Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

THE FICUS DISTRIBUTED FILE SYSTEM: REPLICATION VIA STACKABLE LAYERS

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

April 1990
CSD-900009

The Ficus Distributed File System:
Replication via Stackable Layers*

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

Department of Computer Science
University of California Los Angeles

Abstract

Selective file replication is a key component of any highly reliable, large scale distributed file system. This paper describes the design and implementation of a file replication mechanism which is unique in at least two important ways. First, the service is provided via a *stackable layers* architecture. This structure permits replication to coexist with other extended filing environment features. Further, it implements the replication facility in a way that is independent of the underlying file system implementation, permitting a high degree of configuration flexibility and portability. Second, the system supports a very high degree of availability for write by allowing updates during network partition provided at least one replica is accessible. Conflicts are reliably detected and directory updates are automatically reconciled via algorithms described in [5]. A prototype of the Ficus replicated file system is operational at UCLA.

1 Introduction

This paper proposes a general *stackable* structuring approach to distributed file systems, that allows independent developers to produce value added modules for operating system filing environments, and to incorporate them without disturbing or recompiling other parts of the operating system. The approach is analogous to the streams service [11] used for structuring protocols in UNIX System V[1], and draws on object oriented approaches for its details. As a case study, we describe the design and implementation of an extensive selective file replication service in stackable form. The Ficus file system allows update whenever any copy of the needed data object is available, provides accurate detection of conflicting updates to files, and automatically reconciles independent directory updates where possible. This work has been done both in a general framework, and with compromises that allow the stackable architecture and the replication service to be embedded in any system which contains the Network File System (NFS[2]) [12] and its vnode [8] interfaces.

We argue below that such a modular architecture, as well as the flexible, decentralized replication approach, are valuable components of any very large scale distributed filing system, and also has substantial application in a smaller workgroup to provide increased availability and performance improvement.

1.1 The Extensible Stackable Architecture

It is widely recognized that there are great potential benefits to be had from structuring an operating system so that services can be easily added by independent third parties. The Mach approach [1] is a fine example of the application of that philosophy

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

[1]UNIX is a trademark of AT&T
[2]NFS is a trademark of Sun Microsystems, Inc.

to produce an interface at the virtual memory and process layer that permits multiple and independent operating system designs and implementations to be provided on top. The Unix System V streams design provides an environment and a set of interfaces by which network and device protocols may be added to an operating system at run time, stacking new layers on existing ones [11].

The filing service is a key component of most operating systems. The definition and support of well defined, internal interfaces in this area would allow a variety of services to be introduced to many systems without re-implementation of the rest of the file system. For example, one supplier might provide an extent based storage system; replicated storage might be available from another source; sophisticated multiuser synchronization, including mandatory controls could be built by a third party, and secure encryption with key management could be included for those sites that wished it. Clearly, the ability to snap such components together depends on an architecture which provides effective interfaces, both in terms of assuring that necessary characteristics are present, and doing so in a manner that allows superior quality implementations without forcing significant compromises.

In this paper we propose a *stackable layer* architecture and set of interfaces that we believe achieve these goals. We have evaluated the stackable layers approach by designing, building, and using several new layers. The work uses NFS as a point of departure. Below, we summarize the selective replication layer, both to illustrate the approach, and because we view that service, like stackable file system layers, as valuable components in general distributed file system design.

1.2 Approach to Replicated File Storage

We wish an approach to replication that is suitable both to small work groups, and at the same time can scale to very large environments. The user should retain considerable selective control over replication parameters (what, where, etc.) and availability of data should strictly increase with the number of copies. Costs should be low, in terms of introducing and administrating the service, explication of any new user interfaces, and actual execution overhead. Decentralized operation is essential in order to scale.

A critical aspect of most replication solutions is how they handle updates to replicated objects when copies are stored on sites which are unable to communicate. Most approaches to replication avoid this problem through a variety of means. The primary copy strategy [2] marks one copy as primary; all updates are sent to it, and other copies are updated as backups. Voting [14], weighted voting [4], and quorum consensus [6] methods require a subset of storage sites to be available, using methods to assure that only one subset of communicating sites will decide that update is permitted. Upon reconnection, the update is propagated to the other storage sites. All of these approaches increase read availability at the cost of *decreasing* availability of the replicated object for update. Imagine a replicated file on multiple workstations in a local Ethernet where a terminator is accidentally removed; network communication is widely disrupted. Under previous approaches, despite the presence of a local copy, the file is unavailable for update. In a wide area network, there are many sources of communications outages; see [5] for a more thorough discussion.

Note that all of these filing system methods use "no lost updates", a weaker definition of correctness than strict serializability, the common approach for databases. In the general filing environment, it is often recognized that the weaker semantic model is very attractive from a practical point of view, since it allows much more flexibility while still providing a satisfactory service. Conflicting updates are viewed as occurring reasonably rarely in practice, since there are frequently very few sources of updates to any particular data item. The replication service described here is oriented around this looser and more optimistic approach. Update is allowed whenever the necessary data is available, conflicting updates to files are accurately detected (using version vectors [10]), and directory updates are automatically integrated if possible. The actual reconciliation algorithms are discussed in a companion paper [5]; here we concentrate on the means by which such a replication service may be packaged as a stackable layer.

Several prior efforts with regard to replicated file management deserve mention. Locus [15] provided replication and permitted partitioned update. However, that work was done in the context of an integrated distributed operating system as opposed to a stackable file system. In addition, while they too detected conflicting updates, there was no mechanism to reconcile conflicting directory replicas. The Andrew project [7] takes a rather different view of replication, employing a single central copy of a file located on a server, with multiple copies cached on workstations with check-out/check-in semantics. The ISIS environment's Deceit file system [13], like ours, is built on top of NFS. It does not support further extensibility, and while it has a mode which permits partitioned update, it does not support automatic directory reconciliation.

1.3 Organization of the Paper

The next section is principally devoted to the general architectures that we espouse for file system design, together with their use in replicated storage services. Details of the implementation of the Ficus layers using UNIX and NFS follow in Section Three. We conclude with a summary of the design principles for a layered distributed file system implementation which we hope have been illustrated by the discussion to which we now turn.

2 Architecture of Ficus: Stackable Layers

The architecture of the Ficus Replicated Filesystem is based on the philosophy of *stackable layers*. Filesystems should be designed so that it is feasible to add new services by "slipping in" new transparent layers. With a symmetric interfaces for entering and leaving each layer, the presence of a new layer may be invisible to its neighbors.

Figure 1 shows the layers in a Ficus replicated file system architecture. In brief, the stack is rooted by a standard file system layer which provides the actual storage of individual copies of files on the media.

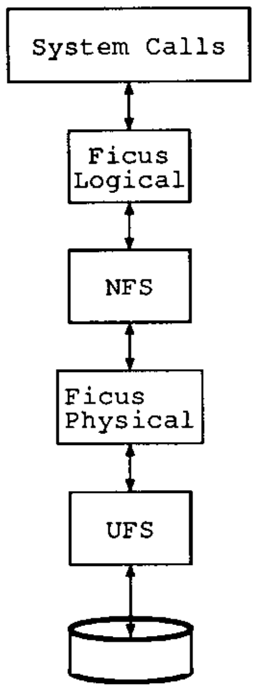

Figure 1: Ficus Stack of Layers

While we currently use the UNIX File System (UFS) exclusively, multiple implementations of the lowest layer can coexist. That is, file storage can be provided by multiple types of machines even running different operating systems or file managers.

The NFS layer facilitates access to remote filesystems[3] and is built on top of the various local file systems. It should be noted that in Sun OS the UFS interface is already identical to that of NFS. These first two layers predate this work and are unmodified.

Ficus provides replication in a pair of layers: the Ficus logical (FL) and the Ficus physical (FP) modules. For every logical file at the upper Ficus layer, there may be one or more files (replicas) at the Ficus physical layer. These physical files are actually stored by the UFS layer and made remotely accessi-

[3] We use the word "filesystem" in the UNIX sense: a subtree of files that can be mounted as a unit. We use the term "file system" (two words) to denote a file management system in general.

ble by NFS.

In the configuration displayed in Figure 1, the NFS layer is shown inserted between the logical and physical modules, permitting them to reside on different machines. Of course there are actually multiple instances of the lower levels of the stack connected to the FL layer, one for each replica. For any local file replica, the logical layer is connected directly to an instance of the physical layer without an intervening NFS module. An NFS layer could equally well be inserted above the logical layer (allowing access to a replicated filesystem from a site on which Ficus modules do not run), or below the physical layer (allowing sites on which Ficus does not run to act as replica storage sites).

2.1 Motivations for Stackable Layers

When we set out to implement our ideas about replicated file services, we desired to maximize the use of existing work. Considerable work has been done on fast local filesystems (see for example [9]) and we did not want to reimplement code to which we had little to add. Neither did we want to undertake a major redevelopment each time a new and improved filesystem is produced. Instead, it would be more attractive if our replication facility could easily be added to a wide set of other file system services. Similarly, it made sense to build on top of a standard facility for accessing remote files. Many sites already run NFS, and compatible implementations exist for many types of machines.[4] Together, these dictated an architecture for the replication service which is strictly separated from standard file system services, yet exploits the existing UFS and NFS modules. The concept of a stackable layered filesystem gives us a methodology for **extensibility** and **code reuse** which was previously lacking in operating system technology.

Transparency is also an important goal; in the same way that NFS makes the location of a file transparent in normal use, users and programs should not need to be aware of the existence of replication in the course of normal operations.[5] If transparency is maintained, the increased fault tolerance due to replication is achieved without the cost of increased complexity in the user model of the environment. In fact, if the potential for fault tolerance is to be realized, replication must be transparent so that other copies of a resource can be substituted for an inaccessible one. The replicated file service must accept the same interface as the existing filesystem if programs which access files are to run unmodified.

The need for a transparent interface leads directly to the concept of a value-added layer. We add the replication service in a transparent layer which "slips in" below the kernel's interface to files and above the underlying filesystems. The replication service gives the illusion of a single local copy of a file; all replica selection, remote communications, and update propagation are handled internally.

Because the interface both above and below the replication service is identical, it is much easier to generalize to multiple layers. Another layer with the same interface characteristics can potentially be inserted on either side of the replication service transparently to users, the replication layers, or the underlying physical file systems. For example, an encryption facility could be provided which would encrypt (or decrypt) the data on read (or write) before calling the read (or write) function in the next layer of the stack. All other operations would be passed on unmodified by the encryption layer.

This stackable structure has been especially useful in the integration of our replication service into a distributed environment. It is this aspect we examine next.

2.2 Layered Architecture in a Distributed System

The major function of the replication layer is to map between the single copy file model provided to the

[4] Other distributed file systems were considered (AT&T's RFS, AFS [7], Locus [15]). However, none besides NFS were widely used, had source code available to us, and ran on our Sun-3 family hardware.

[5] NFS does not provide identical semantics for remote files as for local files but the trade off of "80% of the transparency for 20% of the cost" has, judging from its widespread use, proven "good enough" for many cases in practice.

layer above and the multiple physical storage files provided by the layer below. However, the replication layer and the UFS layer will frequently be separated by a communications network, as each replica may well be stored on a different site. Further, for configuration flexibility, it should be possible for *any* pair of layers in the stack to reside on different physical sites. Thus, there must be a mechanism which maps calls from one layer, across the network, to the next layer down. This mapping must be done absolutely transparently as no layer should have to be concerned about where the layers above or below are located.

Following the stackable architecture philosophy, this transport mechanism which maps the interface across a network is itself a layer in the stack. The *transport layer* accepts the same interface both from above and below, mapping calls transparently across a communications channel to the next layer. A transport layer can then be inserted as desired between any pair of layers.

Permitting the insertion of a transport layer requires an interface definition that enables adjacent layers to execute in separate address spaces. However, in order to avoid associating layers of performance degradation with each layer of architecture, it is important that crossing layer boundaries be very inexpensive. Consequently, using a full remote procedure call mechanism to cross every layer boundary is unattractive; it should be possible to call functions from the layer below by address and to pass data structures via pointers when the layers are colocated. It is clear, however, that when a layer spans machine boundaries, as in the case of the transport layer, multiple address spaces are involved. The transport layer might not only span address spaces, but may also cross heterogeneous machine type boundaries, necessitating data conversion. We must ensure in any implementation that the need to package up data structures and function calls for transmission between address spaces or across machine boundaries does not dictate expensive data copying between purely local layers.

Sun's NFS provides an approximation of such a transport facility. It's existence is relatively transparent to the calling layer on the client side as it provides the same interface as local filesystems. The caller need not be aware that it is talking to NFS rather than a local user filesystem. On the server side, NFS communicates with the local filesystem via the same interface. The low level filesystem is unaware that the request is originating from across a network.

Unfortunately, NFS is not a transparent transport mechanism. It was not intended to be used in a stackable layers architecture. Rather, it was meant to implement access to remote filesystems and hence, its designers' opinions about the semantics of remote file access are built into NFS. Specifically, NFS was designed around the notion of stateless servers in order to simplify failure recovery. Certain interface operations which have no meaning in the context of the designer's view of remote file access (such as *open*, a stateful concept) are not transmitted across the network to the next layer in the stack. The operations are handled (or ignored) internally, and not passed through. In a stackable architecture, the transport mechanism should be truly transparent, with the desired behavior for remote access semantics implemented in a *pair* of layers on either side of the transport layer. What is needed amounts to an RPC mechanism with the same interface characteristics described for file system layers.

2.3 Pairs of Cooperating Layers

Sometimes a facility cannot be implemented totally within a single layer. It needs, instead, to be configured as a pair of cooperating layers with one or more layers providing other services in between. In Ficus, for example, one layer deals with a replicated file at a *logical* level, performing replica selection, reconciliation, etc. This layer is most conveniently associated with the client site. There is another per replica, or *physical* level, generally associated with the file storage site. These two layers which cooperate to provide replication service are sometimes separated by a transport layer. The intervening layer(s) must transmit uninterpreted information so that cooperating layers can communicate. This uninterpreted information may need to be packaged up for transmission across a network. Consequently, the format of this information must be self-describing.

2.4 Local State and Object-Oriented Programming

Any layer holding a pointer to a file object can perform any one of the set of generic operations defined on files. This set of operations makes up the common layer interface. However, the level performing the operation on an abstract file object does not necessarily know the identity (or type) of the next layer in the stack. Similar to operator overloading in programming languages, we can call an operation on a file object without knowing what code will actually run to perform that operation. This situation is quite analogous to object-oriented programming with inheritance.

Each layer in the stack may need to keep some information about the state of objects in its abstract filesystem. Further, the local state information needed by each layer may be different. The logical level, for example, may maintain a bit map of storage sites for the file, while at the UFS level, such a map would be meaningless as a file is purely a local object. Conversely, the set of data page pointers in a UFS-level file representation would be meaningless at the logical level. This situation is again quite reminiscent of object-oriented programming where the private state and public interface are encapsulated in an object.

Only the lowest level of the stack, the UFS in our case, actually stores file data pages and state information on disk. Higher layers are only abstractions. Thus, not only must uninterpreted data be transmitted between pairs of cooperating layers, it must also be passed all the way down the stack to be stably stored, and back up the stack when it is retrieved.

Consequently, each layer must support an "uninterpreted data" portion of the object representation which represents the local state information of the other layers which must be passed down for storage and up on retrieval. Each layer must also provide the full set of interface operations. It is often the case that a layer is not concerned with a particular operation; as with uninterpreted data, it must pass that operation through to the next lower layer unmodified. In this way both the state representation and the set of operations are extensible. New layers can be inserted and their new operations and state information remain transparent to existing layers.

2.5 The Vnode Interface

A key element in achieving a stackable architecture for a filesystem is the interface between layers. The virtual node or *vnode* interface within the Sun UNIX kernel provides the basis for our implementation and goes a fair way towards achieving an appropriate layer interface.

The motivation for creating the vnode interface was primarily to accommodate multiple filesystem implementations which could be "plugged into the kernel through a well defined interface." [8] It permits access to remote UNIX and even non-UNIX filesystems via the same interface as local access. It can, in fact, be viewed as a two level stack: the NFS layer is inserted transparently between the kernel's vnode interface on the using site and the vnode interface to a local filesystem on the storage site.

The vnode interface is implemented using an object-oriented programming style. A file is represented as a vnode which is an encapsulated object containing data and a pointer to a vector of operations on the object. The data part of the vnode has a set of generic file information (in a well-defined format) plus a pointer to an area of layer specific private data. For example, the private data in a UNIX filesystem layer is an inode. The vnode behaves like a typed object with overloaded operators; the vector of operations on the vnode contains pointers to functions which behave correctly for the vnode "type" and understand the format of the implementation specific data.

The vnode interface serves for us as a first approximation of a transparent interface between filesystem layers. Each layer in the stack is represented by a vnode type (and hence a virtual mounted filesystem). The function called by a vnode operation at any given layer can implement the desired functionality directly, pass it through to the next level, modify the arguments before passing it to the next layer, or implement it in terms of one or more other vnode operations at the next layer.

It will be seen that, as it was originally conceived to support only a two-level, non-extensible stack of layers, the vnode interface is not entirely suitable and should be extended. For example, the set of vnode operations are meant to be exhaustive; there is no provision for passing on to the next layer operations which have no definition locally. Consequently, if a feature is implemented via a pair of cooperating layers, any communications between these two layers must be implemented by overloading existing vnode operations rather than by adding additional ones. It is important to ensure that any intervening layers between the cooperating pair simply pass through the overloaded operation. Having to ensure this characteristic of intervening layers violates the abstraction and transparency that should exist between levels.

2.6 Configuring Ficus Layers

Figure 1 showed an example conceptual configuration for a stack of modules in a layered file system. However, Figure 2 more accurately shows the architecture, with each module at the same level, and a vnode interface both above and below. The vnode interface functions as a "switch" calling the appropriate function implementation depending on the type of the target vnode. This section examines how the mount protocol, whose original purpose was to "glue" filesystems into the naming hierarchy, is now overloaded to configure modules into a stack.

There are two seperable aspects to the UNIX file naming service. The first occurs within a filesystem where the name hierarchy is connected by entries in directories. Then, the sub-tree within an individual filesystem is linked into the global hierarchy by mounting its root directory on top of a leaf (known as a mount point) in another filesystem.

Just as there is an abstract object associated with each file (the vnode), there is an object, known as a vfs, for each filesystem. Like the vnode, the vfs contains implementation specific data and a vector of pointers to functions which perform operations on the filesystem. Recall that in UNIX when a filesystem is mounted, the mounted_here field in the vnode for the mount point is set to point to the vfs structure for the newly mounted filesystem. Operations on the mount point are then redirected to the vnode of the root of the mounted filesystem (pointed to by the vfs).

In addition to the filesystem being mounted and the point at which to mount it, the arguments to the mount system call include the type of the mounted filesystem. In Sun OS, the type can be UFS or NFS. We have added FL and FP filesystem types. The type specified in the mount dictates the type of vfs node created for the filesystem and hence the type of vnode created for each file which, in turn, controls the behavior of operations on the file. Let us now examine how the unordered layers shown in Figure 2 become stacked as in Figure 1.

A mount operation performs three actions. First, it creates an instance of a filesystem of the specified type. That is, it creates the correct vfs structure and links it in to the list of filesystems. Second, by making a leaf file in another mounted filesystem into a mount point, it creates the name by which the new filesystem will be addressed. Third, it links the new filesystem to an instance (or instances as in the case of the logical layer) of a level below.

The stack of layers is therefore created bottom up. First, (assuming a root directory is already mounted) a disk partition or device is UFS-mounted (mounted with type UFS) at a point in the root, creating an instance of a UFS, a name for it, and linking it to a lower layer (the raw disk). Then that UFS is FP-mounted creating a new instance of an FP filesystem, a new name for that filesystem, and linking the new FP filesystem to the mounted UFS. Any operations to files named via the new name are applied to FP-type vnodes and inherit their behavior from the FP layer.

This FP filesystem can then be FL-mounted on the same site. A remote FP filesystem can be NFS-mounted and that NFS filesystem subsequently FL-mounted to provide access to a remote replica.

When an operation, say a read, is performed on the replicated file, the corresponding operation is performed on the FL vnode. The FL vnode has pointers in its private data to the vnodes for each of the replicas. Once a particular replica is selected, a corresponding vnode operation is performed on the

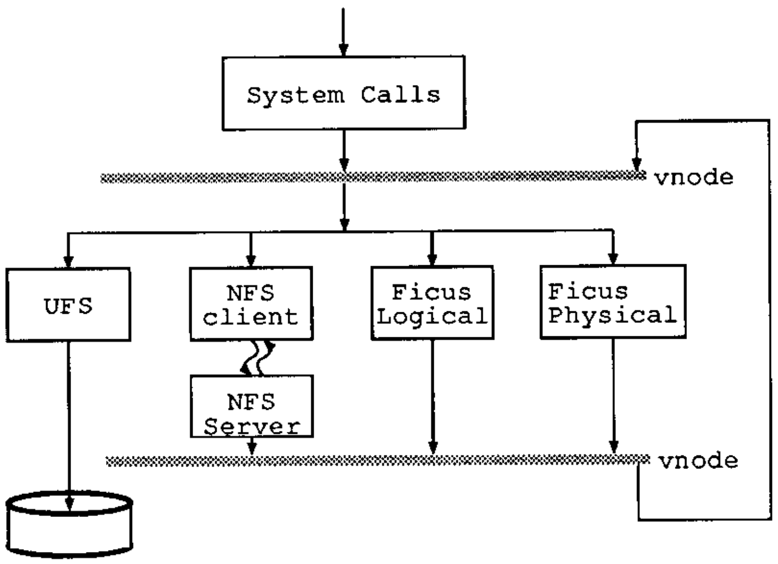

Figure 2: Layered Architecture Using Vnodes

lower level (NFS or FP) vnode. If the storage site is remote, the lower level vnode is of type NFS and it, in turn, transmits the operation via RPC to the next level vnode at the storage site. If it is local, the next vnode is in the FP layer. The physical layer has a pointer to the corresponding UFS vnode which in turn has access to the actual data.

2.6.1 Configuration Flexibility

The NFS layer may, in addition, be inserted transparently either above the logical layer, or between the physical and UFS layers. The former gives us the flexibility to access a replicated filesystem on a site which does not run the Ficus code, perhaps because it is of an unsupported cpu type. The latter lends the capability of using such an unsupported machine type as a storage site for a replica of a Ficus file. For example, an NFS client implementation exists for DOS on PCs allowing them to access remote Ficus filesystems. An NFS server implementation for MVS is available, immediately permitting use of large IBM "disk farms" to support replicated data storage. This configuration flexibility is an immediate payoff of the stackable architecture.

2.6.2 Name Hiding

While a mount of any type creates a name in the name hierarchy for the instance of the filesystem being mounted, users ordinarily interact with only the highest layer in the stack; the names of the intermediate virtual filesystems should be invisible in the course of normal operations. However, special programs (eg. reconciliation) or users occasionally need to bypass higher layers and access to the lower level interfaces directly. This capability is currently provided by mounting all but the top level on mount points whose name begins with "." (such names are normally hidden by UNIX but access to them is permitted). A more sophisticated solution is no doubt called for and will be pursued.

2.7 Scaling Considerations

Our goal is to achieve a very large scale, network transparent, replicated file system. One aspect of transparency is that each site should see the same global name hierarchy. Thus logical file systems should be mounted at the same point in the hierarchy on every site.[6] However, this amounts to a globally consistent replicated mount table with an entry for every filesystem in the network. This is clearly not viable on a large scale; a workstation being switched on in Washington should not cause a mount to be executed on every other computer in America.

We instead employ an automout protocol which dynamically mounts a filesystem at the correct point in the name hierarchy when its mount point is accessed. This mechanism effectively distributes the mount table information at precisely those points where it is used. Further, the very large scale transparent name space is maintained without devoting resourses to portions of the hierarchy which are not in use.

The vnode for a mount point contains a pointer to the vnode of the root of the filesystem to be mounted at that point. Operations on the vnode of the mount point are passed on to the root vnode of the filesystem. If the referenced filesystem has not yet been mounted, the vnode pointer is NULL. In that case, the file corresponding to the mount point contains the location and identity of all replicas for the filesystem. Each replica may then be NFS-mounted and the new filesystem subsequently FL-mounted. Thus filesystems are only mounted when needed, but they are always mounted at the same point in the hierarchy.

3 Implementation of the Ficus File System

This section describes an implementation of the Ficus replicate file system layers. The particular design was developed to operate within existing NFS and UNIX environments without change to them. As we shall see, these constraints required choices that can be more effectively made when there are fewer constraints.

The design of Ficus identified several new types of information that must exist in a replicated file system. An important issue in the implementation of the Ficus replicated file system is how this additional information can be organized and stored on top of existing file systems. New information is kept for each logical file, existing file replica, file name, and for the file system as a whole. Let us examine how each layer in the replicated file system manages its data.

3.1 Logical and Physical File Level Information

In UNIX, each file is represented by a data structure called an *inode*. There is exactly one inode per file. It contains page pointers to the file data, ownership and protection information, link count, and other file-specific data. In a replicated file system, the notion of what constitutes a file is more complicated, since there are now both the logical and physical file levels. For each replica there is a UFS file[7] with an inode containing the traditional inode information, and thus several inodes per logical file.

There are two significant implications for our implementation. First, since there are multiple UFS files for each Ficus logical file, each with a separate inode, the inode number is no longer a unique identifier for a file. A new, logical level file identifier is needed as an unique internal file name. Second, Ficus must store additional information at both the per replica and at the per logical file levels. Following the principles of a layered architecture, we must store this added information without modifying the underlying inode or file storage structure.

The first problem is addressed by the invention of a new identifier called a *fileid*. Fileids provide a system-wide level of naming that is one-to-one with logical files. However, they create a problem with directory entries. UFS level directories map from a path name component to the inode number of the directory or file associated with that name. However, in a replicated file system, directory entries must map name components to logical level fileids, not UFS level inodes. Then a particular replica stor-

[6]Note that NFS permits filesystems to be mounted arbitrarily on a site by site basis.

[7]Of course, given the independence provided by the stackable architecture, the lower layer could be provided by rather different means; even through a non-UNIX file service.

age site must be selected. Finally, the Ficus physical level associated with the selected replica maps the fileid to the corresponding UFS file and inode.

Each Ficus physical level implements this mapping using the UFS level directory mechanism. UFS directories with entries mapping each fileid to an inode number are stored at every replica site.[8] To take advantage of the locality of reference typically exhibited in UNIX among files in the same directory [3], our implementation clusters mapping information based on the directory in which the Ficus logical file resides. While two additional disk page reads may be required to obtain the information to perform the mapping for the first file opened in a directory, subsequent opens in that directory will incur no additional page reads over standard UNIX.[9]

The second problem of additional storage exists because there is extra information that must be kept with each replica beyond what the underlying UFS implementations provide. The Ficus physical level requires information such as the version vector [10] in order to perform conflicting update detection and reconciliation. The Ficus logical layer must also store a list describing where each physical copy of the file is located. In order to avoid a single site failure prohibiting access to the logical level information, our implementation replicates this information at each site which stores the physical level information.

Conceptually, this logical and physical level information is part of the file-specific data traditionally stored in the UNIX inode. However, existing UFS implementations make no provisions for storing uninterpreted data needed by higher layers. Instead, we place this information in an auxiliary file, in effect creating a parallel set of inodes. Like the fileid to UFS file mapping, the additional inode information is clustered by directory so no additional I/O cost is typically incurred beyond the first file open in a directory.

[8]This design is forced on us by the lack of an inode interface to files in current implementations of UNIX filesystems. If such an interface existed, a more efficient in memory data structure could be employed. This same problem prompted the addition of an inode interface for UNIX files by the Andrew Filesystem project [7].

[9]Note that this assumes that additional space is devoted to directory entry and buffer caching. Given current trends in memory prices, this should pose no problem.

3.2 Filename Information

In UNIX, a file at the user level can have more than one name, corresponding to multiple hard links to a single file. Each UNIX directory entry maps a user-level name to a system-level inode number. When two directory entries map to the same inode, the file has multiple "names", all equally valid. Ficus' automatic reconciliation of directory updates made during partitioned operation requires changes to the traditional UNIX directory structure.

Ficus, unlike UNIX, permits a general DAG structure in the directory hierarchy. While UNIX allows files to have multiple names, it permits only a single name for each directory. Our optimistic concurrency control strategy allows updates to directories during a network partition. In particular, a directory might be renamed in two non-communicating partitions. When the two partitions come back together, two names exist for the same directory.

Automatically reconciling conflicting versions of a replicated directory requires that additional information be kept with each directory entry. For example, consider trying to reconcile two copies of a directory, one in which a filename "foo" exists and the other in which it does not. Without the additional information there is no way to decide if the file was newly created on the first site and so should be propagated to the second, or if the file which previously existed on both sites was removed from the second, and so should now be deleted on the first. Fortunately, additional information can resolve these problems and permit almost completely automatic resolution of independent directory updates. A complete description of the algorithms this requires can be found in [5].

Our need to store additional information in each directory entry and the requirement to manage a general DAG structure means that Ficus cannot simply make use of the existing UFS directory service. Because of this, we re-implement directories at the Ficus physical layer, storing our extended directory information in standard data files on the underlying UFS.

We recommend that future implementations of the

UFS layer will provide uninterpreted data storage facilities in both inodes and directory entries, obsoleting the need for our parallel inode tables and directory facility.

3.3 File System Granularity

Finally, there is a small amount of information that must be stored at the Ficus filesystem level. Replicas for a file system may be stored at a set of sites. Placing this information at the file system level allows a simple bitmap to identify replica locations for each file, facilitating partial replication.[10] Some other bookeeping information must be kept at the file system level, such as a record of the last-used fileid.

Traditional UNIX stores filesystem information in the superblock. In keeping with our desire not to change the UFS, we keep a small auxiliary file in the root of the UFS. This information is read and cached when the Ficus file system is mounted.

4 Conclusions and Experience

This paper proposes stackable layers as an architectural paradigm for future distributed file systems. Today's file management systems are exceedingly complex pieces of software and the challenges presented by the globalization of computer networks will make them all the more so. Many features will have to be added: selective replication, data security, user authentication, and type conversion among heterogeneous storage conventions are but a few examples. The stackable layers architecture provides a methodology for extensibility which is crucial for the advancement of distributed file system technology.

The Ficus replicated file system has been implemented and is in use at UCLA. As a case study, Ficus demonstrates that the stackable architecture is logically feasible and can, with care, be made to perform satisfactorily. Both to increase portability and to ease our initial implementation burden, we elected not to modify the underlying UNIX file system or NFS. We devised short term solutions to map the requirements of a stackable architecture into the functionality provided by UNIX and NFS. This effort led us to derive a number of design principles for a stackable architecture to guide future implementations of distributed file management mechanisms.

[10]Partial replication is supported; not every file is replicated at every site which stores the file system.

4.1 Interface Design

The primary problem we encountered in using Sun's vnode interface as the basis for our layer interface implementation is the lack of provision for extensibility. While the set of vnode operations was originally intended to be exhaustive, new layers which add features to the file system will typically support a superset of the original operations. The interface needs to be extensible in several dimensions. It must allow for the addition of new operations on files. As some features need to be implemented in pairs of cooperating layers which may be separated by other modules (and run on different sites), there must be a way for one layer to call functions in its partner, even though the intervening layers do not implement the corresponding function. This implies that layers must pass on, transparently, any operations which they do not explicitly intercept. Finally, the need in intermediate layers for stable storage of their private information necessitates a facility to pass uninterpreted data between layers. We are continuing work on an extensible file system interface with these characteristics.

4.2 The Transparent Network Layer

The current version of Ficus makes use of an unmodified Sun NFS as the network transport protocol. NFS was chosen both to ease our initial implementation burden and to increase the portability of Ficus. In building replication on top of NFS, we came to a number of conclusions regarding the use of a layered architecture in a distributed system.

Any pair of layers should, in principle, be able to execute on different physical sites, transparently to those layers. This necessitates an RPC mechanism which adheres to the same interface as the other modules in the stack. NFS is ideal in this respect as it uses the same vnode interface as the UFS. Our problem is that NFS is not semantically transparent; the stateless server semantics are built into NFS. In particular, NFS does not transmit open and close operations as they are meaningless to a stateless server. This makes it difficult to alter the semantics, say to implement a commit on close. In our initial implementation, we have had to overload certain operations that are transmitted transparently in order to avoid modifying NFS. We propose that the semantics of remote file access be implemented in a replaceable pair of layers on either side of a transparent transport layer. Our ongoing work will investigate such a facility.

4.3 The Base File System

As with the layer interface and network transport module, we have chosen to use unmodified existing code (the UNIX file system) as the base of the stack of layers. As a result, UNIX systems can make use of the Ficus replication facility without modifying their filesystems. However, several modifications to the UFS design would make it a more suitable base for an extensible layered file system architecture.

First and foremost, the major data structures should be extensible. Each layer in a file system stack may need to store some information with the file, and the only way for higher layers to store data on a disk is through the UFS. Without the luxury of an extensible inode, we have had to store the extra data needed for replication in auxiliary files, necessitating additional I/O to open a file, in some cases. Some layers also require additional information in directory entries. The lack of an extensible directory entry prompted us to build our own directory mechanism. However, UNIX's lack of an inode interface to files, means that we have to employ a UNIX directory mechanism underneath our own, again at the expense of occasional added I/O cost. Future versions of the base file system should provide other layers with extensible basic storage structures and an inode-level interface.

4.4 Development Strategy

Developing code to run inside an operating system kernel is many times more difficult than developing application programs. A bug in a kernel program produces a crash rather than a core dump. Kernel debugging tools, when available at all, are very much less convenient to use than application level debuggers. Recompiling and retesting a kernel is very time consuming and involves rebooting a machine.

The layered architecture permitted a development strategy which significantly reduced the time required to implement Ficus. As the first step, we built scaffolding which permitted any layer in the stack to be pulled outside the kernel. This was accomplished using a client and server protocol, much like NFS, to map the vnode interface across a socket to a program outside kernel address space.

New layers can be debugged as an application program using available debuggers. Calls from the external layer to the next layer below (running within the kernel) are achieved by adding a system call interface to vnode operations. Once a layer is debugged in user mode, it is trivial to pull it back inside the kernel. This methodology significantly reduces the cost of debugging and testing new layers.

While experience using the Ficus replication facility is limited, it is also quite positive. Selective replication is an essential capability if high availability is to be achieved in large scale distributed file environments. We conclude that selective replication can be effectively provided in this manner. Moreover, the methodology for extensible file system services using stackable layers is extremely attractive.

Acknowledgements

The authors wish to acknowledge the contributions of Dieter Rothmeier and Wai Mak to the implementation of the Ficus replicated file system.

References

[1] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *USENIX Conference Proceedings*, pages 93–113, June 1986.

[2] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the Second International Conference on Software Engineering*, pages 562–570, October 1976.

[3] Rick Floyd. Directory reference patterns in a UNIX environment. Technical Report TR 179, University of Rochester, August 1986.

[4] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, December 1979.

[5] Richard Guy and Gerald Popek. Reconciling partially replicated name spaces. Submitted concurrently to Ninth Symposium on Reliable Distributed Systems, 1990.

[6] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

[7] John Howard, Michael Kazar, Sherri Menees, David Nichols, M. Satyanarayanan, Robert Sidebotham, and Michael West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

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

[9] 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.

[10] D. Stott Parker, Jr., Gerald Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, May 1983.

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

[12] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Wa lsh, and Bob Lyon. Design and implementation of the sun network filesystem. In *USENIX Conference Proceedings*, pages 119–130, Portland, OR, Summer 1985.

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

[14] 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.

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