Appeared in the Proceedings of the
Summer USENIX Conference,
June 1991, pages 17-29

Management of Replicated Volume Location Data in the Ficus Replicated File System*

Thomas W. Page Jr., Richard G. Guy, John S. Heidemann, Gerald J. Popek†, Wai Mak, and Dieter Rothmeier

Department of Computer Science
University of California Los Angeles
{*page,guy,popek,johnh,waimak,dieter*}*@cs.ucla.edu*

Abstract

Existing techniques to provide name transparency in distributed file systems have been designed for modest scale systems, and do not readily extend to very large configurations. This paper details the approach which is now operational in the Ficus replicated UNIX filing environment, and shows how it differs from other methods currently in use. The Ficus mechanism permits optimistic management of the volume location data by exploiting the existing directory reconciliation algorithms which merge directory updates made during network partition.

1 Introduction

Most UNIX file system implementations separate the maintenance of name binding data into two levels: within a volume (or filesystem), directory entries bind names to low-level identifiers (such as inodes); a second mechanism is used to form a super-tree by "gluing" the set of volumes to each other. The traditional UNIX volume super-tree connection 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) [13] and IBM's TCF [12]; larger scale file systems are exemplified by AFS [7], Decorum [6], Coda [14], Sprite [9], and Ficus [2, 10]).

The problem addressed by this paper is simply stated as follows: in the course of expanding a path name in a distributed file system, the system encounters a *graft point*. That is, it reaches a leaf-node in the current volume which indicates that path name expansion should continue in the root directory of another volume which is (to be) grafted on at that point. How does the system identify and locate (a replica of) that next volume? Solutions to this problem are very much

*This work was sponsored by DARPA contract number F29601-87-C-0072.
†This author is also associated with Locus Computing Corporation.

Appeared in the Proceedings of the
Summer USENIX Conference,
June 1991, pages 17-29

constrained by the number of volumes in the name hierarchy, the number of replicas of volumes, the topology and failure characteristics of the communications network, the frequency or ease with which replica storage sites or graft points change, and the degree to which the hierarchy of volumes spans multiple administrative domains.

1.1 Related Solutions

The act of gluing sub-hierarchies of the name space together is commonly known as *mounting.*[1] In a conventional single-host UNIX system, a single mount table exists which contains the mappings between the mounted-on leaf nodes 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 exceedingly difficult.

IBM's TCF, like its predecessor Locus [12], achieves transparency by renegotiating a common view of the mount table among all sites in a partition every time the communications or 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 change may be considered. Clearly neither of the above approaches scales beyond a few tens of sites.

Cellular AFS [15] (like Ficus) is designed for larger scale application. AFS employs a *Volume Location Data Base* (VLDB) for each cell (local cluster) which is replicated on the cell's backbone servers. The mount point itself contains the cell and volume identifiers. The volume identifier is used as a key to locate the volume in a copy of the VLDB within the indicated cell. Volume location information, once obtained, is cached by each site. The VLDB is managed separately

[1] We will henceforth use the term "graft" and "graft point" for the Ficus notion of grafting volumes while retaining the mount terminology for the UNIX notion of mounting filesystems.

Appeared in the Proceedings of the
Summer USENIX Conference,
June 1991, pages 17-29

from the file system using its own replication and consistency mechanism. A primary copy of the VLDB on the *system control machine* periodically polls the other replicas to pull over any updates, compute a new VLDB for the cell, and redistribute it to the replicas. The design does not permit volumes to move across cell boundaries, and does not provide location transparency across cells, as each cell's management may mount remote cell volumes anywhere in the namespace. Again, this may be billed as a feature or a limitation depending on where one stands on the tradeoff between cell autonomy and global transparency.

1.2 The Ficus Solution

Ficus uses AFS-style *on disk mounts*, and (unlike NFS) readily traverses remote mount points. The difference between the Ficus and AFS methods lies in the nature of Ficus volumes (which are replicated) and the relationship of graft points and volume location databases.

In Ficus, like AFS [5], a volume is a collection of files which are managed together and which form a subtree of the name space[2]. Each logical volume 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. 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 a volume and places the data in the mounted-on leaf (a graft point). 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 may be replicated and manipulated just like any other object (file or directory) in a volume. In Ficus 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 extreme similarity between graft points and normal file directories allows the use of the same optimistic replication and reconciliation mechanism that manages directory updates. Without building any additional mechanism, graft point updates are propagated to accessible replicas, uncoordinated 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,

[2] Whereas a filesystem in UNIX is traditionally one-to-one with a disk partition, a volume is a logical grouping of files which says nothing about how they are mapped to disk partitions. Volumes are generally finer granularity than filesystems; it may be convenient to think of several volumes within one filesystem (say one volume for each user's home directory and sub-tree) though the actual mapping of volumes to disk partitions is a lower level issue.

Appeared in the Proceedings of the
Summer USENIX Conference,
June 1991, pages 17-29

with no constraints on the number of volumes, volume replicas, changes in volume replication factors, or network topology and connectivity considerations.

1.3 Organization of the Paper

This introduction has presented the problem of locating file replicas in a very large scale, geographically distributed replicated file system, and briefly stated our solution. Section 2 gives an overview of Ficus to put our solutions in context. Section 3 details our volume location and autografting strategy. Then, Section 4 examines the consequences of updating volume location information using the optimistic concurrency and lazy propagation policy. Section 5 presents our conclusions.

2 Overview of Ficus

In order to convey the context in which our solutions to replicated volume location information management are employed, this section presents an overview of the Ficus replicated file system. Both its optimistic concurrency control and stackable layered structuring significantly influence the character of our solutions to configuring the name space.

Ficus is a replicated distributed file system which can be added easily to any versions of the UNIX operating system supporting a VFS interface. It is intended to provide highly reliable, available, and scalable filing service, both to local clusters of machines and to geographically distributed work groups. It assumes no limits on the number of sites in the network, the number of Ficus volumes, or on the number of volume replicas. However, while placing no hard limit on the number of replicas, Ficus does assume that there is seldom call for more than a small number of fully updatable replicas of any one volume and hence optimizes for the limited case.

Ficus supports very high availability for write, allowing uncoordinated updates when at least one replica is accessible. *No lost updates* semantics are guaranteed; conflicts are reliably detected and directory updates are automatically reconciled. Asynchronous update propagation is provided to accessible copies on a "best effort" basis, but is not relied upon for correct operation. Rather, periodic *reconciliation* ensures that, over time, all replicas converge to a common state. We argue that serializability is not provided in single machine UNIX file systems, and is not required in distributed file systems. This kind of policy seems necessary and appropriate for the scale and failure modes in a nation-wide file system. Details of the reconciliation algorithms may be found in [3, 11], while for more information about the architecture of Ficus see [10, 4].

2.1 Reconciliation

The essential elements of the optimistic replica consistency strategy are the reconciliation algorithms which ensure eventual mutual consistency of replicas. They are necessary since update is permitted during network partition, and since update propagation to accessible replicas is not

Appeared in the Proceedings of the
Summer USENIX Conference,
June 1991, pages 17-29

coordinated via a two-phase commit protocol. The algorithms guarantee that updates will eventually propagate and that conflicting updates will be reliably detected, even given the complex topology and failure modes of the internet communications environment. The reconciliation algorithms propagate information through intermediate sites in a "gossip" fashion to ensure that all replica sites learn of updates, even though any two replica hosts may rarely or never directly communicate with each other.

The optimistic consistency philosophy applies not only to updates to existing files, but also to the name space (creation and deletion of names and the side-effects of deleting the last name for a file). The Ficus reconciliation system is able to reconcile most "conflicting" directory updates since the semantics of operations on directories are well-known. The exception is the independent creation of two files with the same name (a name conflict) and the update of a file which is removed in another replica (a remove-update conflict) which the system can only detect and report. In the case of operations on arbitrary files where the system cannot know the semantics of the updates, Ficus guarantees only to detect and report all conflicts.

A reconciliation daemon running the algorithms periodically walks the tree of a volume replica, comparing the version vector of each file or directory with its counterpart in a remote replica of the volume. Directories are reconciled by taking the union of the entries in the two replicas less those that have been deleted. Consider for example, the case where two replicas, `A` and `B`, of a directory have been updated independently during a network partition. A new file name, `Brahms`, has been added to `A`, while the name, `Bach`, has been deleted from `B`. When these two replicas reconcile, it is clear that neither can be directly propagated over the other because an update-update conflict exists (both directory replicas have changed). However, given the simple semantics of directories, it is clear that the correct merging of the two replicas should contain the new name `Brahms`, and not the name `Bach`. This is what results in Ficus. The case where one replica knows about a file name create and the other does not is disambiguated from the case where one has the file's entry but the other has deleted it by marking deleted entries as "logically deleted". File names marked logically deleted may be forgotten about when it is known that all replicas know that all replicas have the name marked logically deleted (hence the two-phased nature of the algorithms). When the link count for a replica goes to zero, garbage collection may be initiated on the pages of the file; however, the file data cannot actually be freed until reconciliation determines that all replicas have a zero link count (no new names have been created), a dominant version exists, and it is the dominant replica being removed (there is no remove-update conflict).[3]

Each file replica has a version vector attached to it with a vector element for each replica of the file. Each time a file is updated, the version vector on the copy receiving the update is incremented. File replicas are reconciled by comparing their version vectors as described in [11]. As a part of reconciliation, a replica with a dominant version vector is propagated over the out-of-date replica which also receives the new version vector. If neither replica's vector is dominant, a write-write conflict is indicated on the file and a conflict mark is placed on the local replica, blocking normal access. Access to marked files is permitted via a special syntax in order to resolve conflicts. We will see how these same algorithms may be leveraged to manage the volume location data.

[3] The details of the two-phase algorithms for scavenging logically deleted directory entries and garbage collecting unreferenced files can be found in [1, 3]. The issues are somewhat more subtle than a first glance would suggest.

Appeared in the Proceedings of the
Summer USENIX Conference,
June 1991, pages 17-29

2.2 Stackable Layers

Ficus is also unique in its support for extensible filing environment features via stackable layered structuring. This architecture permits replication to co-exist with other independently implemented filing features and to remain largely independent of the underlying file system implementation. We briefly introduce the layered architecture here; for more details, see [4, 10, 2]

The stackable layers architecture provides a mechanism whereby new functionality can be added to a file system transparently to all other modules. This is in contrast to today's UNIX file systems in which substantial portions must be rebuilt in order to add a new feature. Each layer supports a symmetrical interface for both calls to it from above and with which it performs operations on the layers below. Consequently, a new layer can be inserted anywhere in the stack without disturbing (or even having source code to) the adjacent layers. For example, a caching layer might look for data in a cached local file, forwarding reads to the underlying layers representing a remote file in the case of a cache miss. Thus, stackable layers is an architecture for extensible file systems.

Each layer supports its own vnode type, and knows nothing about the type of the vnode stacked above or below[4]. The vnodes, which represent the abstract view of a given file at each layer, are organized in a singly linked list. The kernel holds a pointer to the head of the list only. Operations on the head of the list may be performed by that vnode, or forwarded down the stack until they reach a layer which implements them. At the base of the stack is a layer with no further descendants, typically a standard UNIX file system.

The term "stack" is somewhat of a misnomer as there may be both fan-out and fan-in in a stack. Fan-in occurs when a vnode is part of more than one stack (for example when a file is open concurrently on more than one site). Fan-out occurs when a single file at a higher layer is supported by several files at the next layer down. For example, in Ficus replication, a single file from the user's point of view is actually supported by several file replicas, each with its own underlying stack.

Replication in Ficus is implemented using two cooperating layers, a "logical" and a "physical" layer. The logical layer provides layers above with the abstraction of a single copy, highly available file; that is, the existence of multiple replicas is made transparent by the logical layer. The physical layer implements the abstraction of an individual replica of a replicated file. It uses whatever underlying storage service it is stacked upon (such as a UNIX file system or NFS) to store persistent copies of files, and manages the extended attributes about each file and directory entry. When the logical and physical layers execute on different machines, they may be separated by a "transport layer" which maps vnode operations across an RPC channel in a manner similar to NFS. Figure 1 illustrates a distributed file system configured with three replicas: one on a local UNIX box, one connected remotely via a transport layer, and a third stored on an IBM disk farm running MVS connected by NFS (with a UNIX box running the Ficus physical layer acting as a front-end). The replicated file system is shown mounted both on a UNIX workstation and a PC (via PC-NFS).

[4] The exception is when a service such as Ficus replication is implemented as a pair of cooperating layers, a layer may assume that somewhere beneath it in the stack is its partner.

Appeared in the Proceedings of the
Summer USENIX Conference,
June 1991, pages 17-29

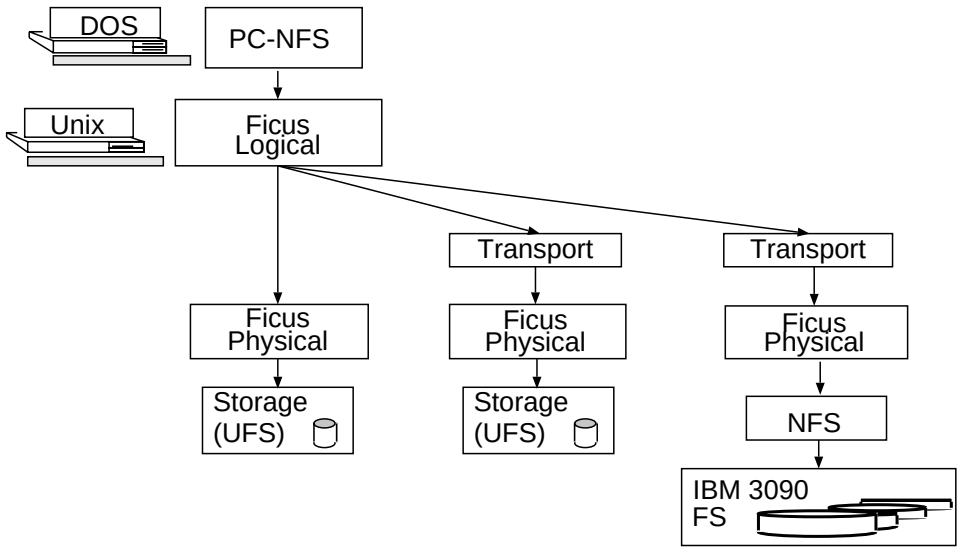

Figure 1: A Ficus layer configuration with three replicas.

2.3 Ficus Layers and Volumes

It is the Ficus logical layer which implements the concept of a volume. Each mounted logical layer supports exactly one logical volume. The Ficus physical layer supports the concept of a volume replica. Mounted under each logical layer is one physical layer per volume replica. The volume location problem concerns how an instance of a logical layer finds the instances of physical layers to attach, and how to move, add, or delete such physical layer instances. Consequently, it is the Ficus logical level which interprets the volume location information contained in graft points; as far as the physical layer is concerned, a graft point is a normal directory.

3 Volume Locating and Autografting

A volume is a self-contained rooted directed acyclic graph of files and directories. A volume replica is represented by a vfs structure (see [8]). Each volume replica must store a replica of the root directory, though storage of any other file or directory replica within the volume is optional.

3.1 Autografting

In a distributed file system which spans the Internet, there may be hundreds of thousands of volumes to which we might desire transparent access. However, any one machine will only ever access a very small percentage of the available volumes. It is therefore prudent to locate and

Appeared in the Proceedings of the
Summer USENIX Conference,
June 1991, pages 17-29

graft volumes on demand, rather than *a priori*. The main memory data structures associated with grafted volumes which are not accessed for some time may be reclaimed. Hence only those volumes which are actually in use on a given site take up resources on that machine. We refer to this on demand connection of the super-tree as *autografting*.

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 a priori requirement that the replication factor (how many replicas and their location in the network) of a graft point match, or even overlap, that of the child volume. If each site which stores a replica of the parent 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.

The graft point object 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 $\langle volid, replid, hostname \rangle$ identifying one replica of the volume to be grafted. The *volid* is a globally unique identifier for the volume. The *replid* identifies the specific volume replica to which the entry refers. The *hostname* is the internet address of the host which is believed to house the volume replica. The system then uses this information to select one or more of these replicas to graft. If later, the grafted volume replica is found not to store a replica of a particular file, the system can return to this point and graft additional replicas as needed.

In the course of expanding a path name (performing a `vop_lookup` operation on a vnode at the Ficus logical level), the node is first checked to see if it is a graft-point. If it is, and a volume is already grafted as indicated by a pointer to the root vnode of that volume, pathname expansion simply continues in that root directory. The root vnodes may be chained allowing more than one replica of the grafted volume to be grafted simultaneously. If no grafted root vnode is found, the system will select and autograft one or more volume replicas onto the graft point.

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 path name 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 [13]) and completes the graft.

A graft point caches the pointer to the root node of a grafted volume so that it does not have to be reinterpreted each time it is traversed. The graft of a volume replica which is not accessed for some time is automatically pruned so it does not continue to consume resources.

3.2 Creating, Deleting and Modifying Graft Points

Creating and deleting graft point objects require operations on the directory containing the graft point to add or remove a name for the object. Updates to the containing directory which create and delete names for graft points are handled identically to updates creating or deleting any entry

Appeared in the Proceedings of the
Summer USENIX Conference,
June 1991, pages 17-29

in a directory. Note that moving a graft point is equivalent to creating a copy of it in a different place in the name hierarchy, and deleting the original name. These updates change the way the super-tree of names is configured and hence are very rare.

The graft points themselves (as opposed to the directory containing them) are modified whenever a volume replica is created, dropped, or moved from one host to another. This type of operation is transparent to the user in that it does not affect the name space.

While updating a graft point is also a relatively rare event, when it does occur, it is generally important. Hence it is not reasonable to require that all, or even a majority of the replicas of the graft point be accessible. Further, the motivation for updating a graft point may be at its greatest precisely when the system is unstable or partitioned. Perhaps the whole reason for updating the graft point is to add an additional replica of a volume for which, due to partitions or node failures, only a single replica remains accessible; this update must be permitted, even though it cannot immediately propagate to all replicas of the graft point.

Hence, for exactly the same reason that Ficus utilizes an optimistic philosophy for maintaining the consistency of files and directories, the same philosophy must be applied to graft points. Rather than adopting a separate mechanism for maintenance of super-tree information, Ficus makes double use of its directory management algorithms. This is achieved by structuring graft points with exactly the same format as directories and, as a sequence of records, very similar semantics.

3.3 Update Propagation

All updates in Ficus, whether to files, directories, or graft points, are performed first on a single replica of the object. An update notification is then placed on a queue serviced by a daemon which attempts to send out update notifications to other accessible replicas. The notification message is placed on another queue at the receiving sites. Graft point and directory update notifications contain enough information to apply the update directly to the replica if the version vector attached to the message dominates the local replica. Normal file updates, on the other hand, are not piggybacked with the notification message and must instead be pulled over from an up-to-date replica. Update notification is on a one shot, best effort basis. Any replica which is inaccessible or otherwise fails to receive the notification will be brought up to date later by *reconciling* with another replica.

The reconciliation daemon running on behalf of each volume replica ensures eventual mutual consistency of both the replicated directories containing graft points, and the entries in the replicated tables. It periodically checks (directly or indirectly) every other replica to see if a newer version exists. If a newer version is found, it initiates update propagation; if an update conflict is found, a conflict mark is placed on the object which blocks normal access until the conflict is resolved. Access to marked objects is permitted via a special syntax for use in resolving conflicts.

Appeared in the Proceedings of the
Summer USENIX Conference,
June 1991, pages 17-29

4 Concurrent Updates to Graft Points

As with directories, the semantics of graft point updates are quite simple, and hence updates which would have to be considered conflicting if viewed from a purely syntactic point of view may be automatically merged to form a consistent result. For example, if in non-communicating partitions, two new replicas of the same volume are created, the two resulting graft point replicas will each have an entry that the other does not have. However, it is clear that the correct merged graft point should contain both entries, and this is what will occur.

The somewhat surprising result is that, unlike directories, there are no conflicts resulting from the optimistic management of the replicated graft point tables that cannot be automatically merged by the existing directory reconciliation algorithms. That is, entries in a graft point experience neither name conflicts or remove-update conflicts. However, as an entry in a directory, the graft point itself (as opposed to the data in the graft point), may be manipulated so as to generate a conflict. The reconciliation algorithms will automatically merge all updates to graft points, and reliable detect any name conflicts or remove-update conflicts on the directory in which the graft point resides.

To understand how the directory reconciliation algorithms double as graft point reconciliators, it is important to know how the graft point entries are mapped into directory entry format. The triple $\langle volid, replid, hostname \rangle$ is encoded in the file name field of the directory entry (the inode pointer field is not used). As a result, graft point semantics are even simpler than directories as name conflicts do not occur. Recall that if two different files are created in different copies of the same directory, a name conflict results, since, in UNIX semantics, names are unique within directories. However, in the case of graft points two independent entries in the table with the same $\langle\ volid, replid, hostname\ \rangle$ may be considered equivalent and cause no conflict[5].

A deleted entry in the graft point table (dropped replica) is indicated with the "logically deleted" bit turned on. This is used to distinguish between entries which have been deleted, and entries which never existed. Like directory entries, deleted graft point entries may be expunged when all replicas have the entry logically deleted, and all replicas know that all replicas know the entry is logically deleted (hence the two-phase nature of the reconciliation algorithms; see [3]).

Now consider moving a volume replica. When a replica storage site is changed, the $\langle\ hostname\ \rangle$ field in the graft point entry is updated. Since this corresponds to a rename (the field changing is part of the name in the directory entry view), it is carried out by creating a new entry and marking the old one as logically deleted. When the two replicas of the graft point table are reconciled, it will be found that a new entry exists and an old entry has been marked deleted; the out-of-date replica will update its graft point accordingly. It cannot occur that the same entry is moved independently in two copies of the graft point, since one needs to be able to access a replica in order to move it[6]. Hence if replica A of the graft point is updated to reflect a change of host for a volume replica, then no other replica of the graft point can update the host field until it has received the update propagation or reconciled with a copy which has, because until then, it will not be able to find the replica.

[5] Due to the way volume ids and replica ids are assigned, it is not possible to create different volumes or replicas independently and have them assigned the same id.

[6] This requires that moving a replica and updating the graft point must be done atomically so that replicas are not lost in the event of a crash.

Appeared in the Proceedings of the
Summer USENIX Conference,
June 1991, pages 17-29

4.1 Reconciliation of Directories Containing Graft Points

While no conflicts are possible within the replicated volume location tables, the graft points themselves are named objects in a replicated directory which may be subject to conflicting updates. As with uncoordinated updates to any replicated directory in Ficus, the reconciliation algorithms guarantee to merge the inconsistent replicas to create a coherent and mutually consistent version of the directory. The same exceptions also apply; reconciliation will reliably detect name conflicts or remove update conflicts, even if they involve graft points.

A name conflict occurs when in one copy of a directory a new graft point is created, while in another replica, another object (be it a graft point, ordinary file, etc.) is created with the same name. The system saves both entries in the reconciled directory, marking them in conflict, and allowing access by an extended disambiguating name for purposes of resolving the conflict.

A remove-update conflict is created when in one replica the last name for a graft point is deleted (link count becomes zero), while in another the graft point is modified internally (by adding or deleting replica entries). In this case, the graft point disappears from the directory in which it occurs, but is saved in an orphanage from which it may be later retrieved or deleted[7].

5 Conclusions

In order to provide a network transparent view of the file name space, all sites in the system must agree on what volumes are grafted where, and be able to locate a replica. Previous methods which rely on either a fully replicated mount table, informal coordination among system administrators, or replication only within a cell, fail when the system scales towards millions of sites. The solution to this problem in Ficus is predicated on the beliefs that the volume location data must be selectively replicated for performance and availability, and the replicas must be optimistically managed since a conservative approach restricts updates unacceptably.

This paper presents a solution to volume location data management in which the super-tree of volumes is glued together by graft points which are stored in the file hierarchy itself. Graft points may be replicated at any site at which the volume in which they occur is replicated. The same reconciliation daemons which recover from uncoordinated update to files and directories in Ficus also manage the graft point objects, detecting and propagating updates. By structuring graft points internally just like ordinary directories, no modifications are needed to the reconciliation implementation to support graft points. While it has always been our contention that the reconciliation algorithms which were originally developed for the express purpose of maintaining a hierarchical name space were applicable in a wider context, this re-use of the reconciliation daemons unmodified provides the first such evidence.

The graft point solution presented in this paper has been implemented and is operational in Ficus. Volumes are autografted on demand, and ungrafted when unused for several minutes. While actual use Ficus in a large scale environment is so far limited to installations including only a cluster of hosts at UCLA connected to individual machines at ISI and SRI, our initial experience

[7] This reflects our philosophy that the system should never throw away data in which interest has been expressed, in this case, by updating it.

Appeared in the Proceedings of the
Summer USENIX Conference,
June 1991, pages 17-29

with this approach to autografting is quite positive.

References

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

[2] 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*, pages 63–71. USENIX, June 1990.

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

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

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

[6] Michael L. Kazar, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Beth A. Bottos, Sailesh Chutani, Craig F. Everhart, W. Anthony Mason, Shu-Tsui Tu, and Edward R. Zayas. Decorum file system architectural overview. In *USENIX Conference Proceedings*, pages 151–163. USENIX, June 1990.

[7] Michael Leon Kazar. Synchronization and caching issues in the Andrew File System. In *USENIX Conference Proceedings*, pages 31–43. USENIX, February 1988.

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

[9] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The sprite network operating system. *IEEE Computer*, pages 23–36, February 1988.

[10] Thomas W. Page, Jr., Richard G. Guy, Gerald J. Popek, and John S. Heidemann. Architecture of the Ficus scalable replicated file system. Technical Report CSD-910005, University of California, Los Angeles, March 1991. Submitted for publication.

[11] 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.

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

[13] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network File System. In *USENIX Conference Proceedings*, pages 119–130. USENIX, June 1985.

Appeared in the Proceedings of the
Summer USENIX Conference,
June 1991, pages 17-29

[14] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[15] 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.