginning of the NFS trace, and the y-axis shows size of escrow.

Figure 9: Global Escrow Size



This figure shows the distribution of non-empty escrow sizes. The x-axis shows time in days from the beginning of the NFS trace, and the y-axis shows the number of reconciliations resulting in an escrow of that size or less.

Figure 10: Non-Empty Escrow Distribution

resent typical uptimes for carefully administered server machines; they are the same as those used in the xFS study [5]. Note that we did not model server failures; our intent is to explore Fluid Replication's *additional* contribution to failures.

For each trial of the simulation, we varied the random seed controlling failures and repairs. We then counted the number of back-fetches that fail due to the unavailability of a WayStation. We took over five million trials of the experiment in order to provide reasonable confidence intervals for a very small mean.

Recall that during our trace there were 24,957 total requests made of the WayStations. Of those, 243 requests required a back-fetch from one WayStation to another. We first simulated Fluid Replication when updates that had not been overwritten or back-fetched were propagated by the WayStation after 1 hour. Across five million trials, 187,376 requests failed, for an average of 1.46×10^{-6} failures per operation, with a 90% confidence interval of $\pm 8.83 \times 10^{-7}$; this is equivalent to five nines' availability. Expressed in time, we observed an average of 0.037 failures per week, or roughly one failed access every 27 weeks for our client population.

5.6 Costs of Escrow

Our final question concerns the storage costs of escrow. To determine them, we fed the week-long NFS traces into our simulator. Each WayStation in the trace reconciles every 15 seconds from the moment it is first named, as before. After an update has been reconciled, it is marked eligible for escrow. Any subsequent write to the file checks for this mark, and, in its presence, preserves the old version in case it is needed. Updates are removed from escrow as described in Section 3.4.

Figure 9 shows how global escrow size changes with time. The x-axis plots days from the beginning of the trace and the y-axis shows the size of escrow in MB. Global escrow size was almost always zero, and never exceeded 10 MB over the course of the week. The large variance matches the burstiness expected of update activity. Over the lifetime of our trace, WayStations collectively handle over 280 MB; compared to this, escrow requirements are modest.

Figure 10 gives a more detailed picture of escrow sizes over time. This histogram plots reconciliations resulting in non-empty escrows, showing the frequency with which each size was seen. Almost 89% of all non-empty escrows are 2MB or smaller.

6 Future Work

There are three main tasks ahead of us. First, we plan to use network estimation [12] and distance based discovery [19] to automate the WayStation migration and location processes. We have developed a technique that estimates network performance along the path between a client and a server using observations of request/response traffic. This estimator, borrowing techniques from statistical process control, follows underlying shifts in network performance while filtering out observed noise. In effect, the estimator adapts its behavior to prevailing networking conditions, selecting for agility or stability as appropriate. This estimator, combined with a cost-benefit analysis of WayStation migration, guides the search for a new replica site when the client moves too far from the old one.

Our second task is to address the need for trust between clients and WayStations [20]. Before a client will agree to use a WayStation, it must be assured of the privacy and integrity of cached data, and of non-repudiation of updates. Preventing exposure through encryption is straightforward, though managing keys can be subtle. It is also easy to reliably detect unauthorized modifications using cryptographic hashes. However, guaranteeing that a WayStation will correctly forward updates is difficult, if not impossible.

Rather than attempt to prove a WayStation trustworthy a priori, we plan to provide a *receipt* mechanism that enables a client to prove that a WayStation did not properly forward an update. When a client ships a version of a file to the WayStation, it receives a cryptographically-signed receipt for that update. The client can later check whether that version was properly reconciled; lack of reconciliation provides evidence of update repudiation. Receipts can be optimized away much as updates can, but clients must retain the authority to apply such optimizations. Furthermore, clients retain the latest version of any updated file until that version is known to either reside on the server or have been invalidated; the space required to do so is modest [13]. This guarantees that the latest version of a file is known to reside on either a client or the server, in addition to any untrusted WayStation copies. Thus, a WayStation that disappears from the system cannot take with it the only current version of a file, though earlier, escrowed versions may vanish.

Our third goal is to gain more experience with Fluid Replication and its use. While the NFS traces are informative, they do not capture precisely how users will use Fluid Replication. We plan to deploy a server in our department for day-to-day storage requirements, and provide users with WayStations and wireless gateways at home. This will allow client laptops to seamlessly migrate between locales, giving us valuable insights into the system's use.

7 Conclusion

As mobile clients travel, their costs to reach back to home filing services change. To mask these performance problems, current file systems reduce either *safety*, *visibility*, or both. This is a result of conflating safety and visibility into a single mechanism. They have different requirements, and so should be provided through different mechanisms.

Fluid Replication separates the concerns of safety and visibility. While traveling, a mobile client associates itself with a nearby *WayStation* that provides short-term replication services for the client's home file system. Updates are sent to the nearby WayStation for safety, while WayStations and servers frequently exchange knowledge of updates through *reconciliation* to provide visibility. Reconciliation is inexpensive and wait-free. WayStations retain copies of advertised updates in *escrow* until they are irrelevant to other replica sites.

An analysis of traffic in a production NFS server validates our design decisions. A modest reconciliation interval of fifteen seconds limits the rate of stale reads or conflicting writes to 0.01%. Reserving 10 MB for escrow space—a small fraction of disk capacity—is sufficient in the worst case. Measurements of a Fluid Replication prototype show that it isolates clients from most wide-area networking costs. Update traffic is not affected at bandwidths as low as 56 Kb/s or latencies as high as 200 ms. These gains offset increased sharing costs, even for workloads with substantial degrees of sharing. Escrow space requirements are modest: at its peak, it is less than 5% of the total data cached at WayStations. Despite the fact that update propagation is deferred, availability does not suffer. Our traced clients could expect five nines' availability.

References

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[2] G. C. Berry, J. S. Chase, G. A. Cohen, L. P. Cox, and A. Vahdat. Toward automatic state management for dynamic web services. In *Network Storage Symposium*, Seattle, WA, October 1999.

[3] M. Blaze. NFS tracing by passive network monitoring. In *Proceedings of the Winter 1992 USENIX Conference*, pages 333–343, Berkeley, CA, January 1992.

[4] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *2000 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 34–43, Santa Clara, CA, June 2000.

[5] M. D. Dahlin, C. J. Mather, R. Y. Wang, T. E. Anderson, and D. A. Patterson. A quantitative analysis of cache policies for scalable network file systems. In *1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 150–160, Nashville, TN, May 1994.

[6] M. R. Ebling. *Translucent Cache Management for Mobile Computing*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, March 1998.

[7] S. Floyd, V. Jacobson, C. G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, December 1997.

[8] B. Gronvall, A. Westernlund, and S. Pink. The design of a multicast-based distributed file system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 251–264, New Orleans, LA, February 1999.

[9] J. S. Heidemann, T. W. Page, R. G. Guy, G. J. Popek, J.-F. Paris, and H. Garcia-Molina. Primarily disconnected operation: Experience with Ficus. In *Proceedings of the Second Workshop on the Management of Replicated Data*, pages 2–5, November 1992.

[10] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[11] P. J. Keleher and U. Cetintemel. Consistency management in Deno. *Mobile Networks and Applications*, 5(4):299–309, 2000.

[12] M. Kim and B. D. Noble. Mobile network estimation. In *7th ACM Conference on Mobile Computing and Networking*, July 2001. to appear.

[13] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[14] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Association Summer Conference Proceedings*, pages 238–247, Atlanta, GA, June 1986.

[15] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201, November 2000.

[16] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proceedings of the 1995 USENIX Technical Conference*, pages 95–106, New Orleans, LA, January 1995.

[17] H. Lei and D. Duchamp. An analytical approach to file prefetching. In *Proceedings of USENIX 1997 Annual Technical Conference*, pages 275–288, Anaheim, CA, January 1997.

[18] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Fifteenth ACM Symposium on Operating Systems Principles*, pages 143–155, Copper Mountain Resort, CO, December 1995.

[19] B. Noble, B. Fleis, and M. Kim. A case for Fluid Replication. In *Network Storage Symposium*, Seattle, WA, October 1999.

[20] B. D. Noble, B. Fleis, and L. P. Cox. Deferring trust in Fluid Replication. In *9th ACM SIGOPS European Workshop*, Kolding, Denmark, September 2000.

[21] B. D. Noble and M. Satyanarayanan. An empirical study of a highly available file system. In *1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 138–149, Nashville, TN, May 1994.

[22] B. D. Noble, M. Satyanarayanan, G. T. Nguyen, and R. H. Katz. Trace-based mobile network emulation. In *Proceedings of the ACM SIGCOMM 97 Conference*, pages 51–61, Cannes, France, September 1997.

[23] T. W. Page, Jr., R. G. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek. Perspectives on optimistically replicated, peer-to-peer filing. *Software — Practice and Experience*, 28(2):155–180, February 1998.

[24] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *15th ACM Symposium on Operating Systems Principles*, pages 79–95, Copper Mountain Resort, CO, December 1995.

[25] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *Proceedings of the Summer USENIX Conference*, pages 137–152, Boston, MA, June 1994.

[26] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 288–301, Saint Malo, France, October 1997.

[27] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[28] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1):33–57, 1994. Corrigendum: 12(2):165–72, 1994.

[29] D. C. Steere, J. J. Kistler, and M. Satyanarayanan. Efficient user-level file cache management on the Sun vnode interface. In *Proceedings of the 1990 Summer USENIX Conference*, pages 325–331, Anaheim, CA, 1990.

[30] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149, Austin, TX, September 1994.

[31] R. Wang and T. E. Anderson. xFS: A wide area mass storage file system. In *Proceedings of the IEEE 4th Workshop on Workstation Operating Systems*, pages 71–78, Napa, CA, October 1993.

[32] M. Welsh and D. Culler. Jaguar: Enabling efficient communication and I/O in Java. *Concurrency: Practice and Experience*, 12(7):519–538, 2000.

[33] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 75–84, San Diego, CA, October 2000.