SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 28(2), 155–180 (FEBRUARY 1998)

Perspectives on Optimistically Replicated, Peer-to-Peer Filing

T. W. PAGE, JR.,* R. G. GUY, J. S. HEIDEMANN,† D. H. RATNER,
P. L. REIHER, A. GOEL,‡ G. H. KUENNING, G. J. POPEK§

Department of Computer Science, UCLA, Los Angeles, California 90095–1596 USA
E-mail:{page,johnh,rguy,ratner,reiher,ashvin,geoff,popek}@fmg.cs.ucla.edu

SUMMARY

This research proposes and tests an approach to engineering distributed file systems that are aimed at wide-scale, Internet-based use. The premise is that replication is essential to deliver performance and availability, yet the traditional conservative replica consistency algorithms do not scale to this environment.

Our Ficus replicated file system uses a *single-copy availability*, optimistic update policy with reconciliation algorithms that reliably detect concurrent updates and automatically restore the consistency of directory replicas. The system uses the peer-to-peer model in which all machines are architectural equals but still permits configuration in a client-server arrangement where appropriate. Ficus has been used for six years at several geographically scattered installations.

This paper details and evaluates the use of optimistic replica consistency, automatic update conflict detection and repair, the peer-to-peer (as opposed to client-server) interaction model, and the stackable file system architecture in the design and construction of Ficus. The paper concludes with a number of lessons learned from the experience of designing, building, measuring, and living with an optimistically replicated file system.

KEY WORDS Optimistic concurrency control Software replication Distributed file system File system structure Disconnected operation

DRAFT COPY: This paper is a draft copy of the version to appear in SP&E Vol. 28(2), pages 155-180 (February 1998). Readers wishing the final version should see that publication.

INTRODUCTION

Despite the publicity generated by the World-Wide Web, the primary computer network-based tool for geographically distributed cooperative work is still electronic mail. Bob in Atlanta might write a new piece of a joint document, and then e-mail it to his

* T. Page is now at the Department of Computer and Information Science, The Ohio State University, Columbus, Ohio, 43210–1277. E-mail: page@cis.ohio-state.edu.
† J. Heidemann is now at the USC Information Sciences Institute, 4676 Admirality Way, Marina Del Rey, California, 90292. E-mail: johnh@isi.edu.
‡ A. Goel is now with the Oregon Graduate Institute. E-mail: ashvin@cse.ogi.edu.
§ G. Popek is also Chief Technical Officer at PLATINUM *technology, inc.*, Inglewood, CA 90301.

CCC 0038–0644/98/020155–26

Received 1 December 1996
Revised 1 July 1997

colleague Alice in Seattle, who then integrates it with her copy. Perhaps she makes some changes and mails them back. Alice and Bob may elect to impose some protocol on themselves to avoid conflicting updates. If the other party could not be contacted, however, chances are they would be *optimistic* and relax their protocol; they would update the document anyway, making a mental note that those changes may need later re-integration. In many situations, the need to "get the work done" outweighs the desire to avoid conflicting updates.

What is actually going on in this example is a form of data replication without any system support. This familiar mode of operation often works out relatively well in practice. Concurrent changes are seldom a problem, and performance is excellent as data is always local (whether composing, receiving, or modifying, e-mail is always local). However, a real distributed replicated file system could surely provide far better support. For example, relying on "mental notes" to propagate changes and check for conflicts risks losing updates and often results in the creation of differing versions of a document whose relationship must be reconstructed from memory. Neither does this informal approach scale in the number of file users. This vignette illustrates the value of an optimistic replicated file system with an automated service to detect and repair conflicts. Ficus represents our efforts to address the software engineering of such a system. This paper reports our experiences designing, implementing, measuring and using an optimistic replicated file system.

Goals of the research

The purpose of this research is to test the practicality of optimistic filing. The following frames our intent in building a replicated file system:

- enable a shared, network-transparent file hierarchy which scales both to large geographic distances and very large numbers of files;
- provide high availability and performance;
- demonstrate a design and implementation capable of providing realistic, general-purpose use, including permitting all machines to operate as peers rather than limiting them to restricted functionality;
- produce a real, usable system that others can run and build upon, not simply a proof of concept;
- test the use of the stackable layers approach to structuring software for a large filing project;
- avoid building large portions of file systems to which we have little to contribute;
- retain as much compatibility with the existing environment as possible.

Optimistic replicated filing

Ficus is a general purpose replicated file system intended to facilitate distributed collaboration in a highly reliable and scalable fashion. It has been implemented as an addition to UNIX, although little of the architecture is specific to that system. Replication offers the potential for improved performance by locating a copy of data "near" where it is needed, even when it is needed simultaneously at geographically dispersed locations. Replication is critical to reliability in networks where site and communications failures are the rule rather than the exception. Mobile computing represents an important example of this situation.

A replicated file system must have both mechanism and policy to keep multiple replicas consistent in the face of updates. A *pessimistic* approach prevents inconsistency by restricting updates that could lead to a conflict. The optimistic approach, on the other hand, takes the view that it is expensive in terms of performance to obtain locks and unacceptably costly in terms of availability to restrict updates, particularly when conflicts are rare anyway. Hence optimistic policies allow conflicts to occur, but detect and deal with them afterwards. While there are certainly environments and applications which call for the pessimistic policy, we argue that the optimistic approach is essential in the general purpose distributed setting.

Unlike most replicated file and database systems, Ficus allows updates so long as at least one replica of a data object is available; this is termed *single-copy availability*. Experience with Ficus, data reported in this paper, and previous results in the literature (summarized later) indicate that conflicting updates seldom occur in practice, for many classes of files. When conflicts do occur, Ficus reliably detects them. For directories and replica location information, the system uses its knowledge of the semantics of updates to resolve most conflicts automatically. If the system does not understand the update semantics, as is the case with an arbitrary file, it reports the conflict to a *resolver* which understands its semantics; if no such software is known, the system informs the file owner (via e-mail), who resolves it manually. In practice, conflict resolution has not been difficult for users, and there is an expanding set of automatic resolvers for known file types.

The Ficus algorithms are based on the view that each machine, including workstations, portable computers and servers, should be empowered with full function so far as replication, file service, and reconciliation are concerned. In this sense, all machines are peers. We believe that peer models are particularly important with portable and geographically distributed computing. For example, if several professionals travel with their notebook computers, at their destination they may connect their machines and have the full benefits of optimistic replication without contacting a "home server." This situation applies to inherently mobile workers, the military, as well as to any situation where remote access can be slow or periodically unavailable. Thus a decentralized peer architecture provides a more robust operational solution than a more centralized one in which a server must be present to share or reconcile. However, it demands more sophisticated algorithms to provide a normally transparent level of service.

The robustness presented to the user by single-copy availability is also apparent in the underlying distributed mechanisms that propagate update activity to all replicas and resolve conflicts. These mechanisms are designed around pairwise "gossip" strategies that idempotently "pull" replica (and meta-data) state from a remote replica, and then merge, update, or resolve as appropriate with the local state, possibly resulting in a new local replica state. Such "lazy" strategies place minimal requirements on the communications environment for fundamental correctness:

- no more than two parties are ever required to be alive and communicating at the same time;
- "all pairs" direct communication is never required, although information must be able to flow indirectly among any pair of nodes in finite time;
- correctness relies only on a "pull" of information, although a "push" may provide

significantly improved performance;*
- network partitions neither jeopardize correctness, nor unduly delay progress within a partition;
- a permanently incommunicado node (either dead or forever partitioned and administratively declared such) does not delay progress;
- messages may be lost, delayed a large (but finite) time, duplicated or delivered out of order.

The family of algorithms that lies at the heart of Ficus replication typically operates in two phases, using bit or scalar vectors to record progress. Their local space requirements are always linear or better in the number of replicas (and thus, quadratic or better overall). The global message complexity is quadratic in the worst case of pathological network partitioning and linear in the best case (a well-connected LAN). Careful piggybacking of related algorithm executions is employed to keep constant factors small, and all ancillary algorithm state is discarded upon algorithm termination.

The combination of single-copy availability, lazy strategies, and two-phase algorithms yields a robust, fault-tolerant, and highly available filing service. Further, the various overhead costs of managing replicas are largely incurred in the background, and not in-line to user file activity: update propagation, reconciliation, and conflict resolution are all normally asynchronous with respect to routine file access.

The next section introduces our approach to optimistic replica management.

REPLICATED DATA CONSISTENCY

The basic operation of Ficus is as follows. When a file is opened (for read or write), Ficus chooses a replica to service subsequent file requests. Upon receiving an application's file update request, Ficus applies the update to the chosen replica. Update notification messages then inform the other replicas of the new data, and those sites pull over the changes asynchronously. The update notification messages may be lost. The other replica storage sites may be down or otherwise inaccessible; if they miss being informed about the update, replicas become temporarily inconsistent.

In order to tolerate the occasional missed update propagation, Ficus uses *reconciliation.* On behalf of each file replica, the file system periodically contacts a peer replica to enquire about missed activity. Each replica has an associated version vector,[1] which summarizes the complete set of updates known to that replica. At a reconciliation enquiry, the local file system compares the version vector of the local replica to that of the remote. If the remote replica is strictly newer than the local, its version vector will dominate and the remote data and version vector can be pulled over to move the system towards consistency. If two version vectors are not equal and neither dominates, an *update/update conflict* is detected and a *resolver* invoked. since propagating either version over the other would cause data from one of the updates to be lost.

The single-copy availability update policy for files and directories rules out guaranteeing one-copy serializability, the traditional definition of correct operation inherited from distributed database theory. However, even in the local case UNIX file systems do not provide transactional semantics so abandoning one-copy serializability for the dis-

* For example, Ficus uses an "update notification" daemon (a push) to tell other replicas asynchronously of a new file version This typically results in a much faster propagation than relying on periodic volume-wide file reconciliation (pulls).

tributed case is acceptable. Ficus instead provides a *no lost updates* guarantee, which we argue is often preferred. No lost updates guarantees that data in which a user may still have interest will never be inadvertently discarded as a result of the optimistic update synchronization policy (e.g., via an over-write due to concurrent updates or removal of new data due to a concurrent update and remove operation).

Volumes (as in AFS) are sub-trees of the file naming hierarchy with a granularity smaller than a conventional file system but larger than (or equal to) a single directory. The reconciliation daemons, while logically reconciling individual files and directories, are actually organized at the volume level. *Selective replication control* within Ficus allows each volume replica to physically store an essentially arbitrary subset of the entire volume. That is, the set of sites which store a volume replica forms the maximal (but not minimal) set of physical storage sites for any given file within that volume. The selective replication mechanism additionally preserves transparent remote access to those objects within the volume that are not locally stored. Due to selective replication, two communicating volume replicas may not store exactly the same set of objects. The reconciliation algorithms dynamically adapt to the current replication patterns and adjust communication topologies to ensure consistency in such cases.[2]

A conflict, once detected, must be resolved by software (or humans) which understand the semantics of the data and the updates which caused the conflict. If the file in conflict is "owned" by an application which has registered a resolver for that file type with the file system, the system invokes the type specific resolver. If no resolver is known, the system reports the file conflict to the owner (via e-mail).

Directories

Unlike user files, directories are managed solely by the file system, and have simple, well-understood semantics. Hence the file system itself supplies the resolver for directories.

A directory contains a set of entries, each of which associates a name with a pointer to a file or subdirectory. Ficus adds a unique identifier to this pair for reasons described in the next section. The only modification operations applicable to a directory are adding new entries and deleting existing ones. Thus it is feasible to take two directory replicas that have been updated independently (normally an update/update conflict), merge the changes and automatically form a single correct version that reflects the effects of all updates applied to both replicas.

A correct reconciliation algorithm must compare the entries in the two directory replicas, identify newly created entries in one replica that have yet to propagate to the other, and also identify newly deleted entries that still appear in the other replica. Missing operations must be applied to each replica to rectify these inconsistencies. Replaying all operations at each replica is not sufficient because the connection history of sites is arbitrary, rendering the decision of what logs to play at what sites in what order distinctly non-trivial,[1] and because reconciliation must be incremental (and hence not atomic) for the volume to be highly accessible. Note that while results may not be serializable (for example, concurrent deletes to a single replica results in an error while independent, concurrent deletes to different replicas is acceptable), they are acceptable given the semantics of directory updates.

Several issues must be addressed to reconcile replicas accurately; we next discuss these issues and their treatment in Ficus.

Insert/delete ambiguity

Consider two copies of a directory: the first has an entry for file F and the second does not. Has the entry for F been newly created and thus should propagate to the second directory replica, or has it been deleted in the second replica and thus should be deleted in the first as well? This is the *insert/delete ambiguity*.[3]

Ficus addresses this difficulty by initially logically deleting a directory entry (changing a flag from `Live` to `Deleted`) rather than physically removing it. Logically deleted entries are ignored, except for purposes of reconciliation. Deleted entries can now be distinguished from inserted ones by the delete mark. This resolves the insert/delete ambiguity at the cost of creating a garbage-collection problem: when is it safe to discard a logically deleted directory entry?

A correct solution must retain a logically deleted entry at least until all directory replicas know that the entry is deleted. Otherwise, a replica which discards a logically deleted entry might later reconcile with a replica whose directory entry had not yet been marked deleted, and conclude that the entry is new and should be propagated, thereby reintroducing the insert/delete ambiguity.

Somewhat surprisingly, simply ensuring "all replicas know" is not sufficient. Suppose that when a directory replica learns (via reconciliation) that the entry of interest is marked logically deleted in all replicas, it discards its own logically deleted entry. If this directory replica later reconciles with another replica (which doesn't yet know that all entry replicas have been marked logically deleted), the question arises, is the logically deleted entry to be propagated to the "ignorant" replica? The insert/delete ambiguity emerges yet again. Successfully eliminating insert/delete ambiguities requires that prior to discarding a logically deleted directory entry, a directory replica must not only know that all replicas are marked deleted, but further must know that all other replicas are also aware of this fact.

Hence, Ficus employs a garbage-collection algorithm to detect the "all replicas know that all replicas know" condition.[4] This algorithm has two important properties: monotonicity and low-cost indirect communication. Monotonicity insures that the algorithm always makes progress, guaranteeing eventual termination and preventing repeated sequences of deallocation/allocation of deleted directory entries. Indirect communication is necessary in networks in which all-pairs connectivity is not guaranteed. Information needed for the algorithm to progress to termination spreads between all data-storing replicas in a gossip-like fashion. Volume replicas that do not store a given object (via the selective replication mechanism) need not participate in its garbage collection.

The garbage-collection algorithm proceeds in two phases. Phase one compiles the list of replicas that know the entry is deleted. Phase one ends and phase two begins at a replica when its list is complete, i.e., includes all replicas of the entry. Phase two compiles the list of replicas that are known to have finished phase one, and concludes when this second list contains all replicas. When phase two completes at a node, that node knows that all replicas know that all replicas have marked the entry deleted, and therefore it is safe to garbage collect the deleted entry. Any other node that ever asks about the status of that entry will get the response "entry unknown" from which it can correctly conclude that garbage collection has finished at that site, and hence can finish at the inquirer as well. There is no ambiguity since the inquirer knows, by virtue of being in phase two, that the other site once knew about the entry and its deletion. Further discussion of these algorithms is available elsewhere.[2, 4]

Global inaccessibility

In UNIX-like file systems, the remove operation does not remove a file, only a *name* for a file. Freeing the storage for a file occurs only as a side-effect of removing its last name; when it is no longer accessible it may be garbage collected.[5] However, with optimistic replication, local inaccessibility does not imply global inaccessibility. Other names may exist in remote replicas, names that have yet to propagate to the local site. The system must not garbage collect file data until it determines that no new name exists for the file anywhere, lest new updates to (or even the last copy of) a file be lost. An important task of reconciliation is therefore to provide an acceptable solution to the problem of garbage collection of a partially replicated, distributed graph structure subject to concurrent changes.

Global zero name count is a stable state, as new names can only be created for objects that have at least one name, and multiple alternative stable state detection algorithms may be employed depending on the network communications assumptions.[6] The specific algorithm used for our environment is analogous to and runs in parallel with the above two-phase algorithm.[4] Briefly, we detect global inaccessibility with another two-phased gossip algorithm. Phase one compiles the set of sites that believe the file to have no names. Phase two permits garbage collection by determining that all sites know that all sites know the file has no more names. Here too, only the data-storing replicas need participate in the algorithm.

Remove/update conflicts

A remove/update conflict occurs when the last name of a file is removed in one partition while the file is concurrently updated in another. Remove/update conflicts are detected by recording the version vector of a file whose name is removed and checking that the removed version is not dominated by, or in conflict with, any other replica. Again, this runs in parallel with the two-phase garbage collection.

There are two policies one might adopt when a remove and an update occur concurrently: favor the remove and discard the update; or prefer the update and ignore the name deletion. In accordance with our no lost updates policy, we must not discard the new data, but neither is it acceptable for a deleted name to reappear. Our policy is to remove the file name, but to save the data in an *orphanage* (a directory for otherwise unnamed files similar to UNIX's `lost+found`) and inform the owner.

Name conflicts

Reconciliation must detect when two entries have been created concurrently which have the same name pointing to different objects. This violates normal directory semantics, which specify that names be unique within a directory. Ficus directory reconciliation detects name conflicts, gives each conflicting entry a disambiguating suffix, and invokes a resolver or informs the files' owner(s). Details of the Ficus resolver architecture are found elsewhere.[7]

Algorithm discussion

These two-phase algorithms are quite different from conventional two-phase commit protocols. In two-phase commit, forward progress is denied whenever a few sites cannot communicate. Here, there is no central coordinator, as all participants are peers. Both phases of the algorithms can be in progress at the same time, since one site may become aware that all sites know of a deletion well before other sites do so. No underlying connectivity is assumed other than the requirement that information can propagate between any two sites in finite time.

A minimum of $o(3n)$ messages is required for reconciliation to complete if information travels around a virtual ring of the n storage sites. Three trips around the ring assure that every site has heard the state of every other site at least twice, allowing both phases to complete everywhere.

We have described a simplified version of the reconciliation algorithms here. In practice, a number of optimizations are quite beneficial, especially in environments where network bandwidth or latency are a consideration. For example, this basic approach adapts well to replicas connected only by a 28.8kb/s modem.[8]

FICUS ARCHITECTURE

The Ficus replication services are added to the UNIX kernel using an extensible layered VFS interface.[9, 10] A file system layer is a software module that conforms to the layer interface with respect to calls to it from above and calls to the layer below. Each layer in a file system stack adds a specific piece of functionality.

Ficus consists primarily of three of layers: the *replica selection layer*, the *logical layer*, and the *physical layer*. The replica selection layer implements a data consistency policy for clients, providing the abstraction of a single-copy, highly available file built from the multiple replicas available to it. The logical layer coordinates updates across multiple replicas and provides other functionality common to different replica selection policies. The physical layer implements the extended attributes needed for replication and maps file storage to a standard UNIX file system (UFS). These layers use a *transport layer* to map the layer interface across a network, and are stacked above a unmodified, standard file system for persistent storage.

Figure 1 shows a layer configuration for a file with two replicas stored on sites 1 and 3. Each replica has a physical layer supporting it. Each client has a logical layer. Logical layers are connected directly to local physical layers, and to remote replicas via transport layers. Above each logical layer is a replica selection layer, the default selection layer for sites 1 and 2 and an alternate selection layer for site 3.

The replica selection layer

Optimistic concurrency control and lazy update propagation can yield a number of file versions, including the possibility of conflicting versions. The volatility and scale of a large geographically distributed environment can make it infeasible even to determine the range of accessible versions. A separate replica selection layer allows different clients to have appropriate version selection policies. (Early Ficus implementations merged the functionality of the replica selection layer into the logical layer.)

The default replica selection layer provides only a very simple policy, choosing a local replica if one exists, and falling back to a randomly chosen replica otherwise. Future requests are also directed to that replica, insuring that the client sees consistent data unless and until that replica becomes inaccessible.

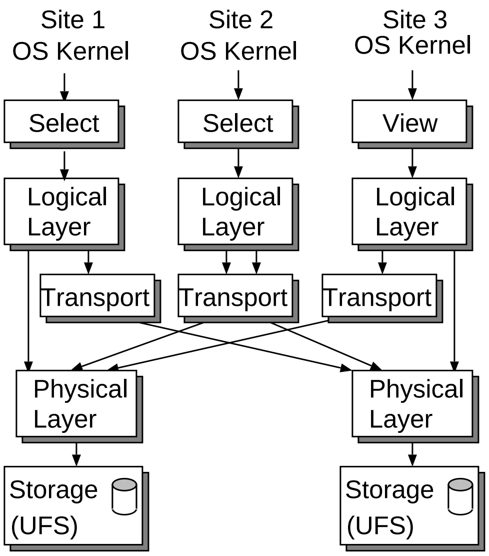

Figure 1. A typical Ficus layer configuration with two storage sites and a third site mounting both replicas.

While this default is suitable for general use, some situations require stronger consistency guarantees. An alternate model, *view consistency* allows each client "entity" to access only versions that are at least as new as what that client has previously accessed. This provides a conservative consistency model to each client entity while allowing optimistic consistency across entities. Entities can vary in size (a process, a user's processes, a machine, or a group of machines); our sample implementation uses machine-granularity entities. The cost of the stronger guarantees provided by view consistency is in the form of additional record keeping (what versions of files has the entity accessed previously) and some performance overhead. A detailed analysis of view consistency, including garbage collection and its effects on performance and availability can be found elsewhere.[11]

Replica selection layers also allow knowledgeable clients explicit access to specific replicas. This service is useful, for example, in programs which repair conflicts.

The logical layer

The logical layer cooperates with the replica selection layer to provide the illusion that each file is highly available and has single-copy semantics. The logical layer performs update propagation and other services which are common to all replica selection layers.

To inform other replicas of an update, a logical layer places a summary of the update on an outgoing update notification queue, and then returns control to the client.* A client is assured that an update has been applied to at least one replica, which is sufficient for the no lost updates guarantee. A daemon periodically services the queue, sending out notification messages containing the version vector of the new version, and a hint about the site known to have stored that version. Notification is a best-effort, one-shot attempt; inaccessible replicas are not guaranteed to receive an update notification later. Optimism releases the system from the burden of ensuring that an update notification is successfully delivered and processed by the receiver. If any replica fails to receive or apply the update for whatever reason, it will eventually learn of the update via reconciliation. This is an example of how optimism permits considerable simplification throughout the architecture. As so much of a system's actual implementation is often concerned with error handling, this improvement is considerable.

On receipt of an update notification, a site places the information contained in the notification in a queue which is serviced periodically by another kernel daemon. That thread, operating at the logical layer, pulls the new data from a more up-to-date site, making sure that this inward propagation occurs atomically. A commit mechanism, using a shadow copy, ensures that a replica's version vector always reflects the replica's data, even in the face of crashes and concurrent operation during propagation.

The physical and persistent storage layers

The physical layer performs three main functions: storing extended attribute information connected with each replica, creating and managing a UNIX file to store the data for each file replica, and implementing the additional naming hierarchy semantics required by Ficus.

The first function of the physical layer is to manage the extended attribute information that must be stored with each replica. Note that this information is not replicated; it applies to each replica individually. The physical layer stores extended attributes in a look-aside "auxiliary" file. There is a single auxiliary file per directory, with a record in it for each file in that directory, an organization which clusters information to take advantage of the expected locality of reference between files in the same directory. When a file is accessed, the data page with its auxiliary information is likely to be cached already and not require an additional disk I/O. In keeping with the layered architecture we have recently experimented with an extended attributes layer that may replace this mechanism.

The second role of the physical layer, that of storing file data in the underlying file system, consists primarily of locating the correct UNIX file, and forwarding the various read, write, create, etc. operations to that file. From the point of view of the logical layer, a file replica is uniquely named by a ⟨*file-ID, replica-ID*⟩ pair. The underlying UFS uniquely identifies files by inode number within a file system. So the physical layer must map ⟨*file-ID, replica-ID*⟩ tuples to inodes in a UNIX file system. In the case of selective replication, when an object within the volume is not stored locally, the physical layer instead provides probable remote locations for the object in question back to the logical layer. Ideally we could bypass UNIX file naming and access these

* Since a file written once tends to be written again very shortly,[12] placing the update on a queue gives the potential to batch the update notifications for many updates with a single message, reducing the number of messages that synchronous notification would require. This batching optimization would be beneficial but has not yet been implemented.

files directly via references to their inodes. Although we hope that future layers will provide this interface, currently we map ⟨*file-ID, replica-ID*⟩ to a two-level directory structure.

The third function of the physical layer is to implement richer naming semantics than offered by standard UNIX directories. To permit directory renames during a network partition, the Ficus name space must relax the traditional strict tree of the UNIX name space, allowing instead a more general graph. Further, the directory reconciliation mechanisms require keeping some additional state information with each directory entry. This additional richness inherent in the Ficus model requires that we implement a full name storage mechanism in the physical layer with the additional capabilities that are needed.

Volume location and autografting

In a distributed file system that spans the Internet, there may be hundreds of thousands or millions of volumes to which one may wish transparent access whenever desired. Any one machine will only access a very small fraction of the available volumes, yet one cannot predict in advance which volumes will be needed. Therefore, Ficus, like Sprite[13] and AFS[14] locates and "grafts" (mounts) volume replicas on demand, rather than *a priori*, and periodically ungrafts remote volumes that are unused for some time, permitting access to an enormous virtual name space while consuming only minimal local resources. Ficus differs, however, in how it manages the data necessary to locate remote volumes.

Volume location data is critical to availability; the data in a file system is of little use if it cannot be named. To keep the site storing volume location data from representing a single point of failure that could render the entire subtree beneath it inaccessible, the location information must be replicated. Further, it is essential that these replicas be managed with an optimistic consistency policy. While updating the information in a graft point is a relatively rare event, it is generally quite important when it does occur. Graft points are updated only when volume replicas are added, deleted, or moved to another host. The importance of updating a graft point may be at its greatest precisely when the system is unstable or partitioned; perhaps the reason for updating the graft point is to add an additional replica of a volume when, due to instability, only a single replica remains accessible; this update must be permitted even though it cannot immediately propagate to all replicas of the graft point. Hence it is not reasonable to require that all, or even a majority, of the replicas of the graft point be accessible for an update to be permitted.

Unlike AFS, a separate volume location database is not necessary in Ficus. Instead, we exploit the same optimistic replication and reconciliation mechanism that manages the directory name mapping function. The format of a graft point is compatible with that of a directory, with a single bit indicating that it contains volume location information. Like directory entries, volumes may be moved, created or deleted, so long as any replica of the graft point and volume is accessible in the partition (single-copy availability). Without building any additional mechanism, updates are propagated to accessible replicas and all conflicting updates are automatically resolved, providing name transparency and high update availability while scaling to very large networks.[15]

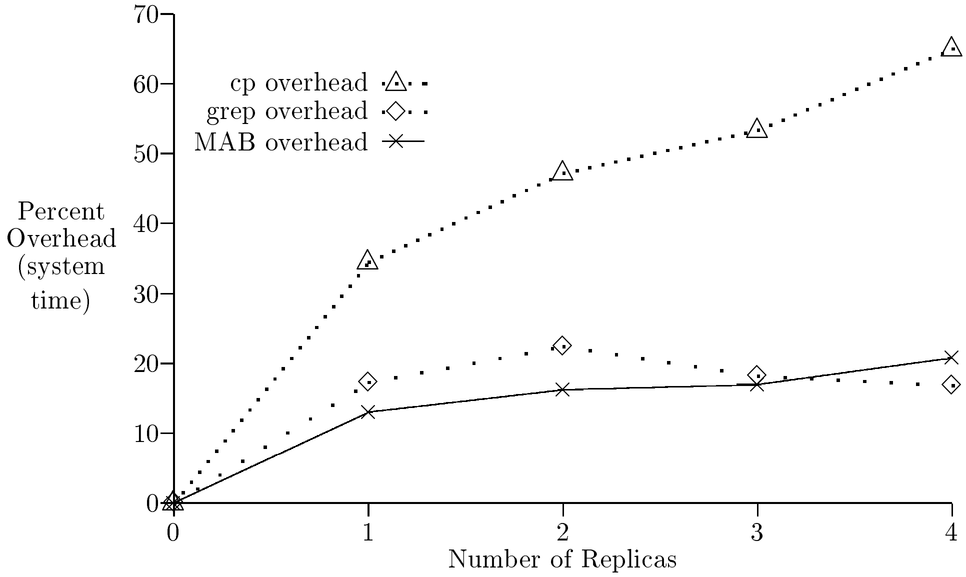

Figure 2. System time overhead for three benchmarks as the number of replicas vary from zero (un-replicated UNIX*) to four replicas under Ficus.*

EVALUATION

This section evaluates Ficus from three perspectives. First, we examine performance in various environments. We next reflect on the appropriateness of the choice of optimistic replica management. Finally, we consider broader lessons learned from our experiences developing and using Ficus.

Performance evaluation

We want to answer the question, "What is the cost of replication as provided by Ficus?" This evaluation employs three benchmarks which are intended to capture the range of operations on a file system.

The first test is a recursive search (`grep`) which reads every file in a large subtree. The tree used is the `/usr/include` hierarchy, which on our system contains 4.2 MB of data and 1191 files, of which 60 are directories. To the extent that one believes that the frequency of read operations greatly dominates writes in a typical filing environment, read performance is a critical measure of file system performance. The second benchmark is a recursive copy (`cp`) over NFS of a large subtree, from a UFS file system into a Ficus volume. This benchmark is chosen to show a worst case performance for Ficus, since creating files is the highest overhead operation. Finally, we use the modified Andrew Benchmark (MAB)[14, 16] to model a normal mix of filing operations, and hence to represent performance in actual use.

Figures 2 and 3 display the system time overhead and elapsed time overhead re-

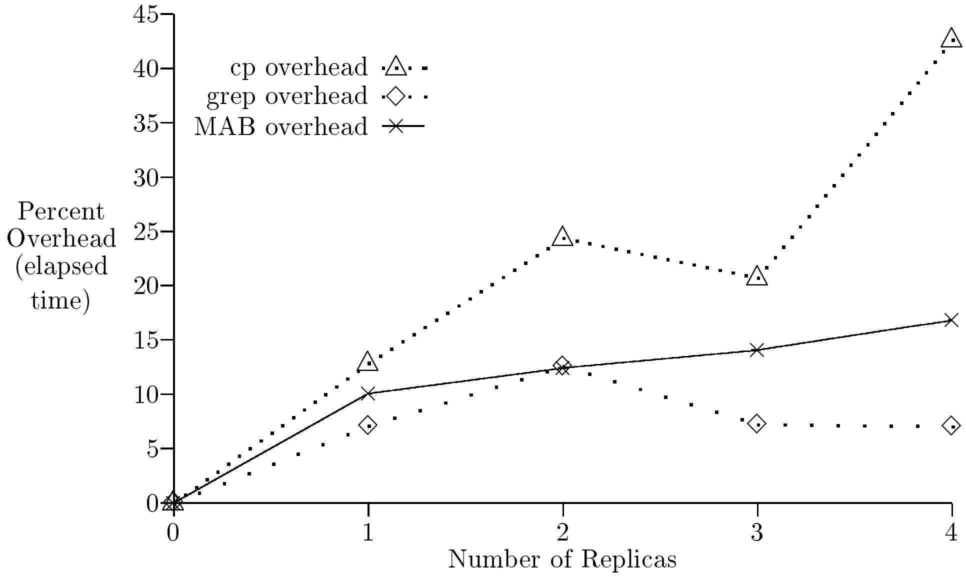

Figure 3. Elapsed time overhead for the same three benchmarks.

spectively for the three benchmarks, as the number of replicas varies from zero to four (within a local area network). The 0-replica case represents standard SunOS without Ficus. One replica means that the files are stored in Ficus, but there is only a single copy. Thus the difference between zero and one replica represents the overhead of running Ficus without any of the benefits of additional replicas. The results are normalized so that the difference between the mean time for each benchmark and the mean time for the same benchmark on SunOS without Ficus is expressed as a percentage of the SunOS-only benchmark. All files were replicated in all volume replicas (no selective replication was used). All measurements were conducted on Sun 3/60s with 8 MB of RAM connected by a 10 Mb Ethernet running a version of Ficus based on SunOS 4.1.1. Each benchmark was repeated at least seven times for each number of replicas; Table I shows the unnormalized means (in seconds) and the 95% confidence intervals (expressed as the width on either side of the mean).

Discussion

The shapes of the curves for both elapsed and system time are as one would expect. The `grep` benchmark is read-only, and so its performance is independent of the number of replicas. It shows a constant 10-12% overhead compared to UNIX.

The `cp` benchmark has a workload that is heavily dependent on the number of replicas, as the site performing the copy must do the extra work both of informing each remote replica of each file creation, and of serving the requests from the remote replicas to propagate the data. Clearly, the performance for this benchmark could be

Table I. Unnormalized benchmark data corresponding to Figures 2 and 3

		UFS		1 replica		2 replicas		3 replicas		4 replicas	
		mean	conf.	mean	conf.	mean	conf.	mean	conf.	mean	conf.
cp	elap.	178.9	15.4	201.9	17.3	222.4	32.3	216.0	14.3	255.1	14.0
cp	sys.	22.8	0.5	30.6	0.3	33.5	1.2	34.9	1.6	37.5	0.5
grep	elap.	59.8	0.7	64.1	0.7	67.4	7.9	64.2	0.7	64.0	0.6
grep	sys.	18.9	0.2	22.2	0.3	23.1	1.3	22.3	0.2	22.1	0.5
MAB	elap.	150.9	2.3	166.1	3.5	169.6	1.9	172.1	5.4	176.3	2.6
MAB	sys.	36.5	0.3	41.2	0.4	42.4	0.5	42.6	0.9	44.1	0.5

improved by being even more lazy about propagation, or by propagating initially to only a few sites, and allowing remaining propagations to get the data from any of several secondary sites. However, since the elapsed time overhead remains below 25% through three replicas, there has been little motivation in practice to implement these optimizations.*

The MAB curve shows a slight upward slope, reflecting the cost of additional replicas in the update portions of the benchmark. The overhead compared to UNIX varies from approximately 10% to 25%. This leads us to conclude that the overhead for a normal mix of filing operations is quite reasonable.

These benchmarks do not tell the entire story. Because of the lack of an inode interface to files in the underlying UFS, the physical layer maps files to the UFS using an extra layer of UFS directories. Unfortunately, the current implementation of this mapping exhibits significant internal fragmentation, leading to a 100–200% overhead in benchmarks such as a recursive `stat`. This overhead does not represent a cost of replication, but rather of the design decision to build Ficus on top of an unmodified UFS. Were we to follow AFS and add an inode-level interface to our underlying UNIX, allowing the removal of the extra layer of mapping, we expect the attendant I/O overhead would be largely eliminated.

Another early design decision which also resulted from the desire to avoid changing the underlying UFS has further negative performance implications. The method by which Ficus maintains extended file attributes (such as version vectors) causes an additional file open to access the extended attribute data (unless the attribute data is found in the cache). Better extended attribute maintenance is clearly possible.

Cost of hosting a replica

The measurements described in this section are aimed at answering the question, "What does it cost to host a replica?" The components of this cost include the storage (media) cost, the cost of update propagation, and the cost of running the reconciliation daemons. Ficus' optimistic approach to consistency pushes some of the work of achieving consistency into the background in the form of asynchronous update propagation and reconciliation. Update propagation is the best-effort approach to update other replicas soon after a file has been updated, while reconciliation is a periodic

* We have rarely found it necessary to store more than three replicas of frequently updated volumes. Widely replicated volumes tend to be read-mostly.

Table II. Elapsed time for rcp and MAB benchmarks with interference

		no	# interfering		
		interference	1	2	3
rcp	mean	266	459	442	441
	ratio*	n/a	1.73	1.66	1.66
	confidence† (±)	18.1	34.6	37.4	13.8
MAB	mean	151	188	244	261
	ratio	n/a	1.25	1.62	173
	confidence (±)	2.3	6.1	5.5	7.0

* Ratio is to the no interference case.
† 95% confidence interval.

mechanism to ensure consistency. Reconciliation runs infrequently (once per hour) at low priority so it should have limited effect on system performance. Our experience is that the cost of reconciliation on workstations can be minimized by scheduling it to run most frequently during "off-hours" and less frequently for volumes with low update rates. For servers hosting many volumes, the costs of background reconciliation can be noticeable. A complete examination of reconciliation's effects on performance is the subject of future work.

Update propagation runs (in the background) on all machines at all times. This section examines the effect of update propagation on workstation performance with the following indirect measurement. A benchmark suite was run on a standard UNIX file system (UFS) on a target machine. Then the benchmark was repeated with a replica of a Ficus volume stored on the same disk as the UFS file system. An update-intensive workload was run on a second machine that also hosted a replica of the same Ficus volume. Hence the target machine received a large number of update propagation requests and was "pulling over" updated copies of its local file replicas. The local UFS benchmark was run with 0, 1, 2, and 3 Ficus volumes replicated on the same disk, each Ficus volume experiencing heavy update propagation from different sites. Table II shows the extent to which the local UFS activity was slowed by hosting the Ficus replicas under heavy load.

The results of the experiments shown in Table II demonstrate that heavy update activity on a remote volume replica can seriously affect a storage host. It should be pointed out that most of the slow-down here is likely due to disk contention. Both the local UFS benchmark and the remotely generated Ficus load are very disk I/O intensive. These results may lead us to rethink the whole-file-propagation nature of Ficus replication. Currently, for every write operation to a replicated file, an update notification is sent to all replica hosts. Those hosts, in turn, pull over a complete new copy of the file, an architecture chosen for its simplicity rather than its performance. Possible changes to address this performance might include page-level propagation, batching, or delay of update notifications such that a flurry of updates to a single file cause only a single update notification and subsequent pull of the contents. These optimizations should reduce the load on volume replica storage hosts.

Table III. UFS, NFS, and 2-replica Ficus over the Internet*

	MAB	cp	grep
UFS	169.1 (n/a)	216.8 (n/a)	149.1 (n/a)
NFS (Internet)	628.3 (3.7)	344.8 (1.6)	243.0 (1.6)
Ficus (Internet)	204.5 (1.2)	259.2 (1.2)	182.2 (1.2)

* Ratios relative to the local (UFS) case are in parentheses.

Other network environments

The previous measurements examined performance in the local area network environment. Ficus is also in use at sites connected via telephone modems, and over the Internet. Table III shows the results of experiments designed to show the performance of Ficus for sharing data across slow links as compared to the current primary alternative, NFS. The Internet connection (between beverly.cs.ucla.edu and earvin.isi.edu) has four gateways and a mean round trip time of 10 ms. By comparison, a local Ethernet round trip averages 2 ms.

These measurements validate the experience of many, that using NFS to share data over the Internet is only marginally feasible (600 seconds for the MAB, about 4× slower than local). However, keeping a replica at all locations where access is required gives all participants performance almost equivalent to local UNIX (204 seconds for Ficus versus 169 for UNIX). One should not be misled by these measurements: very few people would expect to run NFS across a slow link; NFS is not designed for wide-area, low-bandwidth file sharing as shown by its performance. This comparison illustrates that when access to a file is needed in multiple places connected only by the Internet or a modem, replication is the key to giving users in both locations local-quality access.

Experiences with optimistic replication

We have argued that the environments targeted by this work mandate the optimistic replication strategy adopted. However, if inconsistency frequently renders data unavailable, or if the conflicts prove difficult to resolve, then the costs of optimism might outweigh the benefits. Here, we report on the file system usage studies that led us to believe that optimistic strategies would seldom lead to conflicting updates. Then we present results of empirical studies measuring the number of conflicts actually experienced in our working environment. Finally, we address the cost of conflicts both when automated conflict resolvers are employed and when conflicts are resolved manually.

File system usage studies

Analyses of file usage in existing systems suggest that both concurrent read and write sharing is quite rare.[17] In the referenced examination of an academic UNIX system, of all files read during a seven-day study period, under 7% were read by more than one user during the next week (and the vast majority of the files read by multiple users were news articles and hence read-only; only 1.3% of files owned by normal users were read by multiple readers). Update sharing was even rarer. Just 2.4% of all files written (and less than 1% of user-owned files) were updated by multiple writers.

Similar results were found for directories. To the extent that concurrent sharing is rare, conflicts should be minimal.

Analysis of commercial systems suggests similar conclusions. The re-analysis of trace data collected in the 1970s[18] from three commercial IBM timesharing environments found that from 3% to 12% of the files accessed are updated by multiple users.[19] A more recent study looked at traces taken in three different commercial environments. Conflict rates of 0.11 and 0.02 conflicts per user per week were found in the programming and "personal productivity" environments, respectively, after adjusting for easily resolved conflicts caused by the mail system. In the third environment, a shared accounting database generated a mean of 8.57 conflicts per user per week, indicating that either application-specific conflict resolution would be needed or the database should be separated into units of smaller granularity.[20]

In a distributed AFS environment, a study finds that over 99% of all updates are by the same user that made the previous update, and that the likelihood of different users modifying a file less than one day apart was less than 0.75%.[21] Excluding system files, the chance of different users modifying the same object within a week dropped to less than 0.4%. Despite the collaborative nature of the work going on in the measured environment, concurrent write sharing was not common. Again, conflicts should be rare if optimistic replication were employed in this environment. Together, these studies form the basis of the argument that file access sharing is rare, shared update is even rarer, and hence the optimistic approach to concurrency control is feasible.

Experimental evidence

While these results guided the design of Ficus, with a working optimistically replicated file system in use we are now in position to measure the occurrence of conflicts in practice.[7] The first column of Table IV shows the conflict rate observed in an academic environment running Ficus during a nine month period. The experimental environment consists of 15 workstations, one file server, and 12 users doing software development and word processing. All user files and the system source code base are replicated between 2 and 12 times. Most machines are on the same local Ethernet. Three machines operate in a primarily disconnected mode, connecting via modem nightly or weekly to reconcile. As the table shows, with fourteen million file updates over nine months of actual use, 489 update-update conflicts occurred. Of these, 338 could have been automatically resolved. (162 of those 338 actually were resolved automatically, but since we added more resolvers to our suite frequently during this period, some conflicts that could have been resolved automatically by the end of the period were not automatically resolved at the beginning. We report both the number actually resolved and the number that could have been resolved, as the latter is a better indication of how the system currently operates.) Over 700,000 name creations produced only 128 name conflicts. During the measured period, no name conflict resolvers were in use, so all 128 name conflicts required manual intervention. Since then, several name conflict resolvers have been written and exercised.

Table IV contains three lines without values, those for disconnected operation remove/update conflicts, total name creates for disconnected operation, and name conflict rate for disconnected operation. These values are not available for the nine month data set. For a four month subset of the nine month data set, 10% of all remove/update conflicts occurred on disconnected volumes. For the same four month period, approxi-

Table IV. Conflict statistics[7] gathered over 9 months

	all volumes	disconnected volumes
total non-directory updates	14,142,241	1,502,378
update/update conflicts	489	379
automatically resolvable	338	316
not automatically resolvable	151	63
update conflict rate	0.0035%	0.0252%
	1 in 28,571	1 in 3968
unresolvable rate	0.0012%	0.0041%
	1 in 83,333	1 in 24,390
remove/update conflicts	98	see text
total name creates	708,780	see text
name conflicts	128	71
name conflict rate	0.0181%	see text
	1 in 5525	

mately 15% of all name creates occurred in disconnected volumes. For that period, the name conflict rate in the disconnected volumes was around 3 times as high as the rate for all volumes. We cannot guarantee that the same trends occurred throughout the rest of the nine month period, but very likely they did. Even with fairly conservative assumptions, the disconnected volumes' remove/update conflicts and name conflict rate would not be unacceptably high.

Conflicts are the result of unsynchronized concurrent updates to multiple replicas. In our environment, updates of a single file by multiple users in a short period are quite uncommon. Hence most conflicts are the result of the failure of updates to propagate in a timely manner. Because we operate in an experimental environment, machines frequently crash or are rebooted for software changes, temporarily preventing propagation. There is reason to believe that production environments would experience dramatically fewer of this type of conflict.

Propagation is also typically not possible for replicas stored on disconnected sites. The second column of Table IV shows conflict statistics for volumes involved in disconnected operation. Though such volumes receive only about 10% of all updates, they are responsible for over three quarters of the observed update/update conflicts. Despite the lack of update propagation, the conflict rate in disconnected volumes is still quite low. Since most files are the responsibility of a single individual, we can match the movement of the user with the pattern of reconciliation. For example, when the user moves from home to office, reconciliation synchronizes the two environments. In effect, the user functions as a "human write token."[8]

Most conflicts on disconnected sites result from automated programs (such as mail sorting or data collection), or from the number of simple shared databases present in a typical UNIX system. Examples of the latter include various game score files commonly found in academic computing environments. Since these databases are common to multiple users, they experience more frequent concurrent update. Fortunately, their particularly simple semantics make these types of files excellent candidates for automatic resolvers. This observation is reflected by the proportion of conflicts that were automatically resolvable for disconnected volumes being extraordinarily high.

Impact of conflicts

Although Table IV suggests that conflicts are rare (less than two per user per week, counting name conflicts and update/remove conflicts), even those might prove disruptive if they are difficult to resolve. Our experience suggests that resolution is generally easy, for several reasons: many conflicts can easily be resolved automatically; when required, manual resolution of conflicts has not proved difficult; and finally, conflicts on shared files are often avoided by higher-level agreements (even in the unreplicated case). However, it should be noted that our environment is dominated by experienced users. We cannot assert with confidence that naive users would not be troubled by even rare conflicts.

Experience with conflicts indicates that a large percentage occur on files with very simple, regular semantics. Based on this observation, we constructed a number of simple automated conflict resolvers. For example, files are frequently employed to store processed forms of other data; if this form can be regenerated, a resolver can simply discard the processed form. Examples of this class of file include pre-formatted manual pages, compiled object files (if source code is present), and XMH `.xmhcache` files. With slight semantic relaxation, a larger class of files can be automatically handled. For example, at the cost of occasionally seeing a message twice, conflicts on MH `.mh_sequences` files can be resolved by simply taking the largest possible sequence number. Similarly, any concurrent updates to `.newsrc` files can be reconciled almost perfectly. The use of a few simple resolvers such as these automates handling of the majority of update/update conflicts, reducing user intervention by two thirds. On average, the users in this study responded to less than one conflict per week, counting update/remove and name conflicts. Adding name conflict resolvers (as we have done) further lowers this rate.

When Ficus detects a conflict on a file for which no resolver is registered, it blocks normal access to the file and sends a mail message concerning the conflict to the file owner. Conflicting updates of a general text file are usually easy to identify by simply comparing the two replicas. In these cases, it is easy to select the missing update from one version, apply it to the other, and then cause the version vector of the repaired copy to become dominant using tools provided with Ficus.

In our environment, files that are subject to frequent update by multiple users are typically already protected from multiple updates by higher-level protocols such as revision control systems. Note that the case of shared databases is not addressed here, and is the subject of further research.

Lessons learned

Building and using a large software system in a research setting generates a number of important lessons that might not be apparent from a paper design. Here we reflect upon what was learned from our experiences in this effort.

Lessons concerning replication

Optimism works: A most important conclusion is that optimism works well for our type of use. The *laissez faire* approach to update propagation with periodic reconciliation maintains sufficient consistency so that conflicts are rare and not a problem.

Subtle Details: While the basic reconciliation algorithms are relatively straightforward, getting the details right is subtle. These are details which one would tend to overlook unless building a real system for near-production use. An example is handling hard failures of media in which some of the state of the two-phase garbage collection is lost. While the file content data stored in a damaged replica is generally replicated or available on dump tapes, the state of the garbage collection algorithms with respect to that replica is available nowhere else. In practice, a mechanism is required to enable the other replicas to "forget" about the replica which has been lost. Other examples include dealing with replicas in file systems that are full; concurrent creation, deletion, and moving of replicas; replica switch-over on machine failure; etc. These are example of the type of complexity that may be missed without use of a working file system.

Naming model: The semantics of UNIX file naming are very close to being suitable for the distributed replicated filing application. UNIX largely treats the management of the name space separately from the files those names point to, so the user model of naming does not need to change radically. However, optimistic directory management requires that the system tolerate name conflicts and multiple names for directories. The ability to support a DAG as opposed to a strict tree-structured name space necessitated re-implementing the directory service for use in Ficus.

Leveraging reconciliation: Directories are one example of a rather common type of data structure, a sequence of records mapping a name to a value, whose update semantics consist only of creation and deletion. Ficus uses the same management and reconciliation code both for normal directories and for graft points (volume location information). As a result, user-interface programs (such as `touch`, `mv`, and `ls`) can manipulate graft points, again resulting in saving of effort. In hindsight, we would like to have have packaged the directory reconciliation algorithms in an even more general and abstract form so that any similar data structure could utilize them.

Software replication: Ficus demonstrates that a software solution to high reliability and availability is feasible. A software solution (as opposed to hardware-based mirror disks, for example) admits use of commodity hardware at great price/performance benefit. It permits greater flexibility and the changing of replication details without adding or moving hardware, eases incremental growth, and provides the ability to maintain replicas at widely separated geographic locations. Packaged as a module, software reconciliation can easily slide in or out of a configuration on a per-volume basis. The primary cost in the case of software is in the initial development and in simply storing and propagating the extra copies of data.

Benefits of simple resolvers: A small number of very simple resolvers (in addition to the directory resolver) take care of most of the conflicts that do occur. With the addition of a few additional trivial resolvers, users will be shielded from the vast majority of conflicts, and the statistics reported in Table IV can be expected to improve.

Selective Replication: Volume-granularity replication is only part of the solution. Selective control, the ability to control what objects from the volume are locally replicated at each volume replica, is required for performance, functionality, and scalability. However, it further complicates the algorithms and control structures, which must now be robust to changing replication patterns and properly respond to inaccurate data with regard to where specific objects are stored. The added complexity, however, provides benefits to the user in terms of functionality, performance, and usability.

Administrative costs: The administrative costs associated with replication are a potential concern. To the extent that replication is transparent, it leaves all of the other administrative tasks intact; yet it adds more degrees of freedom in deciding how to configure a volume. Which sites should store it? Where should each site store it? How often should they reconcile? How should file system backup dumps be set up? Fortunately, these are largely one-time initial costs, and add little ongoing burden.

Although replica setup requires administrative planning, it can ease other tasks. Parts of a network file system that need to be stored on each machine may be placed in replicated volumes. Then updating one of these files requires changing the file only once, rather than going to each machine. Existing tools such as rdist also allow replication, but in a limited manner (master-slave with updates to the master only). Ficus supports full peer-to-peer, update anywhere replication.

Backups: The existence of file replicas on machines with relatively independent failure modes largely alleviates the recovery-from-hardware-failure motivation for doing nightly backups. Backups serve primarily to recover from accidental removal or overwriting of data by users. Note that a file versioning mechanism (which has been prototyped as a stackable layer), almost completely removes this latter motivation for backups. Perhaps only occasional tape dumps will suffice for recovery from catastrophes that would wipe out all replicas. In our working environment, we do not directly backup disconnected machines and instead rely entirely on replication for their protection from hardware failure.

Lessons concerning layering

Cache consistency: Cache consistency issues arise both between the Ficus layers and between distinct Ficus hosts; when data is updated in one layer or machine other caches should be notified. Consistency can be maintained by treating information as a "hint," invalidating out-of-date information, or avoiding caching. We found that the needs for consistency varied in different parts of Ficus.

Inter-layer cache consistency[22] was implemented late in Ficus' development. Ficus replication does not currently use this mechanism. In some cases, Ficus uses data optimistically, although this can cause problems in some situations (for example, users of another replica may not see a file removal immediately). In other cases Ficus avoids caching data to make inconsistencies impossible (for example, in handling memory mapped files) or we have constructed ad hoc mechanisms to access cached data (in replica selection). Overall, Ficus works adequately in practice without inter-layer cache coherence. Only very occasionally were coherence problems visible to the user. However the structure and in some cases the performance of Ficus would benefit from inter-layer coherence.

The situation with cross-machine cache consistency is somewhat different. Ficus operates with an identical cross-machine caching policy to NFS. While it is possible to exercise the lack of cache coordination, problems rarely occur in practice. Optimism appears justified in this arena as well. However, in those rare times when the cache consistency assumptions are not justified, problems result which are confusing and not repeatable. While a usable system can be built ignoring cross-machine cache consistency, the lack of it will be an occasional source of obscure problems.* Our experiences suggest that applications which are suitable to optimistic replication are less likely to need stronger than NFS-quality cross-machine cache consistency, although we have not tried to verify this hypothesis.

Layer division: In hindsight, we did not fully appreciate the power of the layered filing technology when we initially designed Ficus. One example is our split of the original logical layer into logical and replica selection layers for added flexibility. As another example, much of the physical layer is devoted to dealing with the fact that UNIX does not present inode-level access. Ficus simulates the inode-level interface using the underlying UFS as a flat file service at considerable performance penalty. We should have built a separate layer to simulate such a service; that layer could simply be removed when a storage layer with an inode interface became available. We would also expect and recommend that a commercial implementation of a layered file system support inode-level access.

Sharing files across administrative domains: Ficus, as originally implemented, operated within a single administrative domain. That is, the file ownership and permissions model was as in UNIX, with a coordinated space of user identifiers and a fully shared file hierarchy. This was a clear limitation to scaling Ficus across the Internet. We have relaxed this restriction with a user-id mapping layer. Other extensions to improve security and support cross-domain sharing over a public network are described elsewhere.[24, 25]

RELATION TO OTHER WORK

Ficus is related to, or draws from, a variety of other work. Ficus is the intellectual descendant of Locus[26] in that both have the goal of providing a network-transparent file system that supports partitioned update with automatic recovery. While experimental versions of Locus permitted partitioned update, no optimistic directory update was allowed, and no automatic reconciliation of any object was ever supported. Commercial versions of Locus used only primary-site reconciliation. Ficus avoids the design choices (the need for all sites to agree on the current network topology) that fundamentally prevented the Locus approach from scaling beyond a relatively small number of sites. Further, Ficus is a modular extension to the UNIX file system, where Locus was a full distributed operating system.

The weakly consistent replication protocol in which updates are performed synchronously to a single replica and propagated asynchronously to the others is similar to that used in the Grapevine's electronic mail system.[27] The scaling requirements and failure characteristics of a wide scale Internet environment led Grapevine to this class of solutions just as it led Ficus in this direction.

Bayou[28] is a replicated storage system based on the peer-to-peer architecture. Like Ficus, Bayou provides support for application-dependent resolution of conflicts. However, unlike Ficus, Bayou does not attempt to provide transparent conflict detection. Applications must specify a condition that determines when a conflicting access has been made, and must themselves specify the particular resolution process. Bayou pro-

* Baker[23] reports simulations and instrumentation of the Sprite Operating System which lead to the conclusion that lack of cache consistency is a problem. Either there is considerably more shared file activity in the Sprite environment, or the vast majority of accesses to stale data in NFS-derived systems go unnoticed; both are probably the case.

vides *session guarantees*[29] to improve the perceived consistency by users. Additionally, Bayou establishes strong guarantees about its data—writes can be classified either as *committed* or *tentative*. It does not support any form of selective replication, so the databases (the Bayou replication unit) must be fully replicated at all storage sites.

Ficus derives its notion of volumes from the Andrew File System (AFS).[30] It shares many of the same goals as AFS for scale, and Coda[31] for reliability and availability via optimistic replica management. Ficus differs fundamentally in its peer-to-peer model of machine interaction as opposed to the client-server model employed in AFS and Coda. Coda allows replication among a backbone of closely coupled servers while clients on workstations or mobile machines check out whole files. Where any Ficus machine may also be fully functioning server, Coda clients cannot exchange locally stored data with other clients. Coda clients utilize on-disk caches that allow them to operate while disconnected from servers to the extent that needed files are present in the local cache, whereas Ficus users require a local replica if a mobile machine is to be able to continue access while disconnected. Finally, Coda's client/server and consistency models assume geographically co-located servers for good performance; Ficus' peer model works well with geographically distributed and weakly connected servers.

The ISIS environment's Deceit file system,[32] like Ficus, utilizes NFS. Deceit has a mode that permits partitioned update, but it does not support automatic directory reconciliation.

The Harp file system[33] implements replication for a UNIX client-server environment using primary-copy concurrency control. Harp achieves high performance and reliability by combining write-behind logging techniques with an uninterruptible power supply that allows logs to be forced to non-volatile storage after a power failure.

The stackable layers architecture in Ficus builds on several areas of related work. It is in many ways the file system analog of Ritchie's System V streams,[34] and of the x-Kernel's notion of protocol stacks.[35, 36] It is compatible with and motivated by the micro-kernel philosophy growing out of the Mach work,[37] but it provides modularity through software structuring conventions rather than with servers in independent address spaces. Several efforts at Sun Microsystems have independently explored stackable filing with approaches similar to ours.[38-40]

STATUS AND CONCLUSIONS

The system as described in this paper, including reconciliation, is operational and in daily use since the middle of 1990. Ficus is constructed from modified SunOS 4.1.1 source code, the modifications currently consisting of approximately 42,000 lines of C-language kernel source code and 26,000 lines of user-level utility code. Within the laboratory of its developers, all source code development, user files and shared system binaries, etc., are replicated under Ficus. Given an average project size of 12 people, we have accumulated approximately 720 person-months of user experience. While most of this experience is with its use within an office, Ficus has also been used to share data over the Internet, and over phone lines in primarily-disconnected mode.

We have gained considerable experience from building and working in an optimistically replicated system. All of our experience supports the view that optimistic replication is very attractive. Providing high performance, high availability, scalable distributed computing service demands an optimistic approach, an approach that has proven feasible. It is our hope that the facilities described in this paper will make that high quality service commonplace, as they require no special hardware and can easily be added to many existing systems. Many applications should benefit from the ease with which the basic reconciliation service can be re-targeted beyond its initial use for directory management, as shown by our success in using it to manage Ficus' replicated volume location tables.

All indications are that these conclusions become "all the more so" as scale increases in terms of geographic distance and numbers of files. The alternative, pessimistic replica coordination, becomes increasingly expensive in terms of both delay and availability. Further, the kind of unstructured shared update that could lead to conflicts becomes even less common. Since stronger consistency guarantees can be provided on top of an optimistic base (and the reverse is not the case), we conclude that optimism is the preferred policy at the lowest level.

It has also been our experience that the lack of portability and flexibility in hardware choices that has resulted from the use of a proprietary operating system source code base has been an on-going source of frustration. Partially as a result of the restrictions on wide-spread distribution of a kernel-based implementation, we have extracted many of the ideas, concepts, and algorithms of Ficus and implemented them in an out-of-kernel form. This application, called Rumor,[41] provides peer-to-peer optimistic replication entirely in user space, though it does not support all features offered by Ficus, such as transparent replica selection.

Finally, this work opens up a number of relevant research directions where one can expect to make rapid progress, and provides the tools to investigate them. For example, individual researchers can explore a variety of synchronization and consistency policies in a replicated filing environment, easily adding their own implementations to experiment with functionality. The use of the stackable file system technology has been a boon to this research, and should contribute to the future leverage of the Ficus system.

Acknowledgments

This work was sponsored by DARPA under contract numbers F29601-87-C-0072 and DABT63-94-C-0080 and by the National Science Foundation under Grant No. IRI-9501812.

Ficus is the work of many people. Additional contributors to Ficus include: Deiter Rothmeier (reconciliation and kernel), Wai Mak (kernel), Jeff Weidner (extended attribute layer), John Salomone (administration), Steven Stovall (kernel), Greg Skinner (resolvers), and Michial Gunter (reconciliation performance).

REFERENCES

1. D. Stott Parker, Jr., Gerald Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline, 'Detection of mutual inconsistency in distributed systems', *IEEE Transactions on Software Engineering*, **9**(3), 240–247 (1983).
2. David Ratner, Gerald J. Popek, and Peter Reiher, 'Peer replication with selective control', *Technical Report CSD-960031*, University of California, Los Angeles, July 1996.

3. Michael J. Fischer and Alan Michael, 'Sacrificing serializability to attain high availability of data in an unreliable network', *Proceedings of the ACM Symposium on Principles of Database Systems*, March 1982.
4. Richard G. Guy, Gerald J. Popek, and Thomas W. Page, Jr., 'Consistency algorithms for optimistic replication', *Proceedings of the First International Conference on Network Protocols*. IEEE, October 1993.
5. Dennis M. Ritchie and Ken Thompson, 'The UNIX time-sharing system', *Communications of the ACM*, **17**(7), 365–375 (1974).
6. K. Mani Chandy and Leslie Lamport, 'Distributed snapshots: Determining global states of distributed systems', *ACM Transactions on Computer Systems*, **3**(1), 63–75 (1985).
7. Peter Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek, 'Resolving file conflicts in the Ficus file system', *USENIX Conference Proceedings*. University of California, Los Angeles, USENIX, June 1994, pp. 183–195.
8. John S. Heidemann, Thomas W. Page, Jr., Richard G. Guy, and Gerald J. Popek, 'Primarily disconnected operation: Experiences with Ficus', *Proceedings of the Second Workshop on Management of Replicated Data*. University of California, Los Angeles, IEEE, November 1992, pp. 2–5.
9. Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier, 'Implementation of the Ficus replicated file system', *USENIX Conference Proceedings*. University of California, Los Angeles, USENIX, June 1990, pp. 63–71.
10. John S. Heidemann and Gerald J. Popek, 'File-system development with stackable layers', *ACM Transactions on Computer Systems*, **12**(1), 58–89 (1994). Preliminary version available as UCLA technical report CSD-930019.
11. Ashvin Goel, 'View consistency for optimistic replication', *Master's Thesis*, University of California, Los Angeles, February 1996. Available as UCLA technical report CSD-960011.
12. John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson, 'A trace-driven analysis of the UNIX 4.2 BSD file system', *Proceedings of the Tenth Symposium on Operating Systems Principles*. ACM, December 1985, pp. 15–24.
13. John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch, 'The Sprite network operating system', *IEEE Computer*, 23–36 (1988).
14. John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West, 'Scale and performance in a distributed file system', *ACM Transactions on Computer Systems*, **6**(1), 51–81 (1988).
15. Thomas W. Page, Jr., Richard G. Guy, Gerald J. Popek, John S. Heidemann, Wai Mak, and Dieter Rothmeier, 'Management of replicated volume location data in the Ficus replicated file system', *USENIX Conference Proceedings*. University of California, Los Angeles, USENIX, June 1991, pp. 17–29.
16. John K. Ousterhout, 'Why aren't operating systems getting faster as fast as hardware?', *USENIX Conference Proceedings*. USENIX, June 1990, pp. 247–256.
17. Rick Floyd, 'Short-term file reference patterns in a UNIX environment', *Technical Report TR-177*, University of Rochester, March 1986.
18. Alan J. Smith, 'Analysis of long term file reference patterns for application to file migration algorithms', *IEEE Transactions on Software Engineering*, **7**(4) (1981).
19. Øivind Kure, 'Optimization of file migration in distributed systems', *Technical Report UCB/CSD 88/413*, University of California, Berkeley, April 1988.
20. Geoffrey H. Kuenning, Gerald J. Popek, and Peter Reiher, 'An analysis of trace data for predictive file caching in mobile computing', *USENIX Conference Proceedings*. USENIX, June 1994, pp. 291–306.
21. James J. Kistler and Mahadev Satyanarayanan, 'Disconnected operation in the Coda file system', *ACM Transactions on Computer Systems*, **10**(1), 3–25 (1992).
22. John Heidemann and Gerald Popek, 'Performance of cache coherence in stackable filing', *Proceedings of the 15th Symposium on Operating Systems Principles*, ACM, December 1995, pp. 110–127.
23. Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Sherriff, and John K. Ousterhout, 'Measurements of a distributed file system', *Proceedings of the Thirteenth Symposium on Operating Systems Principles*. ACM, October 1991, pp. 198–211.
24. P. Reiher, T. Page, S. Crocker, J. Cook, and G. Popek, 'Truffles—a secure service for widespread

file sharing', *Proceedings of the The Privacy and Security Research Group Workshop on Network and Distributed System Security*, February 1993.
25. P. Reiher, S. Crocker, J. Cook, T. Page, and G. Popek, 'Truffles—secure file sharing with minimal system administrator intervention', *Proceedings of the 1993 World Conference On Tools and Techniques for System Administration*, April 1993.
26. Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel, 'The LOCUS distributed operating system', *Proceedings of the Ninth Symposium on Operating Systems Principles.* ACM, October 1983, pp. 49–70.
27. Andrew D. Birrell Michael D. Schroeder and Roger M. Needham, 'Experience with Grapevine: The growth of a distributed system', *ACM Transactions on Computer Systems*, **2**(1), 3–23 (1984).
28. Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser, 'Managing update conflicts in Bayou, a weakly connected replicated storage system', *Proceedings of the 15th Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, December 1995, pp. 172–183. ACM.
29. 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', *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, sep 1994, pp. 140–149.
30. Mahadev Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector, and Michael J. West, 'The ITC distributed file system: Principles and design', *Proceedings of the Tenth Symposium on Operating Systems Principles.* ACM, December 1985, pp. 35–50.
31. Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere, 'Coda: A highly available file system for a distributed workstation environment', *IEEE Transactions on Computers*, **39**(4), 447–459 (1990).
32. Alex Siegel, Kenneth Birman, and Keith Marzullo, 'Deceit: A flexible distributed file system', *USENIX Conference Proceedings.* USENIX, June 1990, pp. 51–61.
33. Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams, 'Replication in the Harp file system', *Proceedings of the Thirteenth Symposium on Operating Systems Principles.* ACM, October 1991, pp. 226–238.
34. Dennis M. Ritchie, 'A stream input-output system', *AT&T Bell Laboratories Technical Journal*, **63**(8), 1897–1910 (1984).
35. Norman C. Hutchinson and Larry L. Peterson, 'The x-Kernel: An architecture for implementing network protocols', *IEEE Transactions on Software Engineering*, **17**(1), 64–76 (1991).
36. Larry L. Peterson, Norman C. Hutchinson, Sean W. O'Malley, and Herman C. Rao, 'The x-Kernel: A platform for accessing Internet resources', *IEEE Computer*, **23**(5), 23–33 (1990).
37. Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, 'Mach: A new kernel foundation for UNIX development', *USENIX Conference Proceedings.* USENIX, June 9–13 1986, pp. 93–113.
38. David S. H. Rosenthal, 'Evolving the vnode interface', *USENIX Conference Proceedings.* USENIX, June 1990, pp. 107–118.
39. Glenn C. Skinner and Thomas K. Wong, '"Stacking" vnodes: A progress report', *USENIX Conference Proceedings.* USENIX, June 1993, pp. 161–174.
40. Yousef A. Khalidi and Michael N. Nelson, 'The Spring virtual memory system', *Technical Report SMLI TR-93-9*, Sun Microsystems, February 1993.
41. Peter Reiher, Jerry Popek, Michial Gunter, John Salomone, and David Ratner, 'Peer-to-peer reconciliation based replication for mobile computers', *Proceedings of the ECOOP Workshop on Mobility and Replication*, July 1996.