Computer Science Department Technical Report

University of California

Los Angeles, CA 90024-1596

ARCHITECTURE OF THE FICUS SCALABLE REPLICATED FILE SYSTEM

Thomas W. Page, Jr.
Richard G. Guy
Gerald J. Popek
John S. Heidemann

March 1991

CSD-910005

Architecture of the Ficus Scalable Replicated File System*

Thomas W. Page, Jr. Richard G. Guy Gerald J. Popek†

John S. Heidemann

Department of Computer Science
University of California Los Angeles
3531 Boelter Hall
405 Hilgard Avenue
Los Angeles, CA 90024-1596

Abstract

Ficus is a distributed file system designed to scale up to very large networks of UNIX systems, ranging from portable units and workstations to large file servers. It provides very high availability for read and update, utilizing an optimistic *one copy availability* policy with conflict detection and automatic reconciliation of the name space. Ficus is packaged as a pair of layers which can be configured on top of the UNIX file system, coexisting with other extended file system features using an stackable file system switch.

This paper presents the architecture of Ficus and the rationale behind the design decisions. Measurements of the current implementation are reported which indicate that performance is reasonable both within local clusters of cooperating machines and between geographically distributed clusters.

*This work is sponsored by DARPA under contract number F29601-87-C-0072.

†Author also associated with Locus Computing Corporation

1 Introduction

Network file systems for UNIX have been quite successful at providing a nearly single machine view of the collection of files on a small number of machines connected by a local area network. The extension to wide area networks with very large numbers of machines substantially changes the nature of the problem. Whereas in a small network, node or communications link failure is a relatively rare event, in a system the size of the Internet, the system is always in a state of partial failure (or partial operation). Large scale networks typically span many administrative and organizational boundaries, rendering central coordination virtually impossible. The scale also prohibits use of algorithms which require consistent global state information. Nevertheless, a distributed filing environment providing a very high degree of network transparency would greatly facilitate geographically distributed cooperative work.

Selective file replication is a key component of any highly reliable, large scale distributed filing solution. Replication is important for performance in that a copy of data may be located "near" to where it is needed. Replication is critical to reliability in an environment where node failures and communications interruptions are common. We wish an approach to replication that is suitable both to small work

groups, but at the same time scales to very large environments. Users should retain considerable selective control over the replication parameters (what, where, how many, and so on) and availability of data should strictly increase with the number of copies. Overhead should be low, not only for actual execution time, but also in terms of introducing and administering the service, and explicating any new user interfaces. Finally, decentralized operation is essential in order to scale. It is this environment to which the Ficus distributed file system is targeted.

This paper describes the architecture and design rationale of Ficus. The architecture is unique in at least three important ways. First, Ficus supports very high availability for write, allowing uncoordinate d updates when at least one replica is accessible. *No lost updates* semantics are guaranteed; conflicts are reliably detected and directory updates are automatically reconciled. Asynchronous update propagation is provided to accessible copies on a "best effort" basis, but is not relied upon for correct operation. Rather, periodic *reconciliation* insures that, over time, all replicas converge to a common state. We argue that serializability is not provided in single machine file systems and is not required in distributed file systems. This kind of policy seems necessary and appropriate for the scale and failure modes in a nation-wide file system. This paper describes the architecture of a facility providing optimistic replica management; details of the reconciliation algorithms may be found in [3, 10].

Second, the replication service is packaged so that it may be inserted above the base UNIX file system on any machine running a stackable file system interface. The modular architecture permits replication to co-exist with other independently implemented extended filing features. Modules packaged in this way may be distributed independently of the kernel and other portions of the file system. Further, Ficus implements replication in such a way that it is largely independent of the underlying file system implementation, permitting a high degree of configuration flexibility and portability. We report in this paper our experiences gained from building replication in this manner. One hopes that the insights gained will be useful as other pluggable layers are built, to provide additional extended filing environment features. The reader interested in further details about the extensible interface used in Ficus is referred to [5, 4].

Third, the optimistic consistency policy and reconciliation mechanism used for incorporating uncoordinated updates to replicated directories is exploited to manage the super-tree of replicated volumes (analogous to the AFS volume location database [17]). Thus no new replication and consistency policy or mechanism is needed for this fairly general name service. Not only does this design reduce implementation and simplify the architecture, it provides the appropriate semantics and availability for naming data.

1.1 Relation to Other Work

Ficus is related to, or draws from, a number of existing systems. It is the intellectual descendant of the earlier versions of Locus [21], in that both have the goal of providing a network transparent file system supporting partitioned update with automatic recovery. Ficus uses the version vector scheme [10] that originated in university Locus. Ficus, however, avoids the design choices which fundamentally prevent the Locus approach from scaling beyond a relatively small number of sites. Further, Ficus is a modular extension to the UNIX file system, not a full distributed operating system.

Ficus derives its notion of a volume as a granularity of sub-tree management from the Andrew File System [17]. It shares many of the same goals as AFS for scale, and Coda [18] for reliability and availability via optimistic replica management. Coda makes similar use of version vectors for update-update conflict detection. Ficus differs from AFS and Coda in its basic model of a distributed file system; where AFS and Coda employ a backbone of servers with workstations which check out whole files, Ficus permits full status replicas on any machine in the system. While Ficus addresses file system modularity through stackable layers, Coda emphasizes security.

The ISIS environment's Deceit file system [19], like Ficus, utilizes NFS. Deceit does not support further extensibility, and while it has a mode which permits partitioned update, it does not support automatic directory reconciliation.

The stackable layers architecture in Ficus is related to several other pieces of work. It is in many ways the file system analog of Ritchie's System V streams [14], and of the x-Kernel's notion of protocol stacks [7, 11]. It is compatible with and motivated by the open system and micro-kernel philosophy growing out of the Mach work [1]. Rosenthal at Sun Microsystems is independently exploring similar approaches to layered filing [15]. The Ficus work integrates and substantially extends the referenced concepts. Designing and building an operational system has led to considerable refinement as well.

1.2 Organization of the Paper

The next section presents design rationale underlying the Ficus architecture. Section three then details that architecture. In section four we report on the status of the Ficus implementation and measurements of its performance. Conclusions follow in the final section.

2 Design Philosophies

Our design philosophies are driven by several aspects of the definition of the problem and the target environment which we take as given. Foremost, we intend Ficus to be a real, usable facility and not merely a demonstration of potential. This means that we must base all of our assumptions and constraints on what really is, and not on what is convenient. Further it means that performance must be a primary concern; a solution which does not perform well cannot be considered a solution. This section details the characteristics of what we believe is the real environment in which this system will be used, and presents the philosophies which we derive from these characteristics. We also endeavor to point out important open problems.

2.1 Large Scale Distributed File Systems

Why does one want a scalable distributed file system? The primary answer to this question is that without it, access to remote files is inconvenient. The user shoulders the entire burden of locating remote data and issuing a correct name, dealing with the more complicated and cumbersome syntax and semantics of remote file access tools (such as ftp), finding files that have moved, manually coordinating multiple copies of files for which high availability is required, to name but a few of the inconveniences. This leads us to conclude that scalable file systems must be considerably easier to use than the remote file access tools they supplant.

To be easy to use, a large scale distributed file system should present a familiar interface. The syntax of access to remote files should be the same as that for access to local files. The semantics of file operations should closely match that with which users are familiar from single machine and local area distributed file systems. The failure modes should not be radically different from those with which users and, more importantly, existing programs are accustomed to encountering. Further, no significant performance penalty should be incurred in using the file system.

These requirements are all dimensions of *transparency*. To the degree that physical boundaries are hidden, users have a vastly simplified model of the resource. Name, location, failure, performance, and replica transparency are all major goals of Ficus .

2.2 The Impact of Scale

The scale of a system has tremendous impact on the nature of the solutions that may be considered, on the expected usage and sharing patterns which must be optimized for, and on the failure characteristics. Ficus is designed to scale to a very large number of sites, from a few to thousands or millions. While it places no limit on the number of nodes in the network which may access replicated files, nor on the number of replicated volumes in the universe, it as-

sumes that there is seldom call for more than a small number (up to a few dozen) of fully updatable replicas of any given volume. Fortunately, as the number of required replicas of a file increases, the need for and desired frequency of file update generally decreases.[1] Thus, with little loss of availability for update, we may employ a large number of read-only replicas with a smaller backbone of updatable copies in cases where very large replication factors are called for. Thus while Ficus places no hard limits on the number of first-class replicas, we consider the limited case to be the most important design point.

The notion of second class readwrite replicas which are used primarily in support of disconnected operation of laptops may also be considered. The overhead of creating and supporting a first class replica for every laptop computer that wants to replicate a subset of a volume temporarily (say for a weekend or a trip) may be prohibitive. Work is now underway to construct a layer which provides *second class replicas* in Ficus. A second class replica would not have a corresponding version vector component and hence could not participate in reconciliation. However, when a second class replica is checked in, the new layer would automatically propagate any updates that occurred in the checked out replica so long as no conflicting update had occurred to the first-class replicas, signaling a conflict otherwise. This new layer appears to address the disconnected operation of laptops adequately.

Network partitions are a frequent occurrence in a large internetworked environment. It is typically not possible for a site to determine what sites are in its partition for several reasons: the number of sites may be very large, it may be changing faster than any algorithm which determines partition membership can complete, and partitions are not necessarily clean (the "can talk to" relation is not transitive). It cannot even be assumed that any two sites will ever be able to talk directly to each other.

As a consequence of these characteristics of the network topology, no algorithms which require global agreement may be employed. For example, the mechanism used by Locus [12] to obtain location transparency by maintaining a globally replicated and consistent mount table is infeasible beyond very modest scale networks. The algorithm which recalculates the mount table each time the topology changes could never complete before the topology changed again. A scheme such as that used in NFS, which requires coordination among the system administrators to ensure that each machine mounts each filesystem in the same place, also cannot scale across the many administrative domains. Similarly many of the replica consistency solutions from read-one, write-all to the various flavors of voting cannot be employed in general, as updates would all too frequently be blocked[2].

The distributed file system must be easy to install and administer, particularly since it will span many administrative domains. It must be easy to add new sites, replicas, whole subnets, etc. Local autonomy over access to resources must be maintained. Such seemingly mundane issues as how backup dumps are to be done in a widely distributed, selectively replicated system require careful consideration.

2.3 Availability

Availability along with performance are the primary motivations for building replication for the large scale filing environment. While increasing the degree of replication raises the probability that a copy of a file will be accessible when it is needed, many consistency schemes trade off availability for update in order to achieve high availability and consistency for read access.

We take it as a given that in many cases if a copy of a file is accessible, it should be available for update. Consider the alternative: a user with work to do is informed that an otherwise accessible file may not be updated because a consistency policy might be violated. What does she do? She makes a copy of the file and updates it, taking upon herself the burden of merging and propagating those updates. Given this

[1] This is a sociological phenomenon unrelated to the replication mechanism in place. A file which is viewed as indispensable by many clients is also likely to have an associated assumption of stability, which discourages frequent updates.

[2] That is only to say that the file system at its base level should not enforce one of these consistency control algorithms. Ficus supports implementation of more restrictive policies (with stronger semantic guarantees) for individual files or file systems at a higher layer.

alternative, it is far preferable to permit the update, with the burden on the system to propagate changes as feasible, to reconstruct the directories and maintain the namespace, and to detect any conflicts that may result.

2.4 Optimism, Laziness, Hints, and Caches

The above assumptions about the necessity of high availability dictate an optimistic replica consistency policy. The argument is that shared access is actually relatively rare in practice. Concurrent shared access is even more rare, while conflicting concurrent shared access is rarer still. Yet when concurrent shared access is required, it is important. Hence, it makes more sense to detect and recover from conflicts than to add overhead to the normal case to prevent rarely occurring problems. Further, the overhead is often more than just synchronization costs; since some portion of the network is always inaccessible, assured mutual exclusion will sometimes be unattainable. Optimism appears not only justified, but required as scale increases.

Once we accept optimism as a philosophy and then engineer the system to tolerate situations in which multiple versions of data are simultaneously accessible, even more options become possible. For example, updates to any replicated data structure may be propagated lazily since access to an object which has not yet received an update is equivalent to the update having been made at an inaccessible node. Update propagation may be done on a "best effort" basis as the reconciliation algorithms can restore consistency later. Batched propagation of bursty updates is then a natural consequence that matches well with typical update behavior. Further, it avoids propagation altogether for files which are created and deleted a very short time later, a common scenario in many filing environments.

The optimistic philosophy may be extended to other replicated data objects in the system. For example, when a volume replica is added, dropped, or moved, the replicated volume location tables might become temporarily inconsistent pending update propagation or reconciliation. These tables may be managed lazily if the information therein is viewed as a hint which is self-validating on use. When a hint is right, it is very fast; when it is wrong, one can soon tell it is wrong. The related philosophies and techniques of optimism, laziness, and hints are essential to achieving good performance and high availability in the very large scale environment.

It has been observed that locality is a key reason why computers work at all. As the access time for remote data is often unavoidably slower than for local data, we must make extensive use of locality via caching to come close to achieving performance transparency in the nationwide filing arena. Caching must be employed at many levels throughout the architecture, and information clustered so as to exploit locality. While these observations may sound like motherhood, still their role in achieving acceptable performance is essential, and their influence in the design is pervasive.

2.5 Filing Environment Modularity via Stackable Layers

There is movement to modularize the process and memory portions of the UNIX operating system via micro kernels as exemplified by Mach [1, 13] and Chorus [16]. However, the filing service, a key component of most operating systems, has heretofore retained a largely monolithic implementation. Adding any new features to the filing environment is usually a daunting task, frequently requiring reimplementation of much of the file system. This situation generally prohibits all but the major operating system vendors from providing and distributing new filing services.

While replication is an important component of the solution to very large scale distributed filing, it is certainly not the only such component. Therefore replication services should be added in such a way that does not preclude adding other extended services, and does not require that it be reimplemented in the context of the other new features. This philosophy frees us from having to "do it all." If one's files require replication, use the Ficus layers; if one needs more sophisticated multiuser synchronization, configure in another layer to provide it; if one of the replicas resides on an untrusted machine, configure in an en-

cryption layer. Clearly the ability to snap together independently developed components to configure custom filing services on a site by site basis provides unprecedented flexibility. What results is a mechanism whereby independent researchers or vendors can deliver shrink-wrapped software modules which contribute file system functionality without rewriting, replacing or retesting large portions of the basic implementation. The Ficus project is a case study in providing a key piece of extended filing services using the stackable layers architecture.

2.6 Problems Not Explicitly Addressed

In the work on Ficus thus far, security and authentication facilities have not yet been implemented. It is our intention to integrate Kerberos [20] and other certificate mechanisms into Ficus to manage scaling the space of user identifiers and the associated authentication. An encryption layer is also under development to facilitate storing replicas on untrusted hosts.

While we advocate the one copy availability policy provided by the basic replication system, there are certainly applications which require and are willing to pay (in terms of synchronization overhead) for stronger consistency guarantees. It is our philosophy that these guarantees may be provided via the layered architecture as additional modules which layer above the present logical layer. A prototype consistency layer is now under construction to provide a single system image for a file system via a token mechanism.

3 Ficus Architecture

Ficus implements the replicated filing service in the context of a stackable layered file system architecture. Replication is provided by a pair of layers stacked upon a persistent storage service. Figure 1 shows the layers in a Ficus replicated file system architecture.

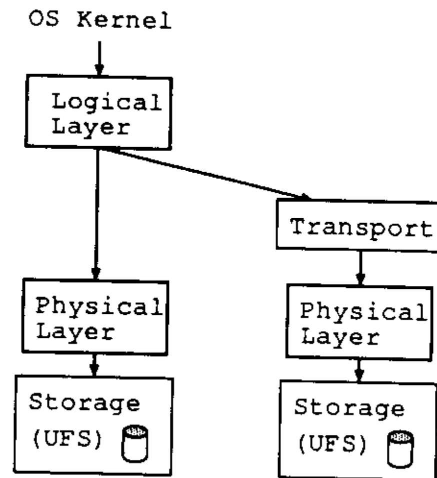

Figure 1: Ficus Stack of Layers

The *logical* layer provides to layers above it the abstraction of a single-copy, highly available file. The *physical* layer implements the concept of a file replica. Underneath the physical layer is a *persistent storage* layer such as the UNIX File System (UFS) in SunOS 4.1.

Ficus layers use an *extensible vnode interface* [5] suitable for constructing a variety of layered services. Of particular importance to Ficus is a *vnode transport* layer which invisibly extends the vnode interface across address space boundaries. This layer is essential in Ficus for constructing a stack which involves more than one site. For example, access to a remote replica is provided by inserting a vnode transport layer between a local logical layer instance and a remote physical layer instance, as in the right hand branch in Figure 1.

The Ficus replication layers also support a file system name service intended for use in a very large scale (nationwide) distributed system. Ficus builds upon an AFS-style *volume* abstraction for file management, NFS-style pathname resolution, *on-disk grafting* to glue volumes together in a hierarchy, and optimistic replication techniques to maintain consistency in the overall name space.

Ficus uses several asynchronous daemons to pro-

mote consistency among replicas. The most important of these are *update notification* and *update propagation* which enable replicas to learn rapidly of and incorporate new updates. Replicas which missed or unsuccessfully processed an update notification eventually learn about those updates from a *reconciliation* daemon that periodically performs a systematic sweep of its replicas' siblings.

Finally, the stackable layers paradigm and extensible vnode interface are a foundation upon which a number of interesting services can be constructed by varying the stack configuration. For example, an NFS layer can be inserted above the logical layer to allow access to replicated services by non-Ficus clients. New "value-added" layers such as transactions and encryption, as well as "invisible" layers (e.g., caching and measurement), can be added.

3.1 The Layers

Each of the Ficus layers adds a particular service, building upon the abstraction provided by lower layers. Here we outline the services provided by each of the key layers and briefly describe the extensible vnode interface.

Logical Layer

The primary function of the Ficus logical layer is to provide the illusion that each file is highly available with single-copy semantics, when in reality a file may be physically represented by multiple replicas whose individual availability is not optimal. The illusion is the product of several mechanisms, including *replica selection* and the consistency-promoting daemons described in a subsequent section.

Replica selection is guided first by consistency policies and second by performance considerations. Optimistic replication, like most other approaches to replication, must often choose which version of a file to use to servicing a file access request. Once a version decision has been made, file access performance differences may guide the final replica selection[3].

Unlike conventional replication mechanisms, optimistic replication has a greater range of version choices. Optimistic concurrency control and lazy update propagation yield a richer set of 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 further problem is that the appropriate version selection policy may well be client, application, instance, or data specific.

The Ficus approach is to provide a very general base level policy which can serve as the foundation for policies with different requirements. We anticipate the creation of a transaction layer, for example, which can be stacked above the Ficus logical layer to provide full transaction semantics. The current logical layer implementation gives priority by default to a local replica, falling back to a randomly chosen replica otherwise.

A number of open questions remain: How should consistency requirements be specified? What state should the file service (as opposed to the operating system) maintain about clients and their past requests? Should version and replica selection be done only at file open, or perhaps for each individual data access request? What versions may be substituted on failure to reach a previously accessible replica? Should a client always see a monotonically increasing sequence of versions with respect to time? What synchronization granularity (client, host, node cluster) should be used? Answers to these questions are the subject of ongoing research.

Physical and Persistent Storage Layers

The physical layer performs two main functions: it supports the concept of a file replica and it implements the basic naming hierarchy adopted by Ficus. The persistent storage layer underneath it provides basic file storage services.

The structure of the current Ficus physical layer re-

[3] One can imagine circumstances in which performance differences are so great as to make version issues a secondary issue. "Nearer before newer" may reasonably apply to utilities, for example, if the choice is between access to a local disk and access via voice-grade serial connection to a remote host.

flects an early design decision to use a standard UFS as the initial persistent storage layer, and beginner's inexperience with respect to the power, ease, and low cost of layering. From a strictly functional perspective, the "ideal" physical and persistent storage layer design is quite different, as we describe below.

The decision to use an unmodified UFS was driven primarily by the desire that Ficus replication layers be easy to "bolt on" to existing kernels, without causing significant disruption to system operation. A secondary factor was simple expediency: the UFS could clearly provide the minimal level of required storage service.

The physical layer's two main tasks, supporting replica-ness and the Ficus name space, proved to be rather poorly served by UFS services: support for replica-ness requires additional attribute storage (mostly for version vectors) beyond that provided by default, and the Ficus name space must allow for a limited directed acyclic graph topology (cf. UNIX tree) in some circumstances. The name space consistency mechanisms also required a small amount of additional storage per file name.

The additional richness inherent in the Ficus model led us to implement a full name resolution mechanism in the physical layer with the slight additional capabilities that were needed, as well as a file attribute service similar in spirit to the Macintosh operating system "resource fork." The UFS layer was relegated to providing a file service with an almost flat name space.

A third critical function of the current physical layer implementation is to map between Ficus low-level file identifiers and identifiers native to the persistent storage layer. Although Ficus file identifiers form a large flat name space which has a trivial UFS-compatible translation, the standard UFS name service is not well-suited for efficient flat name service. To overcome this efficiency problem, Ficus identifiers are mapped into a two-level UFS hierarchy which is carefully constructed to minimize linear searching and exploit expected file access locality patterns.

The current physical layer is somewhat monolithic and largely duplicates (while enhancing) services provided by the UFS persistent storage layer. In hindsight, a cleaner design would separate the UFS and physical layers into several additional layers, each providing a narrower set of services. Replication support can then take several forms: existing UFS storage services can be used as at present, when compatibility is a premium; or, a (new) persistent storage service with a richer file model but flat name space could be used when efficiency is very important. One might also construct a UFS-style layer for placement on top of the new persistent storage layer to provide a migration path away from the old monolithic UFS.

Extensible Vnode Interface

The symmetric layer interface used in Ficus is based on an encapsulated object called a *vnode*. A vnode represents a layer's concept of a file. It exports a set of operations which can be performed on the file, and contains a modest amount of private internal data which normally includes a pointer to another vnode. Multiple vnode types may be present in the system; vnode types differ in the structure of their private data, and how they support the set of operations on vnodes. There is a degree of "operator overloading" since the code that actually executes is dependent on the type of vnode to which the operation is applied and is bound dynamically at runtime.

Adding a new file system type primarily involves implementing each of the vnode operations for that type. That is, a vnode may have pointers to one or more lower level vnodes on which it is layered. The implementation of an operation on a vnode may simply forward the operation to one of its descendants (analogous to inheritance), perform some actions and then forward it to its descendants, call some other operations on its descendants, or even handle the operation entirely internally. Layers know nothing about the type of vnode below it; they simply hold a pointer to it. At the base of the stack is a layer which has no further descendants. For us, this is generally a UNIX filesystem.

In a layered file system, vnodes may be linked together. A stack for a given file is represented by a singly linked list of vnodes, of which only the first is known to the service above. There may be both fan-in and fan out in a stack. Fan in occurs when mul-

tiple vnodes point to a common lower level vnode. An example of this occurs in Ficus when several logical layers concurrently access the same replica. The physical level vnode for that replica resides in several stacks simultaneously. Fan out occurs when a single file at a higher level is represented by multiple files at the next layer down, such as in the Ficus logical layer which constructs multiple vnode stacks beneath it, one per replica. A base UNIX file system is an example of fan-in when it is NFS-mounted on several different sites.

The original vnode interface [8] incorporated a static set of exported operations. A recent explosion in the number of services and layers proposed and implemented has exposed the inability of the original operations set to satisfy the ever growing variety of desired functionality.

To address the shortcomings of static interfaces, we have developed an extensible vnode interface [5]. The extensible interface allows new operations to be exported with ease, which in turn has greatly simplified the task of creating and installing new layers. The new interface is also designed to support stacking to a greater degree than its predecessor.

A key component of this strategy is a new *bypass* operation which allows a layer to "pass on" an operation invocation which the layer does not support. In addition to the benefits of simple inheritance, the bypass operation allows new layers to be wrapped around existing layers which have no knowledge of new operations: when a layer does not recognize an operation invoked on a vnode, it redirects the invocation to the bypass operation, which passes the call down the stack, presumably to a layer which does support it. The bypass operation is of particular importance to the transport layer.

Transport Layer

The transport layer is essentially a remote vnode operation service which maps calls from the layer above to operations on a vnode at a layer below, normally crossing an address space boundary (and often a machine boundary) in the process. The ideal transport layer consists solely of a bypass operation, with no semantic interpretation beyond that necessary to marshal arguments. A transport layer can then be inserted between any two layers without regard for their semantics.

Ficus currently uses NFS as an approximation to an ideal transport layer for the same reasons given above for using UFS as a storage layer: a desire not to introduce an additional protocol when a widely used similar one existed, and to avoid re-inventions when possible. Here again, we learned that a powerful, "almost right" service is helpful in the beginning, but unsatisfactory later.

NFS achieves a certain quality of performance by interpreting, modifying, or intercepting some vnode operation invocations. A layer attempting to leverage NFS as a generic transport service must be constructed with substantial NFS internal details in mind. In our experience, it is certainly possible to use NFS as a transport service, but at the expense of building extensive mechanisms to defeat numerous internal NFS mechanisms.

The extensible vnode interface and bypass operation were developed in large part due to our unhappiness with the never ending set of counter-mechanisms we were employing to outsmart NFS. We have now installed the extensible interface, and used it to add a bypass routine to our NFS layer so that the counter-mechanisms could be discarded.

It is clear that in our model, NFS is best constructed as a pair of layers surrounding an ideal vnode transport service, not vice versa. Nevertheless, retaining NFS compatibility with non-Ficus hosts is important.

Examples of the power of layering, and the importance of standard NFS compatibility, are evident when considering the utility of an NFS layer placed above the logical layer or between the physical and UFS layers. Figure 2 displays several interesting possibilities. When an NFS layer is above the logical layer, an IBM PC running DOS and PC-NFS can access Ficus replicated files without being aware that replication is occurring. Similarly, configuring NFS below the physical layer allows sites on which Ficus does not run to act as replica storage sites. This arrangement would permit a Ficus site to store replicas

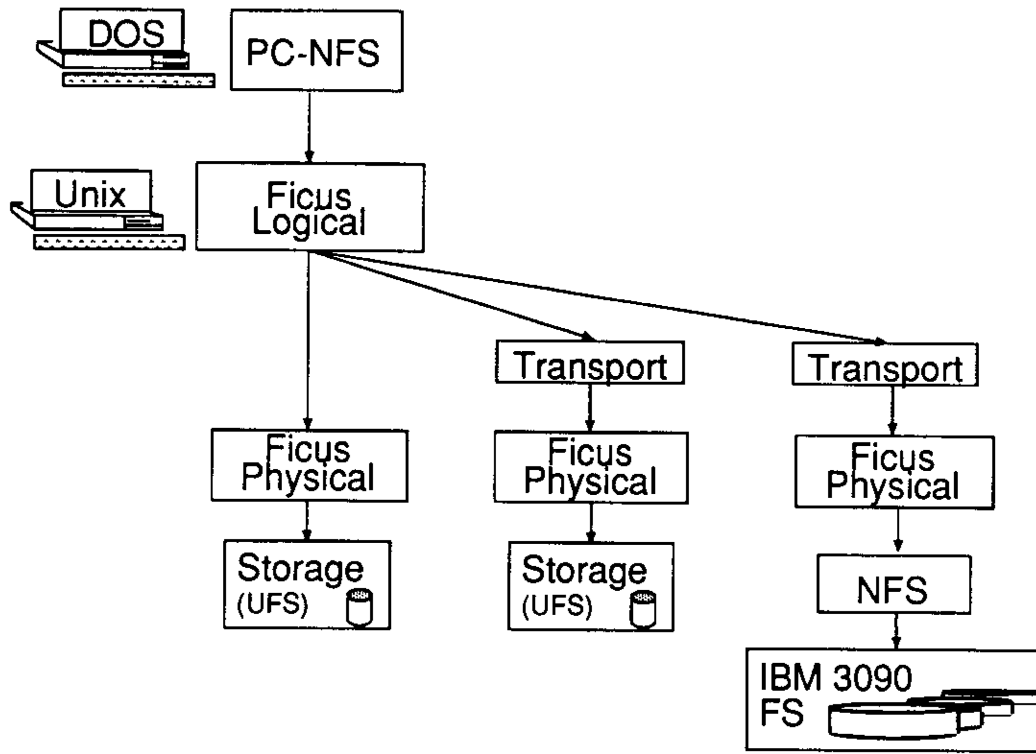

Figure 2: Using NFS to Interact with non-Ficus Hosts

on a mainframe running MVS and NFS.[4]

3.2 Consistency Mechanisms

The optimistic consistency philosophy allows considerable flexibility in many aspects of replicated file management. In addition to the richer choices encountered in replica selection, more options are possible when promoting consistency among replicas. Ficus uses three asynchronous daemons in an optimistic manner to notify replicas of updates, to propagate updates, and to ensure that eventual mutual consistency is attained.

The flow of control for file update notification and propagation is displayed in Figure 3 with further explanation below.

[4]This has been demonstrated using a non-Ficus SunOS host; logistics hinder an MVS demonstration of the concept.

Update Notification and Propagation

In Ficus, a file update is applied immediately to only one replica. When a write operation is received by the logical layer, it is forwarded to the physical layer vnode for the replica selected. After having been successfully applied to one replica, the logical layer may then notify other replicas of the update. The logical layer instance that handled the update places a summary of the update on an outgoing *update notification queue*, and then returns control to the client.

An *update notification daemon* periodically wakes up and services the queue, sending out notification to all accessible replicas that a new file version exists. Notification is a best-effort, one-shot attempt; inaccessible replicas are not guaranteed to receive an update notification later.

Ficus' reliance upon optimism releases it from the burden of ensuring that an update notification message is successfully delivered and processed by the

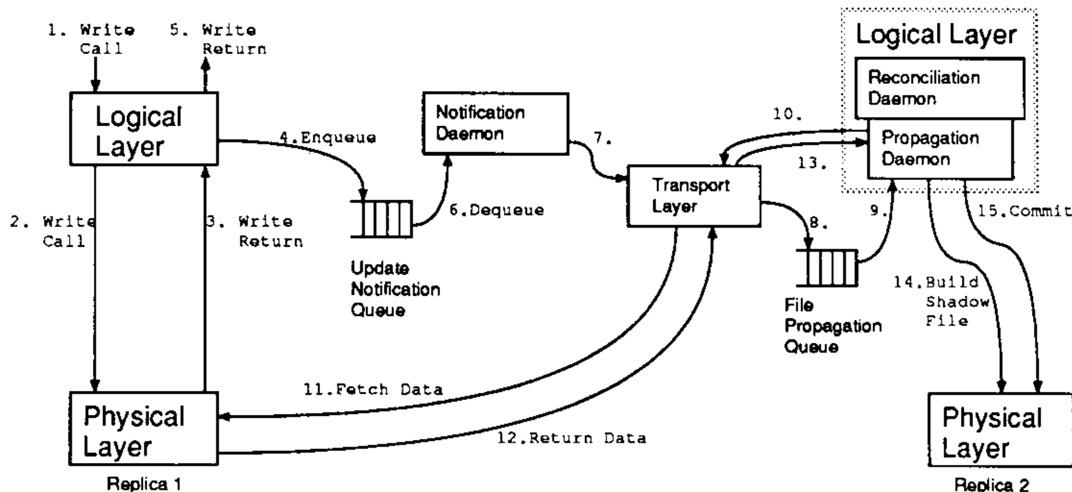

Figure 3: File Update Notification and Propagation

receiver. If the receiver fails to update its replica for whatever reason, it is assured that it will eventually learn of the update via the reconciliation daemon running on its behalf. (See discussion below.)

An update notification is not necessarily placed on the queue as part of every update. If the logical layer has received a "file open" operation, it will delay placing a notification in the queue until a "file close" operation is received. If for some reason a close operation never arrives, no update notification will be sent, which is analogous to a lost update notification message—perfectly acceptable by the optimistic philosophy, since reconciliation will find out about it later.[5]

The notification message contains the version vector of the new file version, and a hint about the site which stores that version. It is then the responsibility of the individual replicas to pull over the update from a more up-to-date site.

When a replica receives an update notification, it places that notification on a queue. An *update propagation daemon* wakes up occasionally to service the queue. The notification message sometimes contains complete information about the update (as is the case for directory updates), and if so it directly applies the update to its replica. If the notification only contains a summary of the update (typical for file updates), the replica which sent the notification must be queried for the new data.

Update propagation is performed atomically. Ficus contains an atomic commit mechanism used primarily by propagation to ensure that a replica's version vector properly reflects the replica's data. A shadow replica is constructed containing the new version, which is then substituted for the original replica.

No locking occurs during the construction of the shadow replica. The commit mechanism verifies that the shadow replica does not conflict with the original replica as part of the commit. It also verifies that the remote replica from which the shadow has been constructed has not changed during that time. If any changes have occurred, optimism allows update propagation to start over or abort the propagation.

[5] A logical layer that is servicing a remote client via NFS will never receive an open or close operation, so it may choose to issue an update notification for every write operation. Again, with optimism, this is all merely an optimization.

Reconciliation

The reconcilition daemon shoulders the responsibility of ensuring eventual mutual consistency between replicas. For each replica housed by a node, the reconciliation daemon directly or indirectly checks every other replica to see if a new (possibly conflicting) version of the file exists. When a new version is discovered, update propagation is initiated and follows the sequence of steps outlined above.

When reconciliation discovers a remote replica in conflict with its local replica, a conflict mark is placed on the local replica. A conflict mark blocks normal access until the conflict is resolved, at which point the mark is removed. Access to marked files is permitted via a special syntax.

Imperfect communication affects how the reconciliation daemon undertakes its tasks at two levels: the order in which local replicas are inspected, and the order and timing of contact with other replicas' nodes.

Is there a preferred ordering of local replicas which the daemon should follow when performing reconciliation? If communications between two nodes is likely to be interrupted at intervals less than the length of time required by the reconciliation daemon to scan through its local replicas and query the remote node, a fixed starting point must be avoided so that files at the far end of the order are not victimized by starvation.

The overall cost of reconciliation among a set of replicas is determined in part by the inter-node pattern in which reconciliation occurs. If each node directly contacts every other node, a quadratic (in the number of nodes) message complexity results, but if indirect contact (through intermediate nodes) is used in an optimal fashion, a small-coefficient linear complexity can be achieved. The interesting problem here is to exploit indirect reconciliation when inter-node communication is in excellent condition (to avoid quadratic complexity costs), but gracefully handle degraded communications when the degradation follows no predictable pattern and may be quite volatile. The Ficus solution uses a two-node protocol that tends to structure indirect communications in a ring topology when communications links permit, and dynamically adjusts to a non-fixed tree topology in response to changes in the communications service [2].

3.3 Name Space

The Ficus name space is a replicated hierarchy derived from UNIX with extensions motivated by large scale distribution and optimism. The optimistic consistency philosophy requires extending the name space topology to provide limited support for a directed acyclic graph, and the scale promoted new mechanisms for connecting portions of the name space.

Topology

The optimistic consistency philosophy in Ficus extends not only to updates to existing files, but also to the name space. Because name space consistency is essential to useful operation, and the semantics of name operations are well-understood, Ficus contains extensive support for automatic reconciliation of name space (directory) replicas. The algorithms used in name space reconciliation are described in a companion paper [3]; here we describe relevant issues from the client perspective.

Like UNIX, the Ficus name space hierarchy is constructed from a disjoint set of sub-hierarchies. Ficus sub-hierarchies (AFS-style volumes) are the primary management basis for replication. A logical volume is represented by one or more physical volume replicas. Each volume replica contains a replica of the volume's root directory; a volume replica may store any or all of the remaining files and directories of the logical volume.

In keeping with the optimistic consistency philosophy, any name space operation that leaves a volume replica internally consistent may be applied to a volume replica; the system (via automatic name space reconciliation) is responsible for propagating the change to other volume replicas and handling any difficulties that may arise out of concurrent activity.

Connection Mechanisms

The Ficus volume hierarchy topology is embedded within the volumes themselves using an on-disk mounting mechanism we call a *graft point*. A graft point is an internal volume object which contains the identity of the logical volume to be "grafted" (mounted) at that point, and the specific identity and location (internet host name or address) of volume replicas. Each host maintains a private table which maps locally stored volume replicas to particular local storage devices.

A graft point that is part of a replicated volume may be replicated just as any normal file or directory can be replicated. The replication factor (how many replicas and where located) of a graft point is independent of the replication factor of the volume to be grafted at that point.

The graft point mechanism serves to fragment and distribute the volume location database [6] or global mount table [12] found in other distributed file systems. This is critical to successful operation of a very large scale distributed file system in which less than perfect communication is routine and large numbers of administrative entities are involved.

We further note that the layering methodology allowed graft points to be implemented such that the Ficus physical layer believes a graft point to be a directory while the logical layer recognizes it as a graft point, and therefore the name space reconciliation mechanism already in place manages graft point updates[6].

Globally Unique Identifiers

A Ficus file is identified by a multi-part globally unique identifier. Its structure is designed in keeping with the optimistic philosophy: any Ficus host may create new logical volumes irrespective of current communications abilities; additional volume replicas may be spawned from any accessible volume replica; and any volume replica can create new files.

The identifier contains fields to specify which network, host, volume, and volume replica gave birth to the file. A further field is required when a particular file replica is specified. The 32-bit fields used in the Ficus prototype appear adequate to support a sizable distributed system.

Since each file must have a globally unique identifier, Ficus name resolution entails an additional level of indirection over that of UNIX. A Ficus directory entry refers to a unique file identifier, which in turn is mapped into a low-level name understood by the persistent storage layer.

Logical and Physical Layer Services

As indicated earlier, one of the logical layer's primary roles is file replica selection. This is turn requires the logical layer to identify and locate volume replicas—exactly the information that is stored in the graft point for the volume in question. Graft point interpretation and management is thus a logical layer function.

The physical layer is responsible for intra-volume name resolution and name space management. The Ficus directory structure is implemented entirely within the physical layer, so that only a simple flat-file model need be supported by the persistent storage layer.

3.4 Configuring Stacks

The SunOS file system mount mechanism has been overloaded in Ficus to serve as a stack construction service. Ficus stacks are static, in contrast to Rosenthal's dynamic stacks [15], and are established at a per-volume (or SunOS filesystem) granularity.

Overloading the existing mount service to configure stacks was a natural outgrowth of our use of the UFS and NFS modules. It produced many immediate benefits and met an initial design goal to leverage NFS and UFS as much as possible. We have used it

[6]When a volume replica is added, removed, or changes hosts, a graft point update (to one graft point replica) is required. Name space reconciliation handles propagation, etc.

to configure stacks with "invisible" measurement layers at various points, and have speculated about the value of other "invisible" layers to perform caching, to simulate network delays, or to produce Byzantine behavior.

While it has been very useful, it is also clear that a per-volume granularity unduly restricts the power of layering. For example, layers that provide data compression, encryption, or transaction services may be more appropriately applied at a per-file granularity. Indeed, one could conceivably construct a typed-file service primarily out of layers: store the order of desired layers as a file attribute, and configure the stack at file open time. We are now investigating a per-file stack construction methodology.

4 Implementation Status

The system as described in this paper, including reconciliation, is operational and in daily use. The system has been tested with up to eight replicas. Ficus replication has been run with geographically remote clusters; specifically volumes have been replicated at USC/ISI, SRI, and UCLA[7].

The user interface to resolving conflicting file updates and name conflicts detected and reported by the reconciliation mechanism is currently quite primitive. In practice, we have rarely observed conflicts that were not intentionally generated or could not be automatically repaired. This is due in part to the relative rarity of conflicting update in general. Nevertheless, this interface requires improvement to support non-expert users.

Operational layers include the transport layer, the logical and physical replication layers, a null layer, and a measurement layer. Prototype implementations of encryption, file versions, cache consistency, and second class replication layers will be completed shortly.

[7]USC/ISI is located in Marina Del Rey and connected to UCLA (after several gateways) via Los Nettos. SRI is located in Palo Alto, California and connected to UCLA via Los Netos to the San Diego Supercomputer Center followed by NSFnet. Thanks to Bob Balzer of ISI and Alan Downing of SRI for arranging for Ficus nodes at their respective institutes.

Ficus is based on Sun UNIX version 4.0.3, though ports to other versions of UNIX already supporting a VFS interface should be quite straightforward. The initial implementation (completed Summer 1989) used an unmodified vnode interface with extended operations supported by overloading existing operations. The current implementation uses the new extensible vnode interface described in [5]. Ficus kernels are about 20% larger than a similar non-Ficus kernel.

4.1 Performance Measurements

This section reports performance measurements for various configurations of replicated and unreplicated volumes. All measurements utilize Sun 3/60s, each with a SCSI disk and 10Mb ethernet connection. All nodes are part of the same ethernet segment, with the exception of the machines at ISI and SRI.

We report two benchmarks. The first is the modified Andrew benchmark [9, 6], designed to reflect a typical mix of file operations. However, since this benchmark is not particularly illustrative of replicated file system performance (it is dominated by the compilation phase which is largely cpu bound), we also report a second benchmark which is much more of a worst case measure of Ficus. The benchmark used is a recursive copy ("cp -r /usr/include .") of an NFS mounted file system (housed on a Sun-3/480 connected to the same ethernet segment)) to the local disk of a Sun-3/60s. In our environment, /usr/include is an unbalanced tree of depth four, with 47 directories and 1465 files totaling 4.7 Mbytes.

Figure 4 shows the overhead (compared to Sun OS without Ficus) for the modified Andrew benchmark as the number of replicas vary from 1 to 8. It also displays the the overhead in terms of system time and elapsed time required to perform the recursive copy benchmark. These are the costs incurred by the site generating the activity, which also stores a replica of the data.

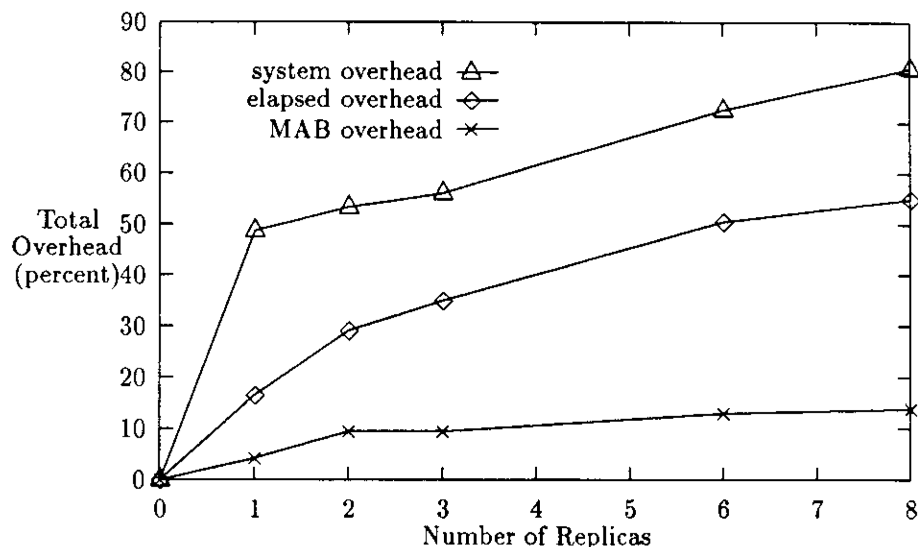

Figure 4: Overhead (as compared to UNIX) for the modified Andrew benchmark and the recursive copy benchmark.

Discussion

These measurements are very encouraging. For the modified Andrew benchmark, even with eight replicas, the overhead is only 14 percent. For the common case of three replicas, overhead is less than 10 percent. There is considerably more impact due to replication in the recursive copy case (between 30 and 50 percent). For each file copied, the system must place an entry in the directory (updating both the directory and its auxiliary information), create the file itself, placing its Ficus-specific attributes in the auxiliary file, notify all replicas of the directory operation to create the name for the new file, notify the replicas of the availability of the file's contents, and serve all of the asynchronous propagation requests as the replicas pull over the file contents. Copy is a worst case operation in terms of overhead for Ficus.

In interpreting these numbers, it is important to remember that Ficus applies an update synchronously to one replica and queues an "update notification" for asynchronous delivery to other "secondary" replicas. Each replica queues incoming update notifications and asynchronously processes the notifications. Directory update notifications completely describe the update, so no interrogation of the primary is needed to process a notification. File update notifications carry no data (only a version vector), so a "pull" is initiated by a secondary to bring its replica up to date. Data for updates is generally served out of the cache on the originating site.

The "system" times for all Ficus measurements are similar because the cost of asynchronous update notifications is in the background (there is some impact of increased replication factors reflected in the measurements as the interrupt handling for pull requests is included in system time).

The increased "elapsed" or "wall clock" time observed when more replicas are employed is attributed primarily to the cost of servicing requests from the the secondary in response to update notifications. It should further be noted that the recursive copy completes and the elapsed time is reported when it finishes synchronously updating the single chosen replica. It is generally the case that many of the remote replicas have not finished their asynchronous pull of the data. Thus the greater the delay in the network, or the slower the remote disks, the slower the requests arrive at the originating replica and hence the sooner the synchronous part completes. Hence, some of the numbers actually look better when the replicas are

further apart.

4.1.1 Long Distance Operation

In the performance graphs shown, all replicas resided on machines on the same physical ethernet cable. We repeated several of these measurements, this time locating replicas on sites connected by the Internet[8]. For the case of the recursive copy, locating one replica at SRI and one at UCLA yielded measurements for both system and elapsed time that were identical to the case where both replicas were on the same local network. Measurements of a three replica configuration (UCLA, ISI, and SRI) resulted in a 36% overhead over UNIX for elapsed time (vs. 35% for the local net case) and 48% overhead for system time (vs. 56% for the local net case). For the modified Andrew benchmark the long distance three replica configuration resulted in an overhead of 19.9% compared to 9.5% when the three replicas were local.

Not surprisingly given the asynchronous update strategy, locating the background replicas at more remote sites has minimal impact on the performance of these benchmarks. Of course, access to the remote replicas is correspondingly slower, equivalent to that achieved by accessing them with NFS.

Only very preliminary efforts have yet been made to optimize the performance of the implementation as work thus far has focused on functionality. There is reason to believe that the numbers reported here will improve substantially with careful analysis and optimization of the system's behavior (especially the effectiveness of its several caching mechanisms).

5 Conclusions

We draw several conclusions from the work reported here. First, all of our experience supports the view that optimistic replication is very attractive, whether it is merely between one's home computer and the office network, or in a very large corporate information system. High performance, high availability, scalable distributed computing service is feasible; we 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 optimistic replication reconciliation service can be retargeted beyond its initial use for directory management, as is shown by our success in using it to manage Ficus' replicated volume location tables.

Next, the use of stackable layers as the framework for the Ficus architecture has been an unqualified boon. The ability to leverage a common filing service directly permitted us to focus on development of new functionality inherent in the replication service, and avoid much of the traditional cost of building an ideal substrate at the outset. The modularity afforded by the architecture, along with the ability of the transport layer to map operations across address space boundaries, allowed us to develop and debug new layers in user space, and move them into the kernel only after they were working, which substantially simplified testing and debugging.

Finally, and perhaps most significant, 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.

Acknowledgements

The authors wish to acknowledge Dieter Rothmeier and Wai Mak for their important contributions to the implementation of Ficus.

References

[1] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael

[8] To avoid excessive retransmissions, the mounts across the Internet use 1K message block sizes where 8K messages are used over the ethernet.

Young. Mach: A new kernel foundation for UNIX development. In *USENIX Conference Proceedings*, pages 93–113. USENIX, June 1986.

[2] Richard G. Guy. *Ficus: A Very Large Scale Reliable Distributed File System.* Ph.D. dissertation, University of California, Los Angeles, 1991. In preparation.

[3] Richard G. Guy and Gerald J. Popek. Algorithms for consistency in optimistically replicated file systems. Technical Report CSD-910006, University of California, Los Angeles, January 1991.

[4] John S. Heidemann and Gerald J. Popek. An extensible, stackable method of file system development. Technical Report CSD-900044, University of California, Los Angeles, December 1990.

[5] John S. Heidemann and Gerald J. Popek. A layered approach to file system development. Technical Report CSD-910007, University of California, Los Angeles, January 1991.

[6] 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, February 1988.

[7] 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, January 1991.

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

[9] John K. Ousterhout. Why aren't operating systems geting faster as fast as hardware? In *USENIX Conference Proceedings*, pages 247–256. USENIX, June 1990.

[10] 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, May 1983.

[11] 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, May 1990.

[12] Gerald J. Popek and Bruce J. Walker. *The Locus Distributed System Architecture.* The MIT Press, 1985.

[13] Richard Rashid, Robert Baron, Alessandro Forin, David Golub, Michael Jones, Daniel Julin, Douglas Orr, and Richard Sanzi. Mach: A foundation for open systems. In *Proceedings of the Second Workshop on Workstation Operating Systems*, pages 109–113. IEEE, September 1989.

[14] Dennis M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[15] David S. H. Rosenthal. Evolving the vnode interface. In *USENIX Conference Proceedings*, pages 107–118. USENIX, June 1990.

[16] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, and Will Neuhauser. Overview of the CHORUS distributed operating system. Technical Report CS/TR-90-25, Chorus systèmes, April 1990.

[17] 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. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 35–50. ACM, December 1985.

[18] 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, April 1990.

[19] Alex Siegel, Kenneth Birman, and Keith Marzullo. Deceit: A flexible distributed file system. Technical Report TR 89-1042, Cornell University, November 1989.

[20] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. In *USENIX Conference Proceedings*. USENIX, Winter 1988.

[21] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proceedings of the Ninth Symposium on Operating Systems Principles*, pages 49–70. ACM, October 1983.