Computer Science Department Technical Report

University of California

Los Angeles, CA 90024-1596

FICUS: A VERY LARGE SCALE RELIABLE DISTRIBUTED FILE SYSTEM

Richard G. Guy

June 1991

CSD-910018

UNIVERSITY OF CALIFORNIA
Los Angeles

Ficus: A Very Large Scale Reliable Distributed File System

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Richard George Guy, II

1991

The dissertation of Richard George Guy, II is approved.

[illegible]

Charles Taylor

[illegible]

Michael K. Stenstrom

[illegible]

Richard R. Muntz

[illegible]

David Jefferson

[illegible]

Walter J. Karplus, Committee Co-Chair

[illegible]

Gerald J. Popek, Committee Co-Chair

University of California, Los Angeles

1991

Table of Contents

Acknowledgments

The path to the completion of this dissertation (and beyond) has been influenced by a number of mentors. These include my parents, who encouraged my adolescent interest in computing despite the strains it placed on our relationship at the time; Steve McClain, who thought it would be interesting to teach programming to a handful of pesky high school students; Vernon Howe, who demonstrated the difference that a gifted and dedicated college teacher can make in his students' growth; Hilmer Besel, who recognized that the ignorance I felt upon graduating with a bachelor's degree in computing was evidence that a college education had taught me something after all; Jerry Popek, who welcomed me into his research group and nurtured my intellectual development throughout the past decade; the LOCUS research team, especially Bruce Walker and Evelyn Walton, who introduced me to distributed computing theory and practice; and, finally, the Ficus research team, who have been a continuing source of enthusiasm, excitement, and encouragement. I am further gratefully deeply indebted to the generous support that the Defense Advanced Research Projects Agency has invested in computer science research,[1] without which I would be (ungratefully) deeply indebted to the federal Student Loan Marketing Association.

[1]**This research was sponsored under DARPA contract number F29601-87-C-0072**

Abstract of the Dissertation

Ficus: A Very Large Scale Reliable Distributed File System

by

Richard George Guy, II
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1991
Professor Gerald J. Popek, Co-Chair
Professor Walter J. Karplus, Co-Chair

The dissertation presents the issues addressed in the design of Ficus, a large scale wide area distributed file system currently operational on a modest scale at UCLA. Key aspects of providing such a service include toleration of partial operation in virtually all areas; support for large scale, optimistic data replication; and a flexible, extensible modular design.

Ficus incorporates a "stackable layers" modular architecture and full support for optimistic replication. Replication is provided by a pair of layers operating in concert above a traditional filing service. A "volume" abstraction and on-the-fly volume "grafting" mechanism are used to manage the large scale file name space.

The replication service uses a family of novel algorithms to manage the propagation of changes to the filing environment. These algorithms are fully distributed, tolerate partial operation (including nontransitive communications), and display linear storage overhead and worst case quadratic message complexity.

local area network services into the wide area network arena are network performance and scale.[2]

1.1 Network performance

From its inception, the ARPANET has primarily used transmission lines with much less bandwidth than an I/O channel on a typical host. The bandwidth ratio in the early 1970's was typically 1:200 (56 Kb/s ARPANET : 12 Mb/s UNIBUS); by the late 1980's, the ratio approached 1:4,000 as common workstation I/O bus rates exceeded 200 Mb/s (VME bus). The dramatic difference between I/O bus and network capacity forced network clients to be conscious of the network presence and conservative in its use.

In contrast, the 10 megabit per second local area network transmission rates of the early 1980's closely matched minicomputer I/O bus rates of the day. This similarity of bandwidth encouraged system designers to utilize network facilities in new ways, such as distributed file systems. The concept of *network transparency* became fundamental, much as virtual memory was accepted a decade earlier. Even with an order of magnitude difference in bus and network bandwidth today, local area network services continue to expand and be effective.

DARPA Internet sponsors are currently responding to the bandwidth disparity. A new Internet "backbone" with a bandwidth greater than one megabit per second is being installed in the United States. Over the next few years, the bandwidth will increase into the gigabit per second range as optical fibers replace copper wires as the primary transmission media. This realignment of network transmission capacity and host I/O bus rates lays a necessary portion of the foundation to provide wide area services that traditionally have been limited to local area networks.

Another important network performance parameter is latency induced by transmission delay. An inherent delay of 30 milliseconds is incurred by a transcontinental round trip message traveling at the speed of light. (The delay increases to $\frac{1}{2}$ second for geosynchronous satellite channels.) Services such as interprocess communication and remote procedure call for which the local latency is usually measured in microseconds are dramatically affected by large inherent latencies.

[2]In reflecting on Carnegie-Mellon University's network, Mahadev Satyanarayanan comments, "The change that would most substantially improve the usability of Andrew [CMU's campus-wide distributed system] would be a distributed file system that completely masked failures from users and application programs. It is still an open question whether this goal is achievable in conjunction with good performance and scalability." [Sat88]

multiply-linked network of communications controllers with an irregular topology. Adaptive routing algorithms are used to locate, evaluate, and select alternate communications paths. In two decades of operation, network partitioning has rarely occurred.[3].

The ARPANET solution is no panacea, however. Multiple (long-haul) links are quite expensive from an economic standpoint; in fact, the new Internet backbone is composed of just a few well-connected regional hubs that interface to area networks. Nevertheless, there continue to be single points of failure which can inhibit communication [Neu87]. A regional hub failure may isolate a significant portion of the network; a gateway or host failure may isolate just a few hosts, or prevent access to data managed by a failed or inaccessible host.

Although partial operation is unavoidable in a large scale, wide area network, its detrimental effects can often be minimized. A wide area file system is an example of a valuable service that can, but need not, be rendered impotent by partial operation. As with the ARPANET communications links example, redundancy at one or more levels is the primary tool for counteracting partial operation's negative consequences. This work proposes *data replication* as an affordable approach to cope with partial operation in a wide area file system.

1.2.2 Data replication

Data replication techniques combat the problems of partial operation by using redundancy to avoid a single point of failure (the data). Rather than accessing a specific single copy, a client accesses a varying subset of data replicas. The number of copies accessed depends on the consistency method used, but the probability of successfully accessing a subset which satisfies the consistency criteria is expected to be greater than or equal to the probability of successfully accessing a particular copy.

A number of data replication methods have been incorporated into local area network file system services: LOCUS [PW85], ROE [EF83], Isis [Bir85], PULSE [TKW85], Eden [PNP86], Guardian [Bar78], Saguaro [PSA87], Gemini [BMP87], GAFFES [GGK87], and a UNIX United variant [Bre86]. Most assume that the number of copies is small, and that they are stored among a small number of hosts. A further assumption is that failures are rare, or of suf-

[3]Three notable exceptions: the 1987 partitioning resulting from a single broken fiber despite 7-fold redundancy [Neu87], the 1980 complete network failure which occurred when a single bit was dropped from a widely propagated status word [Ros81], and the 1988 Internet worm catastrophe which prompted many sites to preemptively shutdown as a means of protection [Spa89]

services, for example, will likely seldom benefit from novel developments. The resulting hysteresis may even condemn both intertwined services to obscurity, rather than just the one. Good modular design mitigates this problem.

1.4.1.1 Target environment

The available pool of candidate cooperating installations is the DARPA Internet community, in which UNIX is the predominant software environment. Of the various UNIX versions in common use, SunOS (Sun Microsystem's UNIX implementation) is attractive. The primary attraction is an internal design that readily supports multiple filesystem implementations; a secondary benefit is that SunOS is widely used within the Internet.

The file system portion of the SunOS UNIX kernel is built around a generic interface known as a *virtual file system*. The interface hides details of the particular file system implementation which is handling a request. AN extended version of the virtual file system interface is used to add the experimental large scale file system service to standard versions of SunOS.

1.4.1.2 Leveraging via stackable layers

Since this research focus is large scale issues, and not file system services in general, it is desirable to leverage existing (and future) services wherever possible. Primary candidate services are raw disk management and network transport of file data.

The virtual file system interface enables file system services to be leveraged in an interesting way: a novel file system service can be added, which utilizes existing services as though it (the novel service) was simply a routine client (such as a system call). This organization results in a "stack of layers" in which each layer exports the same interface.

In addition to its leveraging benefits, the stackable layers concept allows transparent services to be added. For example, a generic file system performance monitoring layer can be "slipped in" between any two stacked layers sharing the virtual file system interface.

This work uses the stackable layers model as an aid to designing and implementing large scale file system services. The existing UFS (UNIX File System) and NFS (Network File System) services within SunOS are used to provide disk management and network transport services, respectively. A new layer (or layers) is constructed that supports the virtual file system interface, and which uses UFS

- local autonomy for name management
- compatibility with embedded names in existing software
- support for large numbers of hosts, clients, and files

Existing naming schemes fail to meet one or more of these goals.

A testbed environment composed of existing installations necessarily places some limitations on the design of a naming scheme. For example, the choice of SunOS as a base system implies that the name space syntax must be compatible with the UNIX naming scheme. These limitations are not significantly constraining, however. The tree-structured UNIX name space can be supported within the context of a more general model, such as a directed acyclic graph.

The goals of simplicity and support for large scale suggest that some form of hierarchy serve as the name space model. It is not immediately clear how to integrate a few million existing, disjoint instances of the UNIX name space into a hierarchy that meets all of the design goals. Major issues of name space design and supporting mechanism must be addressed.

1.4.2 Distributed access

The file system service must be able to locate a file and provide access to it based solely on a client-provided, location transparent file name. Every host supporting the large scale name space should be able to efficiently locate *any* file named within the name space, and then support low-overhead read and update access.

1.4.2.1 Volume mapping

Ideally, the logical organization of a name space is unrelated to the physical location of the files themselves. In practice, effective resource management is achieved by physically grouping files with logically related names, yielding what is sometimes called a *volume*.[4] A name space, then, is structured as a hierarchy of volumes, each containing a rooted hierarchy of file names.

In existing distributed file systems, the macro-organization of volumes is usually represented as a table of mappings between names and volume roots. The

[4]Volumes were introduced in the Andrew File System [HKM88]; they are used here in similar ways, but with important differences in implementation and detail.

1.4.3 Replication

Large scale distributed systems inherently possess two unpleasant characteristics, partial operation and significant communications latency. Data replication techniques have been applied to small scale local area systems to address partial operation and latency problems (although they are much less pronounced). This suggests that replication has the potential to ameliorate these problems in large scale systems.

Data replication methods maintain multiple copies of data, but strive to present the client with the illusion of a single, highly available file. Major issues include replica consistency, management algorithms, and client interface.

1.4.3.1 Consistency

Almost all existing and proposed replication mechanisms adhere to serializability[5] as a consistency definition. It is not clear, however, that enforcing strict serializability is appropriate for all, or even most, file usage in a large scale, wide area system.

Serializability protects applications from interfering with each other. Existing serializability enforcement techniques place substantial availability constraints on access, often trading off read availability requirements against update availability.

For example, the *primary copy* [AD76] strategy requires that all updates to a file be performed on a designated copy; other copies may be brought up to date in parallel with the designated copy, or at a later time. Read-only access may be serviced by any accessible copy. In this case, read availability is very high, but update availability is no greater than with a single copy.

The *majority voting* technique [Tho78] ensures that every update is applied simultaneously to (at least) a majority of replicas. Read access which is to be followed by a related update must involve at least a majority of replicas, to guarantee that the latest version of the file is read.

Quorum consensus [Gif79] is a generalization of the basic voting approach, in which replicas are *a priori* allocated fixed, but possibly differing quantities of votes, and read and update quorums can be assigned different thresholds. The sum of the thresholds must be one greater than the number of replicas, but the update threshold must always be greater than one half of the number of votes.

[5]Informally, serializability requires that all client actions be logically orderable in a serial, non-concurrent fashion. The actions themselves may be physically concurrent, but their logical order must be serial.

responding components of another vector. Otherwise, concurrent update activity has occurred.

Concurrent updates are assumed to represent conflicting unsynchronized activity until some semantic-knowledgeable agent declares the conflict to be resolved. For many concurrently updated files, only the client can determine how the conflict is to be resolved. But concurrent updates can also be applied to the name space itself. Since the name space is closely managed by the name service, the potential exists for automatic resolution of concurrent name space updates.

A name space reconciliation service must first determine what updates should be propagated to a particular replica, and then apply those updates to it. The major issue here is how the reconciliation service learns which updates have occurred.

One approach is to maintain replica-specific update logs. The reconciliation service compares replicas' logs, identifies those updates which haven't been applied to the replica in question, and applies them to the replica and makes an appropriate log entry. Log entries should be garbage collected when they are no longer relevant, i.e., when each replica has applied the update; an algorithm such as that used in Jefferson's Global Virtual Time [Jef85] might suffice.

Name space update logs bear a striking resemblance to the name space to which they refer. It is therefore interesting to consider coalescing or embedding the log into the name space. Major benefits include a reduction in space overhead and the reduced complexity of managing only one structure.

This research investigates a log-less approach in some depth and develops a new class of low-overhead garbage collection algorithms. The algorithms place few requirements on underlying communications services, in that the topology may be arbitrary and continuously changing; no ordering of replicas is needed; global storage requirements are quadratic in the number of replicas, with a small coefficient (a few bits per replica); worst-case message complexity is quadratic, with message length as a linear function of the number of replicas.

This work incorporates these new garbage collection algorithms into the name space reconciliation service for OCA. Further research is needed to extend the basic algorithms to support dynamic growth and/or reduction of the number of replicas. Early indications suggest that growth is fairly easy to support, and reduction is more difficult.

of the family of two-phase algorithms used in optimistic replica management (Chapter 3). Chapter 4 summarizes the research and the conclusions which can be drawn from it, and finally suggests future directions in which this research could readily proceed.

1.5 Related work

The primary collections of related work are found in distributed file systems literature and replica management research. A brief summary of significant research follows.

1.5.1 Distributed file systems

Several interesting distributed file systems have appeared in the last decade. They are reviewed here, along with relevant file access studies.

1.5.1.1 Important system implementations

The file systems mentioned here all support some form of *network transparency*. Each supports distributed access and *location transparent* naming. The extent to which *name transparency* is supported varies widely: it is guaranteed in some (e.g., LOCUS), and must be provided by convention in others (e.g., NFS).

- LOCUS
  The LOCUS [PW85] distributed operating system provides clients with the illusion of a very large, highly reliable single system. The (research) LOCUS filesystem provides extensive file replication services. Within a single partition, a very high degree of file access consistency is maintained. Concurrent, partitioned updates are tolerated, and detected later through the use of version vectors [PPR83].

  LOCUS is intended for small scale environments, such as that typified by office environments using a local area network. Each node regards the others as peers, independent of their functionality.

- NFS
  Sun Microsystem's Network File System is a distributed file system. Rep-

the 'natural' tree is the ***Ficus benghalensis***, commonly known as the Banyan tree, a fig species renowned for a wide-branching visible root structure seemingly grafted high onto a main trunk.

Deceit appears to clients to be an extension to an NFS environment. It also supports a file versioning mechanism similar to that found in the VAX/VMS operating system.

1.5.1.2 File access/placement studies

Three comprehensive file system activity traces have been performed, one in a commercial mainframe setting [Smi81] and two in university UNIX environments [OCH85, Flo86b]. These traces form the basis for a number of analyses [Smi81, OCH85, MB87, Kur88, Flo86b, Flo86a].

These studies generally showed that a few files receive a large portion of file accesses; most files accessed are small, and read sequentially in their entirety. A significant amount of working-set type locality was observed: a reference to a file was frequently followed by another reference to the same file, and with lesser frequency to a file located within the same file directory. Careful cache management strategies were shown to be extremely useful in reducing I/O and other file system overhead. Little file sharing occurred, with the exception of a few heavily accessed (for read) system files.

1.5.2 Replica management

In the past twenty years, dozens of published papers have described various replica management protocols. It is convenient to classify the protocols according to several criteria: failure mode assumptions, pessimism about concurrent updates, and mutual exclusion methodology. Specific replicated file system and directory management proposals are also discussed.

1.5.2.1 Limited failure modes

Early work on replica management assumed that inaccessible replicas had completely failed; in this model, communications failures "do not occur." Ellis' [Ell77] ring-oriented broadcast strategy utilized update history logs to recover failed replicas. Despite its limitations, various improvements have been proposed, and utilized in implementations. An improved atomic broadcast technique reduced the overhead [JB86] and was incorporated in the Isis kernel [Bir85].

Dav84, Wri83]. Optimal ex post facto analysis of updates to determine which should be backed out is an NP-complete problem [Dav84, Wri83], so some updates may needlessly revoked for the sake of efficient analysis.

Optimistic non-serializable approaches have been proposed for computer conferencing [Str81], commercial information retrieval services [ABG87], bulletin board [BJS86], databases [BK85, SKS86, GAB83, All83, Fai81], and file directories [FM82, PPR83, Guy87]. Most of these base correctness criteria upon the semantics of the data and the operations performed.

1.5.2.4 Replicated directory management

The *directory replication* problem has received particular attention because of the central role it plays is designing a highly reliable distributed file system. A wide range of directory replication mechanisms have been proposed, from serializable to non-serializable.

Fischer and Michaels [FM82] presented the first detailed examination of directory replication. They recast the problem as a replicated *dictionary* problem, to focus on the basic *insert* and *delete* operations common to each. Unsynchronized, concurrent directory modifications are resolved via timestamps.

Various inefficiencies in Fischer's work were addressed by Allchin [All83]. Wuu [WB84] offered further improvements. Unfortunately, the successive improvements reduced communications complexity at the expense of storage complexity: each replica in Wuu's scheme is required to maintain a version matrix.

Bloch [BDS84] utilized weighted voting in a serializable approach to directory replication. A novel scheme was adopted in which no single directory replica need (or can be assumed to) contain a true picture of the directory's status. Several replicas must be consulted to compose a correct view of the directory. Bloch's approach is inexpensive for interrogating or adding to a directory, but expensive for deletions.

Guy [Guy87] proposed a non-serializable solution in the spirit of the LOCUS system. It is based on Parker's version vectors [PPR83], and supports a wider range of semantics than the earlier work by Fischer, et al. A novel feature of this method is that it tolerates conflicting updates after the conflict is discovered. (Earlier techniques made an immediate decision on how to resolve the conflict; the haste often resulted in unpleasant effects, such as loss of a newly created file.)

Symbolic file system	File directory manipulation
Basic file system	File meta-data access and management
Protection	Access control verification
Logical file system	Access methods and file structure
Physical file system	Logical to physical address mapping
Device strategies	Initiate I/O, manage buffers, allocate storage

Figure 2.1: Madnick's hierarchical file system design (top-down).

vices into discrete levels of abstraction in which each level communicates only with its immediate upper and lower levels. The resulting static organization provides a framework for implementing a file system, such as the six-layer file system design proposed in Madnick and Donovan [MD74] (see Figure 2.1). The argument presented by Madnick, *et al* for modularity is the traditional one of complexity management, both for ensuring logical completeness and for debugging and quality assurance.

Operating systems have traditionally supported exactly one file system. Sun Microsystems' SunOS[1] implementation of UNIX[2] incorporates a switch which allows multiple file system services to co-exist comfortably within a single operating system.[3]

The Virtual File System (VFS) switch mechanism [Kle86] is designed around a *vnode* data structure which implements a stylized interface between a file system and other portions of the operating system kernel. The interface is essentially an information hiding technique that exports a set of operations (analogous to methods in object-oriented terminology); all data is private to some vnode, and can only be accessed via supplied vnode operations.

The apparent initial motivation for the VFS switch was to support transparent remote file access without embedding remote access mechanisms within the existing (local access) file system implementation [Kle86]. In SunOS, local filing service is handled by the UNIX File System (UFS), while remote access is pro-

[1]SunOS is a trademark of Sun Microsystems, Inc.

[2]UNIX is a trademark of AT&T.

[3]A number of other UNIX implementations now also provide file system switches. These include AT&T's File System Switch in UNIX System V Release 4, Digital's Generic File System for ULTRIX [RKH86], and the 4.3-Reno BSD (Berkeley Standard Distribution) [KM86].

explicitly for simplifying the implementation of network protocols. Primary features of the x-kernel include a uniform interface to all protocols, late binding between protocol layers, and a light-weight layer mechanism.

The x-kernel design is deliberately intended to encourage composition of protocol stacks on-the-fly, with run-time selection of layers supplying the appropriate semantics. Layering efficiency is provided by using procedure calls, not context switches, to pass information between layers. This economy is further used to promote decomposition of protocols into multiple layers as a means of aiding flexibility.

2.1.2 Observations

Operating systems and file systems have grown significantly in size and complexity in the twenty years since Madnick's proposal was presented. Greater complexity has often manifested itself as a tendency towards monolithic system implementations which increasingly defy adequate testing and verification, and hinder improvements—especially those not conceived within the monolithic framework.

In recent years, micro-kernels in systems such as Mach [ABG86, RBF89] and Chorus [RAA90] have emerged as a response to the monolithic implementations of operating systems. However, little has been done to tackle the monolithic filing service portion of most kernels. Adding any new features to a filing environment is usually a daunting task which frequently requires 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, and even then artificially limits the services which will be offered to those which are easily added to the existing product.

The file system switch approaches are an important step in the modular direction, but have yet to be accompanied by a decomposition of monolithic file system services. A few interesting new services (e.g., RAM-based filing) have been constructed, but all are layered onto a fairly standard UFS-type file system base. The new services are therefore constrained by the services and semantics offered by a UFS: access to lower level services within the UFS is not provided, and services must be used even when their richness is unnecessary.

It is also difficult to experiment with new lower level services. For example, Rosenblum's *log-structured* file system [RO91] for the Sprite operating system [OCD88] provides UFS semantics, but uses a very different disk management algorithm from that used by Berkeley's Fast file system [MJL84]. The monolithic

Well-defined operations: The extensibility and address space boundary quirements imply that a layer may need to handle arguments for "unknow operations. These arguments must be self-describing as to type, etc.

Stack fan-in, fan-out: A stack may actually be an acyclic graph, with fan and fan-out at any level. Fan-in occurs when several higher layers stack o common lower layer (as might occur when a file is concurrently accessed several nodes); fan-out occurs when a layer is stacked upon multiple low layers (as might be the case when several files stored on distinct nodes a in use by a single client).

Scale: No artificial limits should be placed on the depth (number of layers) o stack or on the breadth of fan-in or fan-out.

Dynamic composition: Stacks should be customizable on-the-fly, especial when the layers to be added/removed are "invisible" ones, i.e., they a semantics-free value-added layers, such as caching or monitoring layers.

Backwards compatible: Compatibility layers should be constructible whi can encapsulate the new mechanism within the constraints presented l older layering services. A lower compatibility layer allows the new mech. nism to leverage services implemented with other layering mechanisms; a upper compatibility layer allows older mechanisms to utilize new service subject to limitations imposed by the intersection of the various layerin mechanisms. In particular, existing system call interfaces must not be a fected, so that current application software will continue to run unmodifie

2.1.4 Ficus layer mechanism

The Ficus layering mechanism meets most, but not all, of these requirement It is derived from the SunOS VFS/vnode technique, but is much more flexibl and powerful than its predecessor. The mechanism is described in detail by it designer in [HP90, HP91a]; the material that follows summarizes that work.

A Ficus layer is defined to be a set of operations which can be applied to a vnode data structure. A *vnode structure* embodies a layer's concept of a file. It normally contains a list of operations which can be applied to the vnode, and a pointer to a procedure which implements each operation. The private data held by a vnode is layer specific, of course, but usually contains one or more pointers to vnodes from the layer immediately below. Adding a new file system service is primarily a matter of implementing the desired operations for a new layer.

A vnode operation invocation contains a pointer to the vnode on which to operate, the name of the operation, a pointer to the operation's arguments, and a template which defines the type of each argument. The layer mechanism maps the operation name to the proper implementation of the operation for the indicated vnode, and invokes the operation with the supplied arguments and template.

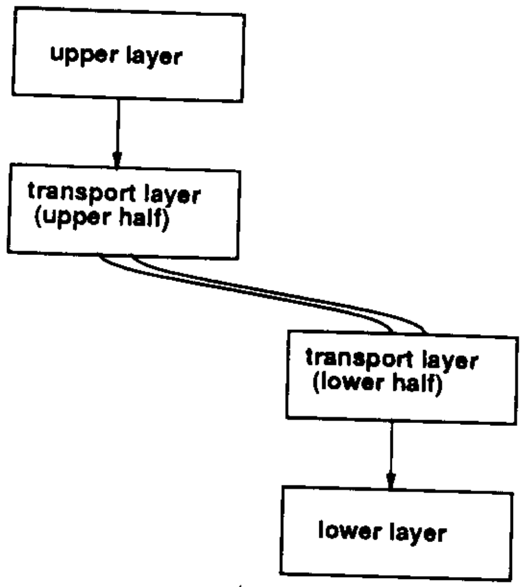

Figure 2.2: Stack with transport layer.

The generic transport layer maps every operation into a transparent bypass operation, that is, one that simply transports an operation's arguments and results.

2.1.4.2 Stack formation

A Ficus stack is defined by the graph formed by vnode-to-vnode pointers. Non-linear stacks with fan-in and fan-out at various layers are commonly seen in remote access and replication services. (For example, see Figure 2.6 on page 40.)

The "original" stacks in SunOS are constructed as a side-effect of the file system *mount* service, whose primary function is to glue together disjoint portions of the filing service name space. For expediency, Ficus overloads the mount service to stack one layer upon another. (The mount service side-effect is to cause one vnode to point to another.) Ficus stacks are therefore constructed bottom-up.

An unfortunate result of overloading the mount service in this manner is that current Ficus stacks are defined *a priori*, static, and apply at a volume granularity. Further mechanism must be constructed to store a stack description at a file-by-file granularity, which in turn could serve as the basis for a typed filing service.

The static nature of the current mechanism is not ideal. Some services, such as caching or encryption, may be most usable in a dynamic fashion: clients

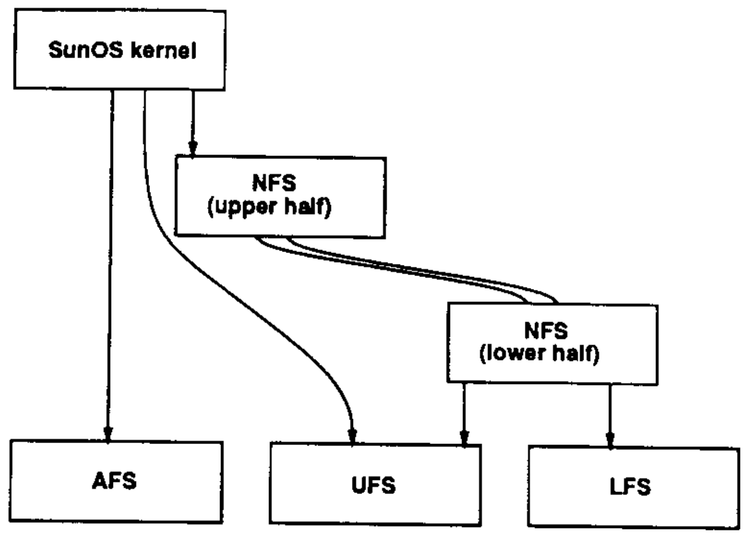

Figure 2.3: File system configuration with thick layers.

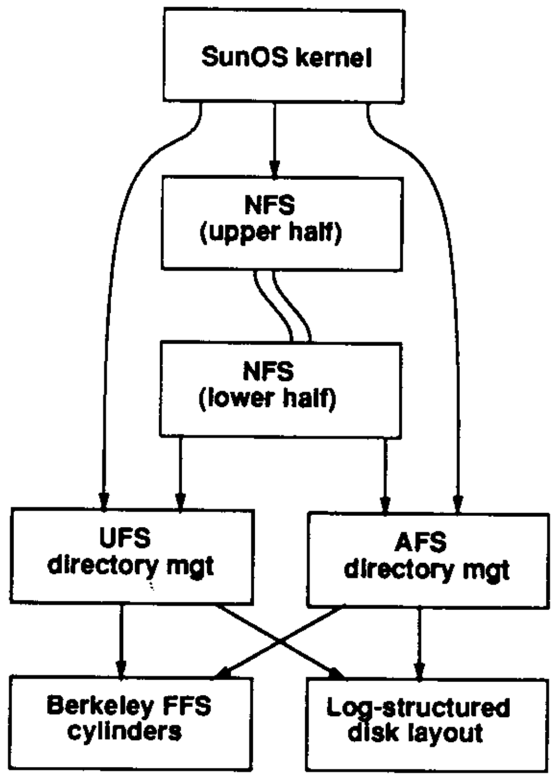

Figure 2.4: File system configuration with thinner layers.

that point in the hierarchy.

The traditional UNIX mounted filesystem mechanism has been widely altered or replaced to support both small and large scale distributed file systems. Examples of the former are Sun's Network File System (NFS) [SGK85] and IBM's TCF [PW85]; larger scale file systems are exemplified by AFS [Kaz88], Decorum [KLA90], Coda [SKK90], and Ficus [GHM90, PGP91]).

In a conventional single-host UNIX system, a single mount table exists which contains the mappings between the mounted-on directories and the roots of mounted volumes. However, in a distributed file system, the equivalent of the mount table must be a distributed data structure. The distributed mount table information must be replicated for reliability, and the replicas kept consistent in the face of update.

Most distributed UNIX file systems to some degree attempt to provide the same view of the name space from any site. Such *name transparency* requires mechanisms to ensure the coherence of the distributed and replicated name translation database. NFS, TCF, and AFS each employ quite different approaches to this problem.

To the degree that NFS achieves name transparency, it does so through convention and the out-of-band coordination by system administrators. Each site must explicitly mount every volume which is to be accessible from that site; NFS does not traverse mount points in remotely mounted volumes. If one administrator decides to mount a volume at a different place in the name tree, this information is not automatically propagated to other sites which also mount the volume. While allowing sites some autonomy in how they configure their name tree is viewed as a feature by some, it leads to frequent violations of name transparency which in turn significantly complicates the users' view of the distributed file system and limits the ability of users and programs to move between sites. Further, as a distributed file system scales across distinct administrative domains, the prospect of maintaining global agreement by convention becomes impossible.

IBM's TCF, like its predecessor Locus [PW85], achieves transparency by renegotiating a common view of the mount table among all sites in a partition every time the node topology (partition membership) changes. This design achieves a very high degree of network transparency in limited scale local area networks where topology change is relatively rare. However, for a network the size of the Internet, a mount table containing several volumes for each site in the network results in an unmanageably large data structure on each site. Further, in a nationwide environment, the topology is constantly in a state of flux; no algorithm which must renegotiate global agreements upon each partition membership in Ficus is represented by a set of volume replicas which form a maximal, but extensible, collection of containers for file replicas. Files (and directories) within a logical volume are replicated in one or more of the volume replicas.[9] Each individual volume replica is normally stored entirely within one UNIX disk partition.

Ficus and AFS differ in how volume location information is made highly available. Instead of employing large, monolithic mount tables on each site, Ficus fragments the information needed to locate volumes and places the data for an individual volume in a *graft point* (the mounted-on directory).[10]

2.2.3 Graft points

A graft point (see Figure 2.5) is a special file type used to indicate that a (specific) volume is to be transparently grafted at this point in the name space. Grafting is similar to UNIX filesystem mounting, but with a number of important differences.

A graft point maps a set of volume replicas to hosts, which in turn each maintain a private table mapping volume replicas to specific storage devices. Thus the various pieces of information required to locate and access a volume replica are stored where they will be accessible exactly where and when they will be needed.

A graft point contains a unique volume identifier and a list of volume replica and storage site address pairs. Therefore, a one-to-many mapping exists between a graft point replica and the volume replicas which can be grafted on it. Each graft point replica may have many volume replicas grafted at a time. The particular volume to be grafted onto a graft point is fixed when the graft point is created, although the number and placement of volume replicas may be dynamically changed.

A graft point may be replicated and manipulated just like any other object (file or directory) in a volume. It can be renamed or given multiple names; it can be a replicated object itself, with replication parameters independent of the referenced volume. Since a graft point resides in a "parent" volume, although referring to another volume, the graft point is subject to the replication constraints of the parent volume. There is no requirement that the replication factor (how many replicas and their location in the network) of a graft point match, or even overlap,

[9]Each volume replica must store a replica of the root node; storage of all other file and directory replicas is optional.

[10]In the sequel, the term "graft" and "graft point" is used for the Ficus notion of grafting volumes while the mount terminology is retained exclusively for the UNIX notion of mounting filesystems.

that of the child volume.

As it happens, the format of a graft point is compatible with that of a directory: a single bit indicates that it contains grafting information and not file name bindings. The syntactic and semantic similarity between graft points and normal file directories allows the use of the same optimistic replication and reconciliation mechanism that manages directory updates. (See Section 2.3 and Chapter 3 for details of these mechanisms.) Without building any additional mechanism, graft point updates are propagated to accessible replicas, conflicting updates are detected and automatically repaired where possible, and reported to the system administrators otherwise.

Volume replicas may be moved, created, or deleted, so long as the target volume replica and any replica of the graft point are accessible in the partition (one copy availability). This optimistic approach to replica management is critical as one of the primary motivations for adding a new volume replica may be that network partition has left only one replica still accessible, and greater reliability is desired.

This approach to managing volume location information scales to arbitrarily large networks, with no constraints on the number of volumes, volume replicas, changes in volume replication factors, or network topology and connectivity considerations.

2.2.4 Volume and file identifiers

When a file is created, it is given a globally unique, static identifier that is carried by the file (its replicas) throughout its existence. A Ficus file identifier has several components, but at its highest level of abstraction, it is a tuple ⟨*volume-id, file-id*⟩. The *volume-id* component is a globally unique identifier for the volume, while the *file-id* component is unique within that volume.

Partial network operation should not hinder a host's ability to create new volumes, so each host must be able to issue new *volume-ids* on its own. Prior to system installation, each Fiçus host is issued a unique value as its *allocator-id* which the host can use in conjunction with a non-decreasing counter to issue globally unique *volume-ids*. A *volume-id* is, therefore, a tuple ⟨*allocator-id, counter-value*⟩.

Allocator-ids are issued by a central (possibly offline) service. Ficus *allocator-ids* contain space for two fields the size of an Internet host address. In most cases, one field will contain existing Internet host addresses; the other field is present to allow easy integration of existing host identifiers from other networks.

If the volume in which the graft point resides is itself a replicated volume, the graft point containing the volume replica location information may also be replicated. If each parent volume replica which stores the directory in which the graft point occurs also stores a copy of the graft point, the location information is always available whenever the volume is nameable. There is very little benefit to replicating the graft point anywhere else and considerable loss if it is replicated any less.

In the course of expanding a path name, a directory is first checked to see if it is actually a graft-point. If so, and a volume replica is already grafted, pathname expansion simply continues in that volume replica's root directory. More than one replica of the grafted volume may be grafted simultaneously, but if no grafted replica is found, the system must autograft one of the volume's replicas onto the graft point.

A graft point is a table which maps volume replicas to their storage site. The sequence of records in a graft point table is in the same format as a standard directory and hence may be read with the same operations used to access directories. Each entry in a graft point is a triple of the form ⟨*volume-id, replica-id, hostname*⟩, identifying one replica of the volume to be grafted. The *volume-id* is a globally unique identifier for the volume. The *replica-id* identifies the specific volume replica to which the entry refers. The *hostname* identifies the host which is believed to house the volume replica.[11] The system then uses this information to select one or more of these replicas to graft. If the grafted volume replica is later found not to store a replica of a particular file, the system can return to this point and graft additional volume replicas as needed.

In order to autograft a volume replica, the system calls an application-level graft daemon on its site. Each site is responsible for mapping from volume and replica identifiers to the underlying storage device providing storage for that volume. If the *hostname* is local, the graft daemon looks in the file `/etc/voltab` for the location of the underlying volume to graft. If the *hostname* is remote, the graft daemon obtains a file handle for the remote volume by contacting the remote graft daemon (similar to an NFS mount; see [SGK85]) and completes the graft.

The system caches the pointer to the root directory of a grafted volume replica so that the graft point does not have to be fully reinterpreted each time it is traversed. (The cache supports pointers to multiple replicas for a single volume.) The graft of a volume replica which is not accessed for some time is automatically

[11]Currently *hostname* is an Internet host address; it could equally well be a Domain Naming System identifier or one from any other host naming or addressing mechanism.

id, repl-id) is encoded as printable ASCII characters in the name field of the directory entry, while the *hostname* component constitutes the contents of the named file.

When a new child volume replica is created, an entry for the new replica (consisting of a new file and associated directory entry) is placed in one graft point replica. The Ficus file and directory automatic update propagation service sees to it that the new graft point data is propagated to all other graft point replicas.

When a volume replica is moved from one host to another, the file contents (at one graft point replica) are simply updated to reflect the new *hostname*. Should a graft point entry be concurrently updated in two graft point replicas, the normal file update conflict detection mechanism will notice that state of affairs and flag each replica in conflict. Similarly, if a graft point entry is updated in one graft point replica (to indicate that the child volume replica has moved to a new host), and concurrently deleted in another graft point replica (to indicate that access to the child volume replica is no longer possible for some reason), the regular file remove/update conflict detection mechanisms detect the conflict. Standard (manual) conflict resolution tools must then be used to resolve the conflict. In the meantime, access to the file data (i.e., the *hostname*) will be blocked as usual for conflicted files.

If a volume replica is destroyed, the graft point entry is eliminated by simply removing the directory entry for that particular volume replica in one graft point replica. Update propagation and directory reconciliation ensures that the other graft point replicas also receive notification of the change.

In general, graft point data is self-validating upon use: if it is wrong in some way (perhaps the replica has been destroyed or moved to a new host) the queried host will respond negatively to a graft request, and the autograft mechanism will try some other replica.

2.2.8 Volume summary

The Ficus volume design is the foundation for a very large scale file system name service. It supports a very large number of volumes and volume replicas, in a flexible manner. Autonomous control of volume creation, placement, and destruction is inherent, as are location transparency and name transparency.

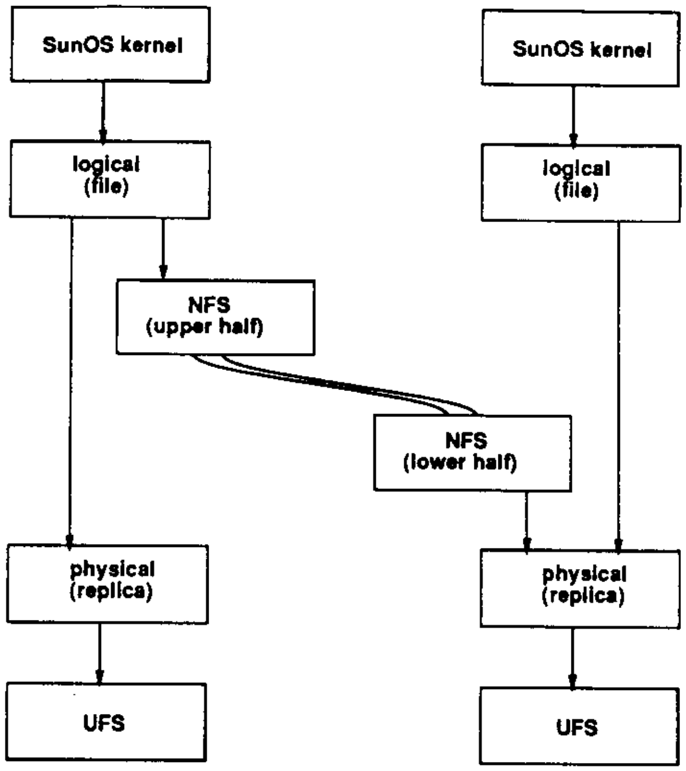

Figure 2.6: Typical Ficus layer stack.

The early prototype of implementation of the replication layers was largely successful in regards to leveraging, but demanded an ever-increasing investment in techniques to compensate for features not anticipated by the UFS, NFS, and VFS designers. In response to this problem, the design and implementation of Ficus has moved steadily towards the development of services better attuned to replication in particular, and stackable layers in general. The design described here is an enhanced version of an earlier one; it reflects the experience gained with layering and leveraging. Current implementation status is indicated as the discussion proceeds.

The Ficus file replication service is packaged as a pair of stackable layers, each building upon the abstractions provided by lower layers. Figure 2.6 shows a typical Ficus layer stack.

The *logical* layer provides to its clients (i.e., 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 with full transaction semantics.

2.3.2.1 Replica selection

Three issues determine which, if any, of accessible replicas appropriately serves the client: consistency policy, cost of replica access, and cost of the selection process itself.

Replica selection primarily occurs at file open, that is, when the logical layer performs a `lookup` operation for a client. If the client specifies a particular replica, file version, or minimum file version, the logical layer will strive to locate a qualifying replica. An error is returned if no appropriate replica can be accessed.[13]

If a particular replica or version is not specified, the logical layer consults a version cache indicating the greatest version of each file opened by the layer.[14] The cache value (if present) is used as an advisory minimum value: if no compatible replica is accessible, an accessible replica will be selected.

Except when a particular replica is specified, replica selection must decide in which order to consult replicas for compatibility and eventual service to clients. The most important factor is nearness (cost of access), but transparency at several levels in Ficus makes it difficult to distinguish degrees of nearness or cost. It is relatively easy, however, to determine if a physical layer is on the same host as the logical, so a crude distinction between local and not-so-local can be made. Ficus exercises a preference for local replicas.

After nearness, Ficus currently uses a random order to consult replicas. Replica selection ceases with discovery of a compatible replica, even though some other (unconsulted) replica with a greater version may exist and be accessible.

Composing a list of file replicas to consult during replica selection is a bit complex because Ficus supports dynamic, selective replication of volumes and files. A volume may be replicated any number of times, and stored by arbitrary hosts. The number and placement of volume replicas is dynamic. Similarly, the number of file replicas is dynamic within the constraints of the replication parameters of the volume that contains the file. (More details may be found in Sections 2.2 and 2.3.3.)

The physical layer ensures that a (minimally skeletal) file replica can always be accessed for a nameable file. A skeletal replica contains a *replica list*, which replica selection uses as a starting point. (Because replica placement is dynamic,

[13]Replica selection tries to access a replica no more than once per replica, per selection.

[14]The cache is volatile, large, but finite sized.

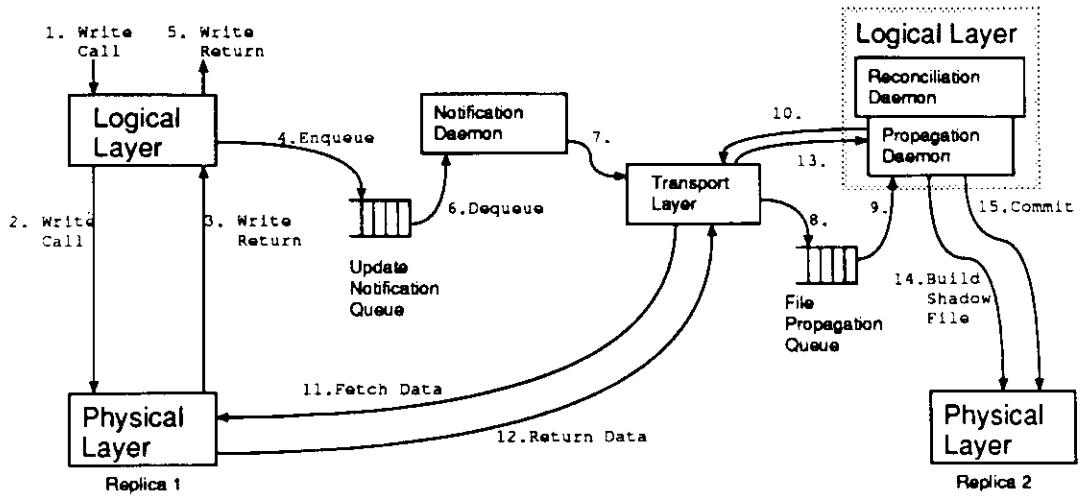

Figure 2.7: File update notification and propagation.

An *update notification daemon* periodically wakes up and services the queue, sending out notification via multicast datagram to all replica storage hosts 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 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 an **open** operation, it may delay placing a notification in the queue until a **close** operation is received. If for some reason a close operation never arrives, an update notification might not 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.[16]

The notification message contains the version vector (see Section 2.3.3) 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

[16]A logical layer that is servicing a remote client via NFS (see Section 2.3.4.3) will never receive an **open** or **close** operation, so it may choose to issue an update notification for every write operation. With optimism, this is all merely an optimization.

every local replica is reconciled within finite time, directly or indirectly, with every other replica of that file. Since a volume has a natural tree-like structure at one level, and a linear structure internally at a lower level, the "obvious" approach for reconciliation is to reconcile files in one or another of the convenient orders. Strict adherence to an order, however, is not robust to communication failure.

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. Further, reconciliation must not be hamstrung propagating a large file that has been updated, but whose pages cannot be transferred during an operational interval. Reconciliation must either reconcile files in a random order, choose a random restarting point each time, or be able to skip over problematic files.

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 automatically adjusts to a dynamic tree topology in response to changes in the communications service.

The current Ficus reconciliation implementation is somewhat simpler than the above design. It performs a breadth-first walk of a volume replica (always beginning with the root directory), communicating with a single other replica per walk. Reconciliation also steps through the volume replica list in a round-robin fashion, without considering the currency of results from recent reconciliations with other replicas.

2.3.3 Physical layer

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 reflects an early design deconcurrently at distinct replicas [Guy87].

In Ficus a directory update is either an **insert** or **delete** of a directory entry; **rename** is **insert** followed by **delete**. Concurrent directory entry insertion may result in more than one entry with identical name fields. Ficus tolerates this violation of directory semantics (all names are unambiguous within a single directory) by setting the *conflict* flag on entries with ambiguous names. This flag blocks normal name resolution pending client resolution of the conflict; it can be ignored by request.

The primary issue faced by the directory reconciliation mechanism is ascertaining which entries have been inserted into the directory replicas, and which have been deleted. To solve this problem, known as the *insert/delete ambiguity* [FM82], Ficus incorporates a logical deletion flag (the *delete* field) and a two-phase algorithm to garbage detect logically deleted entries. The algorithm, fully described in Chapter 3, uses the *bitvector* field to record its progress.

When a file is concurrently renamed, several new names result: each **rename** instance inserts a new entry, and marks the old entry logically deleted. For normal files, no conflict results even though the old entry may be "deleted" more than once. (Repeated entry deletion is not possible in UNIX) A conflict does emerge for concurrent directory rename, however.

Like UNIX, every Ficus directory contains a special entry (..) that references its parent directory. A directory rename, of course, must cause a .. entry to refer to the proper parent directory. Ficus accomplishes this task by deleting the old .. entry and inserting a new one with the correct referent. Concurrent directory rename results in a directory with multiple parents (one for each **rename** instance), and multiple ambiguous .. entries in the directory itself, each referring to the respective parent directory. The .. entries are marked in conflict, as usual, so backward pathname resolution will fail with a name conflict error until the conflict is resolved by removing all but one of the new directory names. Forward resolution is not affected by ambiguous .. entries.

2.3.3.3 Extended UFS abstractions

The Ficus directory structure does not map cleanly onto a standard UNIX file system due to the additional directory entry fields used in reconciliation and the possibility of multiple directory names at one time. Ficus also requires a few additional file attributes, such as the replication list and version vector, which do not comfortably fit within the space normally reserved for attributes (the UFS *inode*). Yet, an early design goal to stack upon an unmodified UFS remains

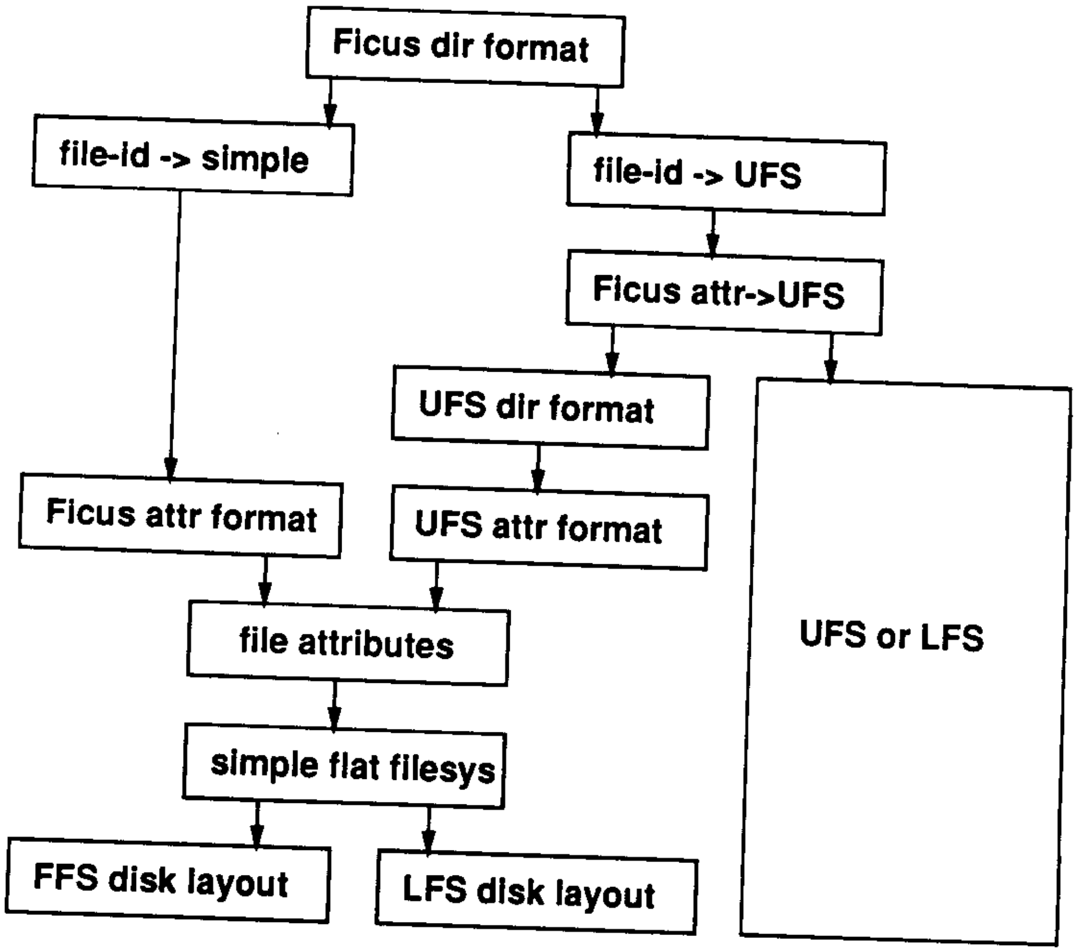

Figure 2.8: Decomposed physical and storage layers.

use of a *control file*, i.e., a pseudo-file with a distinguished name known to both layers as the channel over which version vectors are to be passed. NFS caches file data blocks, so a repeated read on the control file frequently returned the cached block without actually querying the next layer—the required course of action. To defeat caching, a mechanism that cycles read request offsets through a large cycle was employed. This was not immediately successful, as NFS maintains a notion of file size so that it can optimize for read requests to non-existent data pages by simply returning a null page in response to such a read request, rather than generating network traffic to transfer a "known" data page. The control file mechanism had to be repaired so that it always reported to NFS that the file "size" was larger than the cycle size in use.

The extensible vnode interface and bypass operation were developed in large part due to unhappiness with the never ending set of counter-mechanisms being employed to outsmart NFS. The extensible interface has been installed in the current Ficus implementation, and it has been used to add a bypass routine to the NFS layer so that the counter-mechanisms could be discarded.

2.3.4.2 Multi-layer NFS

The emergence of an ideal vnode transport layer suggests the possibility of reconstructing NFS as a pair of remote file service layers surrounding a vnode transport layer. (Note that the original Ficus design attempted just the reverse: to construct an ideal vnode transport layer around an NFS base.) Various beneficial data page and attribute caching features of NFS are retained, but the core protocol is different (see Figure 2.9).

In this design, the transport layer is common to both NFS and the Ficus logical and physical layers. It can also serve as the basis for any future boundary-crossing services, or be easily replaced in its entirety by a similar transparent service.

2.3.4.3 NFS compatibility

Altering the core NFS protocol renders the new NFS layer set incompatible with the standard NFS protocol suite, yet 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 2.10 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 permits a Ficus site to store replicas on a mainframe running MVS and NFS.[17]

2.4 Synchronization

The optimistic consistency perspective is not universally appropriate: some applications require stronger consistency guarantees such as serializability. This section considers the synchronization problem of coordinating access by multiple clients (possibly on distinct hosts) to a single file; multi-file synchronization is not directly addressed.

The goal here is to identify and address issues peculiar to the Ficus environment, especially the stackable layers architecture and the use of an optimistically replicated filing service as a base. In the course of the discussion, a series of related synchronization service designs which provide successively greater flexibility and robustness is presented. These designs implement standard approaches to synchronization in the layered, optimistically replicated filing context by placing a (multi-layer) synchronization service above the Ficus logical layer.

A complete design for a particular synchronization policy is beyond the scope of this work. These designs are not currently implemented, nor are a few necessary minor enhancements to the existing layering mechanism and logical and physical layers.

2.4.1 Issues

Recall that the Ficus replicated filing service (see Figure 2.6, page 40) is constructed from three layers (logical, physical, UFS) whose execution environment(s) are intended to be set apart from the remainder of the operating system. Recall also that a logical file is typically represented by multiple physical replicas. In this context, several questions immediately arise. First, for whom is synchronization to be provided? Second, what actions are to be synchronized? Third, what is the subject of synchronization? A synchronization service must have answers to each of these in order to be well defined.

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

vices is thus tightly confined to a single layer; the task of porting Ficus to other environments focuses primarily on this layer, and few others.[18]

Once the necessary client identification data has been obtained, the layer can determine whether it already possesses (in a volatile database) an appropriate capability to attach to the request, or whether it must first obtain a new capability from the synchronization service. The default policy used when a client has not provided a capability will vary according to environment. For example, in an academic environment a reasonable policy might be that all non-capability requests possessing the same credentials (UNIX user identifier, group identifier) be treated as emanating from a single "client" and so should be associated with a capability unique to that "client."

The identification layer supports two similar sets of file access operations: the standard set of operations (read, write, etc.) which do not include a capability parameter, and a parallel set that contains a capability. The standard (non-capability) operations consult the capability database so that a capability can be attached to the request before it is passed on to the next lower layer. The capability-included operations simply pass the request on immediately.

The important point here is that a synchronization service must assume that some appropriate notion of "client" exists, even when that concept is not well supported by either the host operating system or the layering methodology. A combination of capabilities and a layer instituting a client abstraction satisfy this requirement.

2.4.1.2 Client actions

The synchronization service acts as a funnel through which all file access requests pass. The primary actions of interest are `read` and `write`, although other requests (open and (`close`, for example) may also be of interest depending upon the synchronization policy to be enforced.

Suppose, for example, that the policy is to provide single-copy semantics to all callers, much as would be provided by a standalone UNIX host. The synchronization service must ensure that all `read` and `write` requests are (or appear to be) directed to a single file replica, and that any data caches maintained by lower layers are coherent with respect to synchronized access.

Two architectures to support single-copy semantics are shown in Figures 2.11 and 2.12. The first design forces all requests to pass through a single layer

[18]The transport layer is necessarily hardware architecture specific, and might be operating system specific as well.

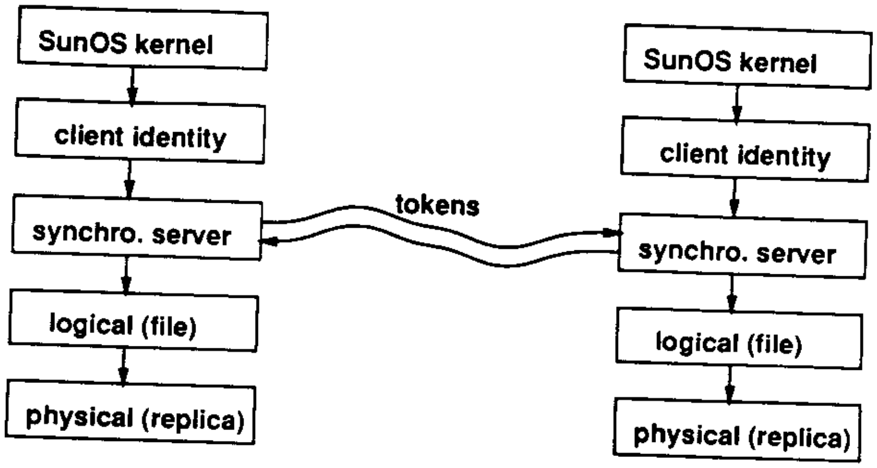

Figure 2.13: Quorum-based synchronization service.

instance, while the second uses a token mechanism to ensure that all `read` and `write` operations appear to happen to a single file, and are immediately visible to all clients.

The centralized design incorporates an additional layer that directs a request to a particular synchronization server. The "direction" information is extracted from the capability, where it was placed when generated by the server. This information consists of a host identifier and any other data required to locate the indicated server.

2.4.1.3 Objects

At its lower levels, the synchronization service must be cognizant of optimistic replication so that the appropriate quality of service may be provided. A particularly useful distinction which can be made is between synchronized access to a single file replica and synchronized access to a single logical file.

Replica-based service is adequately provided by the designs in Figures 2.11 and 2.12. Such service is constrained, however, by the robustness of the hosts maintaining the replica and executing the synchronization servers.

A synchronization service with increased robustness is readily constructed by exploiting replication to provide file-based synchronization. Figure 2.13 presents a design that uses quorum consensus replica management techniques for synchronization.

Each file replica contains a *read quorum* and *write quorum* attribute. The read or write). If the supplied version vector does not match the actual version of the replica, an error is returned to the caller. This change to the logical and physical layer services supplies the atomicity required by a synchronization service to ensure that data is read and written as expected, without undetected interference from unsynchronized activity.

2.4.2 Control flow examples

This section outlines the typical control flow for several common synchronization scenarios. The first two examples consider groups of processes that are unaware of the replication or synchronization services. The third example assumes that the clients are cognizant of the synchronization service, but oblivious to replication.

2.4.2.1 Process family

In this example, the collection of clients requiring synchronization is a process family formed by an off-the-shelf UNIX application; it has no knowledge of the synchronization service, replication, or any other non-traditional UNIX service. This example assumes that the default synchronization policy (e.g., serializability) provided by the synchronization server layer is appropriate for the application.

Suppose that the default client identification policy is that all processes issuing requests through the client identity layer but which do not include a capability are to be treated collectively as a single "client." The identity layer could then use a single, static capability (acquired during initialization at boot time) to attach to each request it passes on down the stack.

This scenario exemplifies a basic approach to providing backwards compatibility for UNIX applications on Ficus. Treating all clients as peers, though, may place undue strictures on availability for some clients. The following scenario assumes a simple reduction in synchronization granularity based on credentials.

2.4.2.2 Independent processes with identical user-ids

Here the processes of interest are initiated by standard applications, but the default identity granularity enforced by the client identification layer is a single process. In this case, synchronization can be effected by including a user-id based synchronization registration request as part of each login instance for the particular user-id.

ate.

2.5 Large scale replication

This section considers a number of issues with regards to very large numbers of replicas.

2.5.1 Version vectors

The theoretical ability of a version vector to scale up to any number of replicas does not obscure the practical reality: as the number of replicas increases, the processing, storage, and bandwidth overhead of a version vector eventually overshadows the costs of the replicated object itself. It is not readily apparent that the version vector mechanism is appropriate for large scale replication.

The basic mechanism can be adjusted in several ways to facilitate large scale replication. All of these alterations preserve the integrity of version vectors' support for "no lost updates" semantics. Some, however, may erroneously report an update/update conflict in certain circumstances; such anomalies are carefully noted in the sequel.

These adjustments all exploit various aspects of locality, with attendant implications for consistency mechanisms. Replica selection, update notification and reconciliation must each be re-examined in the context of modified version vector methods.

2.5.1.1 Compression

If a significant number of version vector components are zero-valued, conventional sparse matrix compression and manipulation techniques can be used to reduce overhead. But are version vectors sparse?

A non-zero version vector component indicates that the corresponding replica has been directly updated (as opposed to receiving propagated data) at some time in the past. Version vector sparseness, therefore, is a function of locality with respect to update access. Locality, in turn, is a function of the amount of update by distinct clients, and the concurrency displayed by shared update access.

File access locality studies have shown that shared update is rare in practice [Flo86b, Flo86a] in general purpose (university) settings. If, then, the (typi-

2.5.1.3 Upgradable read-only replicas

The most onerous aspect of read-only replicas manifests itself when the only available replica is read-only, and yet the ability to apply an update is highly desired. One solution to this problem is to allow a read-only replica to be upgraded to a writable replica.

This may be accomplished in a manner similar to adding an entirely new replica: simply add the replica's *repl-id* to the (local) replica list and permit the update to proceed. The appropriate version vector component will be incremented as usual. The newly-updatable replica then needs to persuade an established writable replica to propagate the replica list modification, either by issuing an update notification or by otherwise coaxing the other replica to initiate reconciliation with it so that it can learn of the list change.

The basic reconciliation algorithms (see Chapter 3) must be slightly adjusted to cope with additional replicas. In particular, they must assume that the replica list is a monotonic, length-increasing data structure. Aside from the difficulty that received replicas lists must be routinely sorted to account for divergence in the component ordering, the algorithms must tolerate a dynamic number of participants. Further details are found in Section 3.2.4.

Upgrading a read-only replica to writable status is irreversible in this model. If upgrading is a common occurrence for a single file (which would indicate that access locality is not a property of this file), the replica list may grow to an unmanageable size. Second-class replicas address this situation.

2.5.1.4 Second-class replicas

A second-class replica begins as a read-only replica. When an update is necessary, but only a read-only replica is accessible, the update is applied to that replica. Rather than append a new slot to the version vector, a flag is set which indicates that the replica is a logical descendant of the version indicated by its version vector.[20]

As with the upgradable strategy, a flagged replica must persuade a writable replica to reconcile against it. A qualifying writable replica has a version vector that is less than or equal to the flagged version vector. Any replica with a version

[20]The idea of supporting an "updatable" replica while avoiding the expense of allocating and maintaining a version vector component was first suggested for Coda in [SKK90]. Coda version vectors are quite different in detail from the original Parker version vectors [PPR83] that are used in Ficus.

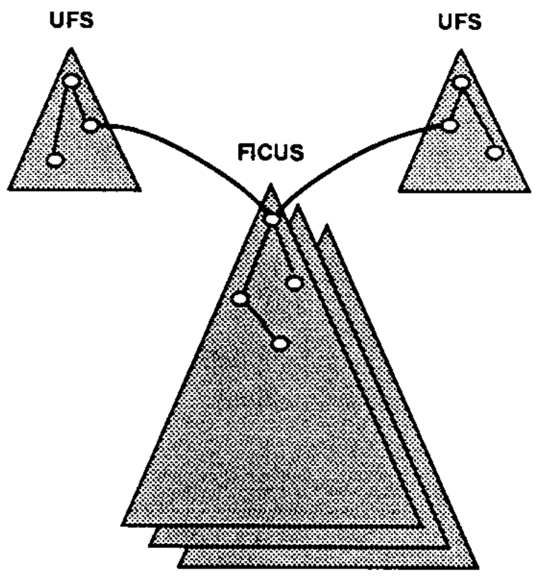

Figure 2.14: Current UFS and Ficus name space relationship.

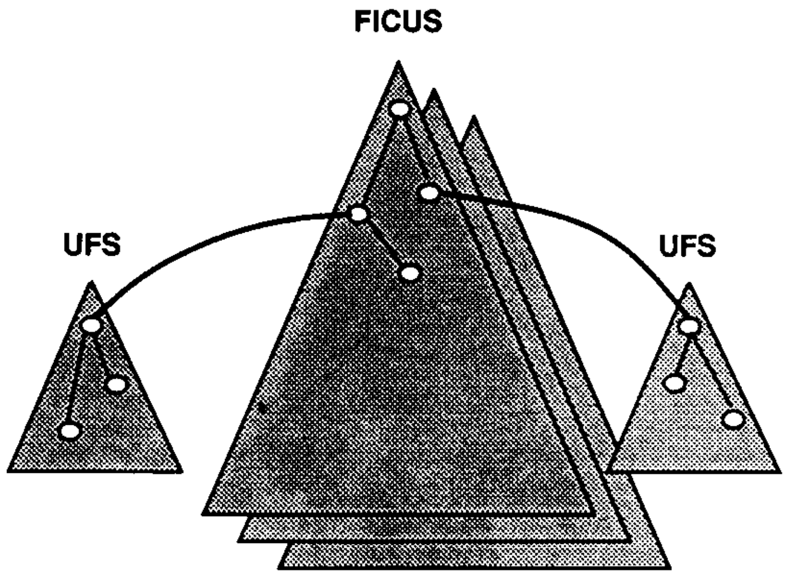

Figure 2.15: Ficus global name context.

be interpreted in a context associated with that file. A process should always be free to redefine its own context, and to ignore contexts provided by other objects or processes.

Closure support, as with some aspects of synchronization (see Section 2.4), straddles both file system and process management. Leveraging the existing SunOS process management services necessarily constrains the completeness of closure support in Ficus, to the extent that no process service enhancements are made. A further constraint is the pre-processing that SunOS file access system calls perform on file names: names are parsed by a system call routine, and partially or completely interpreted at the system call level without ever calling the file service proper. Nevertheless, a rudimentary closure service provided as a stackable layer is feasible.

The Ficus closure service design utilizes a new distinguished character ("@") as a prefix to signify a fully qualified file name; other names are interpreted in the traditional UNIX working directory and working root contexts. An additional context attribute is provided for each file. The closure layer is normally placed somewhere above the logical layer in a Ficus layer stack.

At system initialization time, the closure layer sets the default host working root to be a fully qualified name for a volume analogous to the traditional UFS "root filesystem". An internal pointer to this volume is inherited by all processes when they are created; this is the context in which the system call pre-processing routines will interpret "/"-prefixed names.

This approach offers immediate backward file naming compatibility, except when an existing file name (component) begins with "@".[23] It also offers a context-knowledgeable process the ability to establish contexts for files, and to inquire about file contexts. A process can simply prepend a context to a file name before presenting it to the system call service for translation.

This method does not automatically provide a custom context in which an executable image will execute, nor does it provide inherited contexts for process families. Ideally, a context would automatically be prepended to file names, without the need for a process to actually do so itself, and it would be inherited by child processes as are other process environment attributes. Support for these features is dependent upon modifications to the process creation and management services.

[23]This exception appears to be an unavoidable consequence of not modifying the existing process management and file access system call mechanisms. It is easy to avoid the exception by appropriate enhancements.

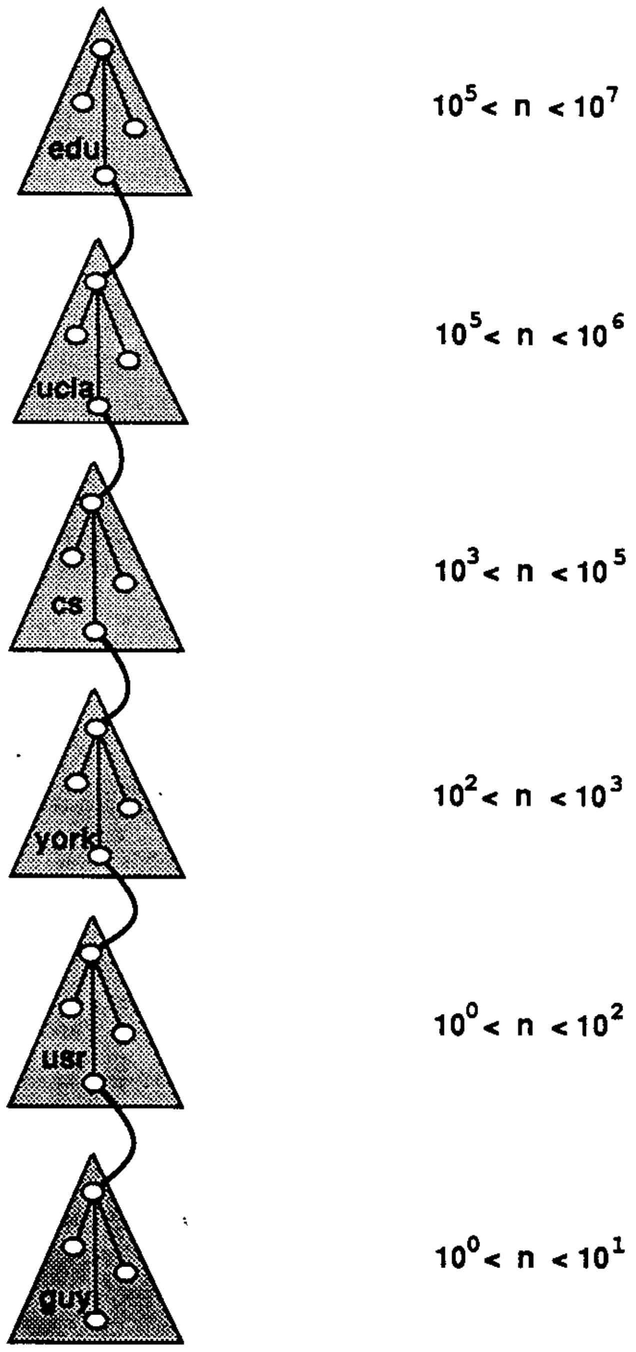

Figure 2.16: Volume support for global name space. (n = |replicas|)

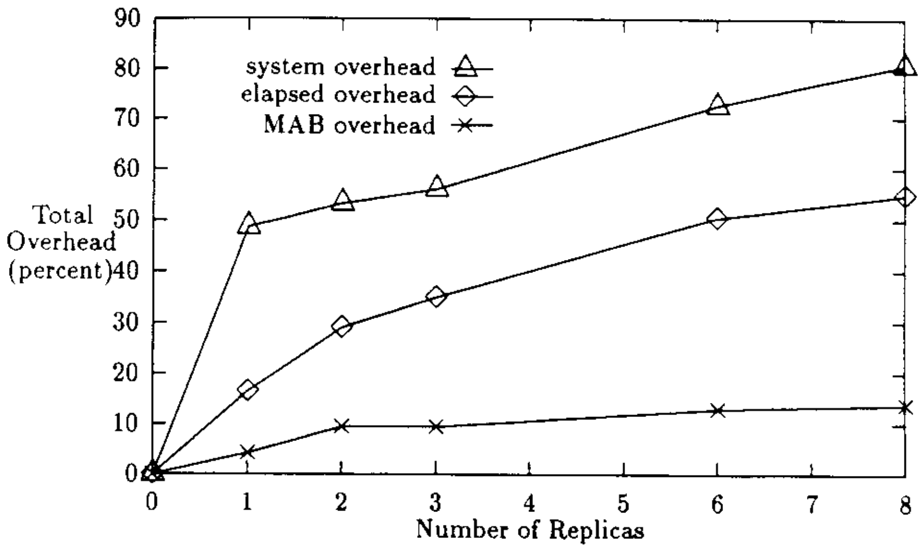

Figure 2.17: Percentage overhead versus number of replicas.

benchmark is not particularly illustrative of replicated file system performance (it is dominated by a largely cpu bound compilation phase), a second benchmark is used that is a much better worst case measure of Ficus. This second benchmark is a recursive copy ("cp -r /usr/include .") to a disk local to a Sun-3/60 from an NFS-mounted file system housed on a Sun-3/480 connected to the same ethernet segment. In the local environment, /usr/include is an unbalanced tree of depth four, with 47 directories and 1465 files totaling 4.7 Mbytes.[27] The costs studied were incurred by the site generating the activity. These measurements do not account for costs incurred by other processors.

Figure 2.17 shows the extent of Ficus costs (overhead) over normal SunOS performance. (A horizontal plot along the x-axis would indicate that Ficus was no more or less expensive than SunOS for a particular service.) Three sets of results are plotted, each measuring Ficus overhead as the target volume replication factor ranges from one to eight. One plot (*MAB overhead*) shows the overhead experienced by the Modified Andrew Benchmark (MAB) normalized against base MAB results obtained on standard SunOS. The other two plots show system time overhead (*system overhead*) and elapsed time overhead (*elapsed overhead*) displayed by the recursive copy benchmark, normalized against SunOS test results.

[27]Prior to execution of each benchmark, operating system caches were flushed to remove all references to the source and target volumes. The disk partition containing the target volume was reinitialized before each run of a benchmark. The measured node performed no other processing tasks during the benchmark.

2.6.3 Wide area operation

In the performance graphs shown, all replicas resided on machines on the same physical ethernet cable. Several of these measurements were repeated, this time locating replicas on sites connected by the Internet[28]. 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 essentially 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 can be improved substantially with careful analysis and optimization of the system's behavior (especially the effectiveness of its several caching mechanisms).

2.6.4 Implementation effort

Serious implementation work on Ficus has been underway for twenty four months. On the average, three experienced systems programmers have been engaged in full-time development. In addition, an average of two persons have worked on the design in parallel with the implementation. This author contributed approximately one man-year to the implementation effort, in addition to ongoing design research for more than three years.

The implementation has been done entirely in the C programming language. The logical and physical layers comprise about 9,500 and 12,000 lines of code, respectively.[29] The logical layer includes the update notification and propagation

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

[29]About 5,000 lines of physical layer code is devoted to Ficus directory manipulation and compatibility code to leverage UFS. Much of this code would be eliminated in a decomposed implementation, as outlined in Figure 2.8, page 50.

CHAPTER 3

Algorithms

Management of related, replicated objects is often fundamental to the design of reliable distributed systems. We are concerned both with the objects themselves: propagation of updates and reclamation of storage; as well as management of the possibly replicated directories used to keep track of and find the objects.

This chapter presents a family of algorithms for use in managing replicated objects and the accompanying graph structured directory systems. Members of this family are presented in order of increasing power and flexibility, followed by discussion of their correctness. The use of the algorithms in a replicated file system context is outlined throughout the presentation.

3.1 Introduction

Desires to improve availability and performance of information serves to motivate replicating information at locations "closer" to the data's intended use. A continuing difficulty in the operation of replicated information storage services, however, is unsatisfactory support for consistent update. Conventional methods achieve mutual consistency of data and the directories which refer to them by restricting availability for update. In the face of communications limitations, methods such as primary site, majority voting, quorum consensus, and the like reduce the performance and availability for update as the number of copies of an object or directory references is increased. This pattern is the reverse of what is desired.

There are numerous environments for which replicated storage is quite valuable. In some of these, rapid communication among sites is not suitable or even possible. Interesting examples include conventional high availability systems using redundant hardware, significant numbers of workstation users collectively engaged in a large software development project, an office workgroup composed of several widely geographically separated workgroups, large numbers of laptops operating while disconnected, and military systems subject to communications silence. These examples share several common characteristics:

3.1.1 File systems

The algorithms presented in this chapter are designed to provide for management and garbage collection of distributed, selectively replicated graph structures with associated resources. In practice, they have been extensively applied to the support of an optimistically replicated hierarchical filing system and accompanying name service for UNIX(see Chapter 2).

Consider the primary components of a typical UNIX file system. Files are hierarchically organized, with designated files (directories) containing the structural details (pathname components) which indicate a file's place in the hierarchy. The hierarchy is usually a restricted form of a directed acyclic graph.

Two types of "objects" are present, files and file names. For replication management purposes, each object can be treated independently, including independent consideration of a file and its names. In the algorithm model below, a UNIX file corresponds to a multiply-named logical object, while the file's names are considered to be singly-named objects in their own right.

Although we have applied these algorithms in the context of a standard UNIX file system, they can readily be used in other applications. For example, a distributed name service such as the DARPA Internet Domain Name System could directly use these algorithms to manage its databases.

3.1.2 Outline

The next section specifies the problem to be solved in more formal terms and presents a family of algorithms to address it. Section 3.3 presents correctness arguments which aid understanding of the algorithms. Algorithm applications are discussed in Section 3.4; an outline of related research in Section 3.5 concludes the chapter.

3.2 Algorithms

The task of a management algorithm is to support the propagation of changes to names and objects, and to identify and recover all resources supporting the existence of a logical object. This section presents a simple model of object and names, followed by several reclamation algorithms which address various combinations of object properties.

- static versus dynamic naming
- object mutability
- equivalence of name and object replication factors
- static versus dynamic replication factor

The algorithms presented in the next several subsections vary in their ability to handle these issues, ranging from the simplest combination (fixed name, immutable object with equivalent static replication factors for both name and object) to the most difficult (dynamically named mutable object with non-equivalent dynamic name and object replication factors).

To aid clarity of discussion, we assume that no more than one replica is stored by any given node. The algorithms generalize directly to multiple replicas per node.

We make minimum assumptions about the available communications environment to assure successful operation of the algorithms in practice. All we require is that information be able to flow from any node to any other in the network over time if relayed through intervening nodes. More formally:

> nodes N_1 and N_m are *time flow connected* if there is a finite sequence of nodes $N_1, N_2, \ldots, N_m$ such that for $1 \leq i < m$, a message can successfully be sent from N_i to N_{i+1} at time t_i, and $t_i < t_{i+1}$.

We require that every pair of nodes is time flow connected starting at any time.

This property does not require, for example, that any pair of nodes be operational simultaneously, but it does mean that no relevant node can be down indefinitely.

We also assume that nodes are truthful: Byzantine behavior does not occur. Finally, history only moves forward: a node must never "roll back" from a reported state, so stable storage of any reported state must precede that report.

3.2.2 Basic two-phase algorithm

The basic two-phase algorithm is appropriate for the simplest kind of replicated object: static single name, immutable object, and equivalent fixed name and object replication factors. The task at hand is simply to garbage collect. Subsequent more difficult types of management tasks adapt this algorithm to accomplish their goals.

```
/* variables and data structures:
   Let set R := replication factor,
        r,s   element drawn from R,
        self  is element of R,

        P1[]  binary vector of size |R|,
        P2[]  binary vector of size |R|,
        R[]   binary vector of size |R|.
*/
begin:  while (my name replica is not
                        marked deleted)
                { donothing; }
        mark my object replica deleted;
        P1[r] := 0, for all replicas r;
        P2[r] := 0, for all replicas r;
phase1: P1[self] := 1;
        while (P1[r] == 0 for any r) {
                R[r] := 0, for all r;
                choose some r to query;
                ask r for its P1 vector;
                if r responds {
                    R[] := r's response;
                    foreach (R[s] == 1)
                       { P1[s] := 1; }
                }
        }
phase2: P2[self] := 1;
        while (P2[r] == 0 for any r) {
                R[r] := 0, for all r;
                choose some r to query;
                ask r for its P2 vector;
                if r responds {
                    R[] := r's response;
                    foreach (R[s] == 1)
                       { P2[s] := 1; }
                }
        }
postlude:
        reclaim object replica resources;
        reclaim name replica resources;
```

Figure 3.1: Basic two-phase algorithm

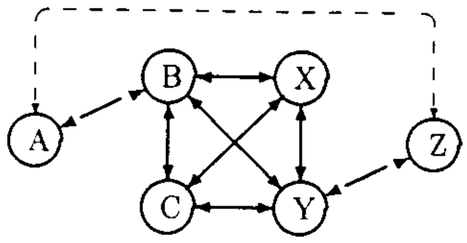

Figure 3.2: One phase network example

Node A notes that it is self-aware of reclamation, and begins the process of acquiring knowledge about other replicas' reclamation status. Suppose that the link between nodes A and B is the only remaining (albeit weak) link from Node A to the others.

Now consider Node B's possible perspectives upon receiving an inquiry from Node A (which contains the information that reclamation is in progress): Node B is either aware of the object already (because a replica exists at Node B), or it is not aware (no replica exists at Node B).

In the first case (Node B is already aware of the object), Node B notes that reclamation is in progress and Nodes A and B are cognizant of it. Node B, in turn, attempts to contact other nodes. Suppose the well connected group of nodes (B - Y) rapidly succeeds in learning that reclamation is in progress, and even manages to get a response back from Node Z acknowledging that reclamation is in progress. Further suppose that Node B is the first node to learn that every node is aware of the intent to reclaim. Node B therefore reclaims its resources and forgets entirely about the object (including the fact that its replica was reclaimed).

Continuing the scenario, when Node B receives the initial inquiry from Node A, it replies quickly. Unfortunately, congestion on the link causes the reply to be lost. Node A eventually sends another message to Node B inquiring about Node B's reclamation status, since it failed to get a response to the first message. By the time Node B receives the second message, its replica is already reclaimed. This scenario forces consideration of case two: Node B is unaware of the object.

Another way in which Node B might be unaware of the object is that Node B has never learned of the object before receiving the inquiry from Node A. (Perhaps Node A learned of the object directly from Node Z via the very weak link between them.) From Node B's perspective, these two situations are indistinguishable, yet its response must differ for the two scenarios: in one, Node B must establish a replica (whose body may be empty) to support indirect communication to other nodes about the reclamation initiated by Node A; in the other, all nodes are (or were) aware of reclamation, and do not need (or want) to re-establish replicas.

Failure to support indirect communication that may be critical to algorithm termination is unacceptable. It is also unacceptable to simply re-establish replicas just in case indirect communication support is needed: re-establishment in the above scenario is a side-effect of an event (successful transmission of a message) whose frequency is both unbounded (by the algorithm) and may not contribute to progress towards termination.

The two-phase nature of our algorithm provides an ignorant node with the ability always to distinguish "never knew" from "forgotten". An ignorant node which receives a phase one message correctly concludes "never knew", and establishes a replica to provide support for indirect communication. An ignorant node concludes "forgotten" upon receiving a phase two message, and does not establish a replica. (An ignorant node's reply to a phase two inquiry–"I know nothing"–implicitly means "I finished phase two, and so can you," which is all the inquirer needs to know to reclaim its replica's resources and terminate.)

3.2.3 Intermediate algorithm

The basic algorithm in the previous section applies to fixed name, immutable objects with identical static name and object replication factors. In this section we relax the first two constraints to allow dynamic naming and object updates, while continuing to require name and object replication factors to be both identical and static.

Dynamic naming and object mutability each complicate the reclamation problem, and the combination of the two is especially difficult. Dynamic naming introduces a global stable state detection problem, while object mutability requires special mechanisms to prevent inadvertent data loss.

3.2.3.1 Dynamic naming

A necessary, but not sufficient, condition for object reclamation is that the object have no names. In our model, 'no names' means that every name replica referring to an object replica has been marked deleted.

In the basic two-phase algorithm, object reclamation inevitably follows name removal; the two phases ensure that all replicas will be reclaimed, exactly once. In the dynamic naming case, it is much harder to determine whether or not reclamation is to occur: names may be added or removed at any node at any time, since optimism allows unsynchronized concurrent updates across non-communicating

The second phase proceeds similarly, with two parallel vectors of total name count values and report indicators. In this phase, the total name count values recorded reflect a replica's total name count value at some point after the queried replica has finished phase one.

As a node is collecting values in the second phase, it compares the newly reported values with those recorded in its phase one vector. If any discrepancy is discovered (i.e., the corresponding values are not identical), the algorithm aborts, initializes its vectors, and restarts phase one. This abort occurs when the transient behavior described above occurs.

A node finishes its second phase when all replicas have reported values to it, and the values are identical to those collected in the first phase. At this point, all object replicas are guaranteed to be permanently inaccessible.

3.2.3.3 Mutability

As presented, the intermediate algorithm will determine that an object is globally inaccessible. A further condition is necessary (and sufficient) to allow physical reclamation to proceed: data must not be lost inadvertently as an unavoidable consequence of optimism. We are not concerned here with the kind of 'inadvertent loss' that results when a client mistakenly removes a name, but with the consequences of concurrent update and name removal.

Consider a scenario with one object, two names, three replicas, and three clients. (Imagine a journal paper draft, with three collaborating colleagues.) Suppose that each of the nodes is isolated, but optimism allows each author to continue working. One author makes revisions to his object replica. Each of the other two authors decides (differently) that one of the two names is superfluous, and removes it. Each of the clients will be understandably disappointed if the object is reclaimed (since it eventually will be declared globally inaccessible), especially the one who updated it.

Our approach to the general problem of *remove/update conflicts* is to assume that name removal is undertaken in the context of an object replica. We invest the name removal operation with the additional semantics that a client wishes object reclamation (when no names exist) if no other object replicas are newer than (or in conflict with) the object replica which is initially affected by the name removal.

To accommodate the additional semantics, the reclamation algorithm must determine which of the object versions represented by the replicas is the latest, and which is the latest version to provide a context for name removal. (Opti-

Figures 3.3 and 3.4 show the intermediate algorithm.

3.2.4 Advanced two-phase algorithm

The previous algorithms each assume that object and name replication factors are fixed at creation time, and are identical. In practice, these constraints are not attractive. Changing circumstances of network behavior or object usage may necessitate adding, deleting, or moving replicas, which can not be usefully predicted when an object is created. It should also be possible to change an object's replication factor without directly affecting object names.

Note that an object (or name) replication factor is itself a replicated data structure which is used to manage other replicated data structures. The version vector technique used to manage updates to replicated data can not easily be applied to managing updates to version vectors themselves.

Our system supports an approximation to an ideal flexible replication factor mechanism: a replication factor can grow to be very large (2^{32} replicas), with masks used to 'shrink' a replication factor. One mask is used to indicate that particular replicas should be (irrevocably) ignored during algorithm execution. The second mask permits an object replica to avoid the expense of storing the object itself any further, but the 'skeletal' replica must continue to participate in algorithm execution. In short, a replication factor monotonically increases in physical size, with adjustments available to reduce the actual number of physical copies of a client's data which are maintained.

Increasing a replication factor is straightforward. Any replica's replication factor can be augmented simply by adding a (globally unique) replica identifier to its list of replicas. A replica can form a new replication factor while executing the one of the two-phase algorithms by taking the union of its replication factor and that reported by another replica.

A replication factor's 'ignore' mask provides a way for a replica to be forever ignored. This is especially useful when recovery of a destroyed replica is impossible or too expensive. Indicating that a replica is to be ignored is an irrevocable action. Like an increase in replication factor, a new ignore mask is formed by taking the union of the local mask and one reported by another replica.

The 'skeletal' mask indicates which object replicas don't actually store any client data. This mask is maintained in an optimistic fashion, but without conflict detection: mask updates cause a new timestamp to be generated for the mask; the mask with the latest timestamp is deemed to be correct.

3.3 Correctness discussion

The basic two-phase reclamation algorithm is *correct* if and only if these conditions are satisfied:

- object reclamation occurs if, and only if, the object is globally inaccessible
- for each replica of an inaccessible object, reclamation occurs exactly once, in finite time
- all algorithm executions terminate in finite time
- all algorithm executions are free from deadlock

We first show that reclamation occurs *if* an object is inaccessible, followed by the *only if* direction. We then show that reclamation occurs exactly once in finite time by proving that it occurs at least once, and at most once.

3.3.1 Reclamation *if* inaccessible

The "information flow" requirement governing network behavior ensures that it is possible for each node to learn of status changes at every other node. Since each node periodically uses the propagation protocol to incorporate other replicas' status changes into its own replica, and since all replicas are guaranteed to be available at the same time, each node will, in fact, learn in finite time of the status of every other replica. Therefore, every logical name deletion will eventually be reflected at every node, as each name replica will be indelibly marked deleted.

Following the deletion of every name for an object, in finite time all name replicas will be marked deleted. Each object replica will, in turn, have a zero-valued reference count, and be inaccessible.

The first phase of the algorithm simply collects the information that, when consulted, each replica was inaccessible. The second phase similarly collects information from each node. By the previous argument, each node is guaranteed to learn the desired information. At the conclusion of executing its second phase, a node reclaims its resources. Since each node is guaranteed to finish its phases if the object is inaccessible, each node will reclaim the resources consumed by the object.

```
  phase2: P2[self] := 1;
          while (P2[r] == 0 for any r) {
                NCR[r] := 0, for all r;
                choose some r to query;
  *             ask r for its C, NC2, P2;
  *                 VV, RC
  *             if r responds with C==0 {
  *                 NCR[] := r's NC2;
  *                 NV[] := r's P2;
  *                 foreach (NV[s] == 1) {
  *                   NC2[s] := NCR[s];
  *                   P2[s] := 1;
  *                   if (NC1[s] != NC2[s])
  *                       goto begin;
                    }
  *                 VVR := r's VV;
  *                 RCR := r's RC;
  *                 if (VVR conflicts SVV or
  *                     RCR conflicts RC)
  *                     { RU := 1}
  *                 if (VVR >= VV)
  *                     { SVV := VVR; }
  *                 if (RCR >= RC)
  *                     { RC := RCR; }
              } else if (C > 0)
                  { goto begin; }
          }
postlude:
        if (RU == 0) {
          reclaim object replica resources
          reclaim name replica resources
        } else {put object into orphanage}
```

Figure 3.4: Intermediate algorithm, phase two.

one A phase one message may or may not indicate that the replica has ever existed. If it does not indicate that the replica existed, a replica must be established. If it indicates that a replica once existed, an anomalous condition has been encountered. (See discussion below.)

two A prerequisite for entering phase two is that all replicas have been consulted, which implies that all replicas exist. Therefore, the replica has previously existed, been reclaimed, and must not be re-established.

A node which has already reclaimed its replica normally expects to receive only phase two messages, because a condition of phase two completion is to determine that all other nodes have finished phase one. Since phase two messages can not cause a replica to be re-established, only the receipt of phase zero or phase one messages after reclamation might cause a replica to be established again.

Phase zero or phase one messages received by a node which has completed phase two and reclaimed its replica could only have been sent *before* the sender began phase two. Such messages have been delayed in transit, long enough for the sender to finish phase one and the receiver to finish phase two.

The algorithm is resilient to delayed messages which are received within the next phase: phase one messages received by a node in the midst of phase two are quite normal, as are phase zero messages received during phase one. It is only when message delay exceeds one phase that replica re-establishment might occur.

We assume that message delays does not exceed the time required for one complete phase. If this bound is invalid, algorithm execution can be artificially slowed to increase the length of a phase until a valid bound is achieved. It is, therefore, feasible to prevent phase zero or phase one messages from arriving after reclamation occurs.

The hypothesized message received after a replica has been reclaimed must be from one of the three phases, but since delayed phase zero and phase one messages can be prevented and phase two messages do not cause replica establishment, no message which could cause replica establishment will be received. This contradicts the hypothesis that such a message might be received, and so replica re-establishment (and subsequent reclamation) after an initial reclamation is not possible.

3.3.4 Termination

We show that the algorithm terminates by defining a partial order on the possible states of a node during the algorithm's execution, and showing that all state tran-

3.4 Applications and observations

Which two-phase algorithms are appropriate for managing a UNIX file system? UNIX files are mutable, dynamically named objects, so at least the intermediate algorithm should be used for them. File names (directory entries) are simple objects which can be managed with the basic two-phase algorithm.

While the intermediate algorithm is a sufficient base upon which to construct a usable file system, the additional cost of implementing and using the advanced algorithm (with flexible replication factors) is negligible. Ficus incorporates the advanced algorithm to manage its files.

The advanced algorithm is also used to support the name service that connects subtrees together to form a large connected hierarchical filing environment. This name service plays a role similar to NIS (Yellow Pages) in NFS, or volume support in AFS. The implementations of these two applications (file hierarchy and volume hierarchy) are common, so multiple facilities were not required.

3.4.1 Directed acyclic graphs

The UNIX filing environment is a simple directed acyclic graph (dag) structure. These algorithms may be applied to an arbitrary graph structure as well, so long as there are no disconnected self-referential subgraphs. Additional mechanism is needed to handle that case.

In fact, modest mechanism beyond that discussed in this chapter is required even to handle dags. That is because the discussion was cast in terms of a single logical object. The additional facilities are simple, and discussed in [Guy87].

3.4.2 Performance

Performance of these algorithms is, of course, important. A suitable measure is the number of messages that must be exchanged in order to cause a set of n nodes with replicas to reach agreement. One would expect that the worst case could be expensive, since the underlying minimum communications assumptions do not allow a stylized pattern of interaction always to be employed. The worst case indeed requires $O(n^2)$ messages, as most nodes talk to most of the other nodes to complete each phase.[5]

[5]In each phase, in the worst case, a first node pulls from the $n-1$ other nodes to become knowledgeable. A second node then pulls from the remaining $n-2$ unknowledgeable nodes, and then the first, knowledgeable one. The third node pulls from the remaining $n-3$ unknowl-

Allchin [All83] and Wuu and Bernstein [WB84] expanded upon Fischer and Michael's approach to use two-dimensional timestamp matrices to reduce the number of messages exchanged, with small variations in semantics.

None of these works addressed the general problem of reclaiming resources of named replicated objects; they were concerned with "dictionary entries" as isolated entities.

Wiseman's survey [Wis88] of distributed garbage collection methods includes several techniques based on reference counting, but none are designed for use on replicated objects, and none are directly applicable to imperfectly connected networks.

by its successful use to manage Ficus' replicated volume location tables.

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 one 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 new layers to be developed and debugged in user space, and then moved into the kernel only after they were working. This substantially simplified the testing and debugging enterprise. Layers can indeed be transparently inserted between other layers, and even surround other layers. A replication service can be added to a layer stack without modifying existing layers, and yet perform well.

The availability of a general reconciliation service is also very useful. Usually, one must deal with the many boundary and error conditions that occur in a distributed program with a considerable variety of cleanup and management code throughout the system software. Instead, in Ficus failures may occur more freely without as much special handling to ensure the integrity and consistency of the data structure environment. The reconciliation service cleans up later. For example, volume grafting was made considerably easier by the (easy) transformation of its necessarily replicated data structures into Ficus directory entries. No special code was needed to maintain their consistency. There is thus reason to believe that services such as those provided by Ficus will be of substantial utility in general, and easy to include as a third-party contribution to a user's system.

The two-phase algorithm family addresses the heretofore unsolved garbage collection problem for replicated data structures. It may also be useful in similar contexts, such as cleaning up storage used in reliable broadcast protocols.

The final and perhaps most significant conclusion is that 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. The following section provides several suggestions for future work.

4.2 Future research directions

The most prominent area of immediate future research is to demonstrate that large scale is workable in practice. Further, a variety of additional layers are of interest. General service areas include performance tuning, security, and databases.

the file; the attribute would then later be used at file open time to construct a custom, “typed” file service for that one file.

[BGS86] Daniel Barbará, Hector Garcia-Molina, and Annemarie Spauster. "Protocols for Dynamic Vote Reassignment." In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pp. 195–205, August 1986.

[Bir85] Kenneth P. Birman. "Replication and Fault Tolerance in the Isis System." Technical Report TR 85-668, Cornell University, March 1985.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. "A Logic of Authentication." *ACM Transactions on Computer Systems*, **5**(1):47–76, February 1987.

[BJS86] Kenneth P. Birman, Thomas A. Joseph, Frank Schmuck, and Pat Stephenson. "Programming with Shared Bulletin Boards in Asynchronous Distributed Systems." Technical Report TR 86-772, Cornell University, August 1986.

[BK85] Barbara T. Blaustein and Charles W. Kaufman. "Updating Replicated Data during Communications Failures." In *Proceedings of the Eleventh International Conference on Very Large Data Bases*, pp. 49–58, August 1985.

[BMP87] Walter A. Burkhard, Bruce E. Martin, and Jehan-François Pâris. "The Gemini Replicated File System Test-bed." In *Proceedings of the Third International Conference on Data Engineering*, pp. 441–488. IEEE, February 1987.

[Bre86] O. P. Brereton. "Management of Replicated Files in a UNIX Environment." *Software–Pratice and Experience*, **16**(8):771–780, August 1986.

[BY87] F. B. Bastani and I-Ling Yen. "A Fault Tolerant Replicated Storage System." In *Proceedings of the Third International Conference on Data Engineering*, pp. 449–454. IEEE, February 1987.

[CL85] K. Mani Chandy and Leslie Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems." *ACM Transactions on Computer Systems*, **3**(1):63–75, February 1985.

[CM89] David R. Cheriton and Timothy P. Mann. "Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance." *ACM Transactions on Computer Systems*, **7**:147–183, May 1989.

[Flo86b] Rick Floyd. "Short-Term File Reference Patterns in a UNIX Environment." Technical Report TR-177, University of Rochester, March 1986.

[FM82] Michael J. Fischer and Alan Michael. "Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network." In *Proceedings of the ACM Symposium on Principles of Database Systems*, March 1982.

[GAB83] Hector Garcia-Molina, Tim Allen, Barbara Blaustein, R. Mark Chilenskas, and Daniel R. Ries. "Data-patch: Integrating Inconsistent Copies of a Database after a Partition." In *Proceedings of the Third IEEE Symposium on Reliability in Distributed Software and Database Systems*, pp. 38–44, October 1983.

[GGK87] Shai Gozani, Mary Gray, Srinivasan Keshav, Vijay Madisetti, Ethan Munson, Mendel Rosenblum, Steve Schoettler, Mark Sullivan, and Douglas Terry. "GAFFES: The design of a globally distributed file system." Technical Report UCB/CSD/87/361, Unviversity of California, Berkeley, June 1987.

[GHM90] 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." In *USENIX Conference Proceedings*, pp. 63–71. USENIX, June 1990.

[Gif79] D. K. Gifford. "Weighted Voting for Replicated Data." In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pp. 150–162. ACM, December 1979.

[Guy87] Richard G. Guy. *"A Replicated Filesystem Design for a Distributed UNIX System."*. Master's thesis, University of California, Los Angeles, 1987.

[Her86] Maurice Herlihy. "Dynamic Quorum Adjustment for Partitioned Data." Technical Report CMU-CS-86-147, Carnegie-Mellon University, September 1986.

[HHL88] Sandra M. Hedetniemi, Stephen T. Hedetniemi, and Arthur L. Liestman. "A Survey of Gossiping and Broadcasting in Communication Networks." *NETWORKS*, **18**:319–349, 1988.

System Architectural Overview." In *USENIX Conference Proceedings*, pp. 151–163. USENIX, June 1990.

[Kle86] S. R. Kleiman. "Vnodes: An Architecture for Multiple File System Types in Sun UNIX." In *USENIX Conference Proceedings*, pp. 238–247. USENIX, June 1986.

[KM86] Michael J. Karels and Marshall Kirk McKusick. "Toward a Compatible Filesystem Interface." In *Proceedings of the European Unix User's Group*, p. 15. EUUG, September 1986.

[Kur88] Øivind Kure. "Optimization of File Migration in Distributed Systems." Technical Report UCB/CSD 88/413, Unviversity of California, Berkeley, April 1988.

[MA69] Stuart E. Madnick and Joseph W. Alsop, II. "A modular approach to file system design." In *AFIPS Conference Proceedings Spring Joint Computer Conference*, pp. 1–13. AFIPS Press, May 1969.

[MB87] John H. Maloney and Andrew P. Black. "File Sessions: A Technique and its Application to the UNIX File System." In *Proceedings of the Third International Conference on Data Engineering*, pp. 54–61. IEEE, February 1987.

[MD74] Stuart E. Madnick and John J. Donovan. *Operating Systems.* McGraw-Hill Book Company, 1974.

[MJL84] Michael McKusick, William Joy, Samuel Leffler, and R. Fabry. "A Fast File System for UNIX." *ACM Transactions on Computer Systems*, **2**(3):181–197, August 1984.

[Neu87] Peter G. Neumann. "ARPANET Partial Outage Despite "Redundancy"." *ACM Software Engineering Notes*, **12**(1):17, January 1987.

[Neu89] B. Clifford Neuman. "The Need for Closure in Large Distributed Systems." *Operating System Review*, **23**(4):28–30, October 1989.

[NPP86] Jerre D. Noe, Andrew B. Proudfoot, and Calton Pu. "Replication in Distributed Systems: the Eden Experience." In *Proceedings of the Fall Joint Computer Conference*, pp. 1197–1209. IEEE, November 1986.

[NRC88] National Research Network Review Committee of the National Research Council. "Toward a national research network." National Academy Press, 1988.

[PW85] Gerald J. Popek and Bruce J. Walker. *The Locus Distributed System Architecture.* The MIT Press, 1985.

[RAA90] 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.

[Ran68] Brian Randell. "Towards a methodology of computer system design." Working paper for the NATO conference on computer software engineering at Garmisch, Germany, October 1968.

[RBF89] 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*, pp. 109–113. IEEE, September 1989.

[Rit84] Dennis M. Ritchie. "A Stream Input-Output System." *AT&T Bell Laboratories Technical Journal*, **63**(8):1897–1910, October 1984.

[RKH86] R. Rodriguez, M. Koehler, and R. Hyde. "The Generic File System." In *USENIX Conference Proceedings*, pp. 260–269. USENIX, June 1986.

[RO91] Mendel Rosenblum and John K. Ousterhout. "The Design and Implementation of a Log-Structured File System." Technical report, University of California, Berkeley, March 1991.

[Ros81] Eric Rosen. "Vulnerabilities of network control protocols." *ACM Software Engineering Notes*, **6**(1):6–8, January 1981.

[Ros90] David S. H. Rosenthal. "Evolving the Vnode Interface." In *USENIX Conference Proceedings*, pp. 107–118. USENIX, June 1990.

[RT88] Robbert van Renesse and Andrew S. Tanenbaum. "Voting with Ghosts." In *Proceedings of the Eigth International Conference on Distributed Computing Systems*, pp. 456–461. ACM, June 1988.

[Sal78] J. H. Saltzer. "Naming and Binding of Objects." In R. Bayer, editor, *Operating Systems*, volume 60 of *Lecture notes in Computer Science*, chapter 3, pp. 99–208. Springer Verlag, 1978.

[Tho78] R. H. Thomas. "A Solution to the Concurrency Control Problem for Multiple Copy Databases." In *Proceedings of the 16th IEEE Computer Society International Conference*. IEEE, Spring 1978.

[TKW85] G. M. Tomlinson, D. Keeffe, I. C. Wand, and A. J. Wellings. "The PULSE Distributed File System." *Software-Pratice and Experience*, **15**(11):1087–1101, November 1985.

[VM87] K. Vidyasankar and Toshimi Minoura. "An Optimistic Resiliency Control Scheme for Distributed Database Systems." *Lecture Notes in Computer Science*, **312**:297–309, July 1987.

[WB84] Gene T. J. Wuu and Arthur J. Bernstein. "Efficient Solutions to the Replicated Log and Dictionary Problems." In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, August 1984.

[Wis88] Simon R. Wiseman. *Garbage Collection in Distributed Systems*. Ph.D. dissertation, University of Newcastle Upon Tyne, November 1988.

[WPE83] 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*, pp. 49–70. ACM, October 1983.

[Wri83] David. D. Wright. "On Merging Partitioned Databases." In *Proceedings of the 1983 Annual Meeting of theACM Special Interest Group on Management of Data*, pp. 6–14, May 1983.

[ZE88] Edward R. Zayas and Craig F. Everhart. "Design and Specification of the Cellular Andrew Environment." Technical Report CMU-ITC-070, Carnegie-Mellon University, August 1988.