University of California
Los Angeles

Selective Replication:
Fine-Grain Control of Replicated Files

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Computer Science

by

David H. Ratner

1995

This document is available as Technical Report CSD-950007 of the UCLA Computer Science Department. A PostScript version is available via anonymous ftp to `ftp.cs.ucla.edu` as `pub/ficus/ucla_csd_950007.ps` or via the World Wide Web via `ftp://ftp.cs.ucla.edu/pub/ficus/`. The author can be reached by electronic mail by `ratner@cs.ucla.edu`.

This document was last changed January 18, 1995.

The thesis of David H. Ratner is approved.

Eli Gafni

Mario Gerla

Wesley W. Chu, Committee Co-Chair

Gerald J. Popek, Committee Co-Chair

University of California, Los Angeles

1995

They say that fear is only to warn us of danger,
not to make us afraid of it
-Anonymous

Table of Contents

List of Figures

List of Tables

Acknowledgments

This work would not have been possible without the support of my advisor, Dr. Gerald Popek, to whom I am most grateful. I would also like to thank the other members of the Ficus Project at UCLA. In particular I would like to thank John Heidemann for our many discussions regarding the vnode interface at the Ficus physical layer, Richard Guy for our discussions regarding the modifications of his garbage collection and reconciliation algorithms, and Dieter Rothmeier for his initial implementation of the NOREPL-type vnode.

I also would like to gratefully acknowledge the support of the Advanced Research Projects Agency. In the course of this research I was supported by ARPA contract N00174-91-C-0107.

Abstract of the Thesis

Selective Replication: Fine-Grain Control of Replicated Files

by

David H. Ratner
Master of Science in Computer Science
University of California, Los Angeles, 1995
Professor Gerald J. Popek, Co-Chair
Professor Wesley W. Chu, Co-Chair

Modern methods of reliable and efficient data sharing require replication of one form or another. Existing services take a number of different approaches, but none offer high data availability, fine-grained replication control, and non-restrictive communication models. As such, the existing services restrict how users can share data and the environments in which they operate. Systems which do not provide high availability cannot guarantee data access on demand. Coarse-granularity replication strategies force users both to replicate more than the desired data set and maintain consistency on the "unwanted" data, costing vital disk space and communication time. Services which impose restrictions regarding communication partners lack the flexibility to adapt to changes required by mobility. A fine-granularity, optimistic replication system solves these problems and allows users to work effectively and collaborate easily.

This thesis presents design issues and requirements of such a service as well as its implementation in the Ficus file system. The implementation incorporates a family of distributed algorithms to manage the propagation of changes to the filing environment. Additionally, the thesis illustrates that fine-grain control does

not alter performance from that of coarse-grain replication solutions.

CHAPTER 1

Introduction

The advent of the computer meant that users could now store large quantities of data in one central and easily-accessible location. In addition, the data could be processed and manipulated faster than ever before. Networking software, such as Sun's NFS, introduced the idea of straightforward data sharing — data could now easily be shared among a group of locally connected computers, increasing productivity by allowing multiple users to access the same data simultaneously. However, such localized sharing of data is giving way today to a more global form of sharing, in terms of the scale on which sharing occurs, the geographic distances separating the collaborators, and the demand for more ubiquitous file access. For example, consider a few of the modern methods of data sharing and their necessary communication characteristics.

Home-Office: Files are stored both at the office workstation and at the home personal computer. Work, potentially involving updates, is performed at both locations.

- Communication characteristics: occasional, expensive, slow.

Traveling Professional: Files are stored at the office server and on a personal laptop. In the office, the server and network are utilized, but when traveling or away from the office, updates and file modifications are performed on the laptop.

- Communication characteristics (connect laptop by modem): occasional, expensive, slow.
- Communication characteristics (connect laptop directly to network): occasional.

Collaborative-Work: Files and data are stored at several geographically separated locations so that distant project members can collaborate on common projects and easily work together.

- Communication characteristics: intermittent, unreliable.

We believe these new sharing models and their accompanying communication characteristics will require three major system services in order to fully exploit their potential.

Optimistic replication: Data must be distributed among a multitude of weakly-connected sites, and each site needs to be allowed to perform updates. Such characteristics imply that data must be replicated, and that reliance on voting, primary-site, or other conservative replication schemes will not be feasible. In each situation, users need the ability to update their local replicas, even when contact cannot be established to some or even any of the remaining replicas. Conservative replication strategies by definition cannot offer this ability uniformly. Instead, the above modes of use require *optimistic replication*, which allows independent updates to all replicas, detecting and resolving possible conflicts later [KS93, RHR+94].

Support for peer replication: The models' communication patterns are not always known a priori. A given developer may communicate with a variety of other collaborators, depending on the specific data subset and the colleagues' geographic locations. Mobile users may wish to directly connect laptops in a "portable-workgroup" mode to permit group development while traveling or at a conference, or may wish to connect directly to a new, local server when visiting other office sites. Finally, even home-use users may wish to connect occasionally to other sites than the usual office server, due to communication requirements, expense, or office failures. All these examples suggest that the system cannot restrict a given replica's possible communication partners. All replicas must be *peers*, to permit any-to-any communication topologies.

Fine-grain replication control: Existing optimistic services, such as Ficus [GHM+90] and Coda [SKK+90] replicate on the *volume* granularity, where a volume is defined to be smaller than a disk partition but larger than a directory [SHN+85]. For example, a user's home directory and all sub-directories might constitute a volume. The volume is used because it provides several key benefits, such as integrity, locality, and naming services. However, the volume is often too coarse a granularity for replication control. Collaborating colleagues want to locally replicate just the files with which they are directly involved, thus avoiding having to maintain consistency and physically store personally-unimportant data. Users connected by modem or other slow links want to avoid unnecessary communication time: replicating files which are never accessed nevertheless forces users to download their updates by virtue of replication. Finally, machines with limited data-storage capacity should still be allowed to replicate files stored on large capacity machines. However, replicating volume by volume would overload the smaller machine, and would hinder the sharing of data stored in multiple volumes between the two.

Providing useful replication in all of these environments requires an optimistic, peer replication service with control on a per-file basis. We have named this *selective replication*, and have integrated it with the Ficus file system.

The next section gives a brief overview of optimistic replication and the Ficus file system. Chapter 2 discusses replication design decisions, and the choices made in our selective replication service. Chapters 3, 4, and 5 describe the implementation, discussing the system modifications, the reconciliation, and garbage collection algorithms respectively. Performance data is presented in Chapter 6. The thesis concludes with a discussion of related work in this area (Chapter 7) and future work which could build upon or improve this service (Chapter 8).

1.1 Optimistic Replication and Ficus Overview

The guiding principle of optimistic replication is that if any replica is accessible, it should be available for full use, including update. In this way, optimistic replication provides the impression of a highly-available filing service, because even isolated or partitioned users connected to only a single replica can generate updates. However, allowing partitioned updates provides the potential for concurrent updates and resulting conflicts, so optimistic schemes must reliably detect these conflicts after the fact. Once detected, conflict resolution generally should take place before normal file activity can resume. In addition, optimistic schemes must have a method of notifying replicas about updates lost due to network partitions or system crashes. While all replication strategies must notify unknowledgeable replicas in one way or another, the problem is more difficult in optimistic replication due to the nature of disconnected updates. The potential for disconnected updates, or updates produced at isolated machines, implies that the lost-update notification scheme must be a two-way process, because when two partitioned machines resume communication, each could be knowledgeable of updates unknown by the others.

Ficus [Guy91, GHM+90] is one system that supports optimistic replication. Ficus provides a *peer-to-peer*, rather than client-server, model, meaning there is no master copy. Updates generated at any one replica are just as valid as those generated at any other. This notion of peers is also be referred to as *first-class* replication. In contrast *second-class* replicas, supported in other systems, are not peers; rather, they are more like copies of an original. They can only reconcile and exchange information with the master replica from which they were cloned. While less powerful, their implementation is often less complex, due to simpler control algorithms and their limited effect on the rest of the system.

The Ficus file system maintains consistency with a two-pronged attack. Update notification messages are sent to other accessible replicas at the time of the update, informing them of the change. However, these messages may not be deliverable to all or even any other replicas, due to failed communication media or machines. Thus, the update-notification messages are not guaranteed to maintain consistency. A process called *reconciliation* runs periodically, when communication can be established, to inform less-knowledgeable sites of new updates, and resolve any problems caused by the communication loss.

Reconciliation is organized as a "pull" of information from one replica to another; each replica reconciles with another by requesting the necessary data and status for shared files. The particular communication patterns form the *recon-*

ciliation topology. Many different topologies are possible, such as a ring, a star, or a random configuration. While the reconciliation algorithms are topology independent, the actual topology can affect both message complexity (the number of messages which must be exchanged between all replicas) and real-time performance (the length of time required to accomplish a user-visible action). Ficus uses a *ring-based* topology, in which replicas are assigned an order in a ring, and reconciliation typically proceeds only between ring neighbors. Such a topology utilizes *gossip communication*, meaning that a given replica can relay information to a second about the data structures and status of a third. The ring topology minimizes overall message complexity, but only operates correctly if the nodes are *gossip-connected*: that is, for each pair of replicas i and j, replica n_i can eventually learn about the status of n_j by communicating with some replica n_k.

Updates are tracked using *version vectors* [PPR+83]. A version vector is an array of monotonically-increasing counters, one counter per replica. Each counter i tracks the total number of known updates generated by replica i. Each replica independently maintains its own version vector for each replicated file; by comparing two version vectors, the update histories of the corresponding file replicas can be compared. The comparison function (comparing version vectors $V1$ and $V2$) has three possible output states:

$V1$ **dominates** $V2$: Each element of $V1$ is greater than or equal to its corresponding element of $V2$. If comparing version vectors of different lengths, the unmatched elements of $V2$ must be zero; the unmatched elements of $V1$ are irrelevant.

$V2$ **dominates** $V1$: The above case with the roles of $V1$ and $V2$ reversed.

$V1$ **and** $V2$ **conflict** : $V1$ does not dominate $V2$ and $V2$ does not dominate $V1$.

This definition allows for two version vectors to both dominate each other if they are equivalent element by element. This reflects the peer nature of replication in Ficus.

When version-vectors are found to conflict, their respective file replicas are said to be "in conflict". Special mechanisms must be invoked to resolve the conflict [RHR+94]. When one version-vector dominates the other, the corresponding replica's data is more recent. As replicas communicate and the pairwise version-vector comparison continues throughout the network, the most recent data propagates, and eventually all replicas converge to a common global state.

As part of reconciliation, Ficus contains a distributed algorithm to perform *garbage collection* — the deallocation of resources held by inaccessible file system objects. While a relatively simple process in a centralized system, distributed systems face new problems in detecting the correct time to garbage collect. Detecting that a file has no links locally is relatively trivial, but verifying that there are no links globally across all replicas is more complex, due to dynamic naming. For example, a file could only transiently have no local links, because a new name for the file might exist at another replica. A global consensus must be reached to confirm the global number of links. In addition, distributed garbage collection is made more difficult by the possibility of long-term communication barriers. Since any replica could potentially know about new links, garbage collection cannot complete until all replicas have been heard from at least once. Ficus uses a fully-distributed two-phase algorithm to ensure that all replicas are knowledgeable of the garbage collection process and can eventually complete it, although all participants may never be present simultaneously [GPP93].

Ficus replicates at the volume granularity. Volumes can be created at arbitrary positions in the namespace using simple tools. In addition, volumes can be dynamically added, deleted, or moved at any time. Users wishing to share data first create a volume and then add volume replicas at all necessary remote sites. However, users need not store a local replica of the volume in order to access the data within. The physical location of a volume replica is transparent to the user, as Ficus provides transparent access to remote volume replicas. Ficus automatically selects an appropriate replica to access using selection criteria.

Ficus has been in operation since early 1991, and as of December of 1994 supports the research efforts of thirteen users at the University of California at Los Angeles. In addition, Ficus is operational at Trusted Information Systems and at the Ohio State University. Ficus is constructed from modified SunOS 4.1.1 source code. The modifications currently consist of approximately 42,000 lines of kernel code and 26,000 lines of user-level utility code.

CHAPTER 2

Selective Replication Design

The design of any replication service must begin by asking fundamental questions about the desired quality and type of replication. The solutions must consider the service's target environment as well as implementation complexity. For instance, certain questions have already been answered by the proposed usage in the three sharing models of Chapter 1. Due to the models' communication properties, optimistic replication is required, and thus we have already answered (at a high level) the question of how consistency will be maintained. Other questions remain, however, which must be addressed before the service can be designed. These are:

1. What replication granularity is required?
2. What class of service should be provided?
3. How will replicas communicate to maintain consistency?
4. What kinds of performance requirements should be enforced?

Each of these will be discussed in detail below.

2.1 Replication granularity

The volume is a useful concept for several reasons:

Locality and Performance: Logically-connected files can be grouped together in one physical place. Performance-intensive tasks need only be done once for the volume, rather than once per file in the volume.

Integrity: Volumes provide natural firewalls for preventing the propagation of errors and for establishing fundamental security barriers.

Naming: A collection of logically-connected files can easily be identified and acted upon as a single unit.

However, these functions are orthogonal to the actual replication of the files within the volumes. Furthermore, replicating at the volume granularity restricts the types of data sharing which can be employed. Several alternate choices exist, such as:

1. The sub-volume
2. The directory

3. The file

Each of these will now be discussed.

2.1.1 Sub-volume granularity

A new concept, called the "sub-volume", could be defined and used as the unit of replication. It would be smaller than a volume but larger than a directory, designed to ease the "large-volume" problem. However, the sub-volume is a non-tangible concept, much like the volume itself. Replicating at the sub-volume level does not provide the user with real fine-grain control, because the sub-volume is still a large container. Users desiring replication control at this granularity could simply enforce a maximum volume size to achieve the same result.

2.1.2 Directory granularity

Directory-granular replication is an interesting choice. On the surface, it seems to provide the flexibility required, because directories are relatively small and fine-grain. Yet, replicating at the directory does not address many common scenarios. For example, it is not uncommon for a single directory to contain several different types of files, such as both source and derived object code. Developers with limited disk space would probably only want to replicate the smaller source code files: the object code files are larger and not necessary for development, just the actual building of the executable. A second common scenario are directories containing large files, and users only want to replicate a subset of them, due to their size. Such directories could contain sounds, images, or even large reports.

The above examples could be addressed by requiring the user to break each single directory into multiple directories. Building on the first example above, the source code could become one sub-directory, and the object code another. While clearly a possible solution, such a model forces the user to alter the namespace and normal pattern of work to permit fine-grain replication. In fact, such a policy basically admits that users desire the ability to replicate on a finer granularity than the directory but does not support it. In conclusion, the desired grouping for replication control often differs from the grouping as defined by semantic relationships, such as those which govern directory usage.

2.1.3 File granularity

A better solution is to replicate at the file granularity. It offers the users the flexibility of control without requiring altering the namespace organization and semantic groupings of files. The file granularity provides fine-grain control and allows users the freedom to select individual files from directories for replication.

File granularity replication does add complexity to the system, both in terms of implementation and user or administrator management. Implementation complexity arises from the new algorithms, data structures, and system operation which must exist to support file granularity replication. Administration complexity increases because the finer-granularity of control leads to more possible choices when replicating data. Additionally, new user-level utilities must be added to the system, increasing the set of available commands. However, it is not more expensive than other granularity choices, as illustrated by the performance study in Chapter 6.

2.1.4 A two-level granularity structure

Our selective replication service actually combines two of the above choices to form a two-level granularity structure. Rather than simply making the file the ipso facto replication unit, volumes are retained for all of the reasons outlined in Section 2.1. The volume is ideal for integrity, locality, and naming purposes. However, it does not offer the user the replication freedom required. Therefore, within the coarse-grained volume, fine-grain replication control at the file level is provided. Tools, analogous to their volume replica counterparts, permit dynamic addition and deletion of file replicas. Files not stored locally are transparently accessed via remote file replicas. Replication *masks*, or mappings onto the set of volume replicas, indicate where files within the volume should be replicated.

The two-level granularity model allows for the necessary system functionality to be logically separated by function granularity: those which performance reasons dictate must operate on a large-granularity object, and those which users dictate need to be finely-controllable. For example, security perimeters erected around a single large object retard attacks more effectively and efficiently than many small perimeters around individual objects. Security holes and accompanying barriers need only be dealt with once for all objects within the large container, thus amortizing the cost across all individual entities. Since a volume has only one entry point, namely the volume root, there is only a single point that requires entry security and protection. If intruders cannot even enter the volume, there is no need to pay additional expense to protect each individual file.

Like security perimeters, other features can be enabled or disabled on a volume-by-volume basis so as to distribute the cost over all volume members. Large containers provide simple inheritance mechanisms, allowing the files inside to automatically inherit properties and attributes of the volume, such as encryption, authentication, or compression. Finally, coarse-grained containers provide a trivial way of naming all members.

In contrast, usage modes dictate that replication controls need to be finely controllable. Without doing so, users are severely restricted in terms of potential modes of use and models of operation. Thus, we present the user with a two-level granularity structure to provide performance, efficiency, and replication freedom.

2.2 Class of replication

Regardless of the actual replication granularity, there are two basic replication quality choices. These are identified as *first-class* and *second-class*. First-class replicas, or *peers*, are equals; second-class replicas are clones of a parent, first-class replica. Any peer replica can communicate with any other peer replica, but second-class replicas can only communicate with their one parent replica. Additionally, updates generated at second-class replicas must be "acknowledged" by their parent replicas before being introduced to the other first-class replicas as a valid update; such is not the case with peer-generated updates. The two replication classes will be discussed in more depth below.

2.2.1 Second-class

Second-class replication has several advantages, perhaps the biggest being implementation simplicity. Because second-class replicas only interact with the system in very limited ways — notably in their communication and update patterns — their implementation may prove to be simpler, although the system as a whole may be made more complicated by the presence of two distinct replication classes. Additionally, second-class replicas may prove to be more light-weight than first-class objects, in terms of creation and destruction cost, again because of their limited affect on the system. Second-class replicas in this respect are clearly preferred for read-only caching, when no requirement for update exists. Nevertheless, second-class replicas suffer from two distinct disadvantages: their communication patterns and update-generating ability. Since they cannot communicate among themselves, and can in fact only do so with their particular parent replica, the following uses are precluded:

- Working at another office site, across town or across the world. Users operating in this manner want to interact directly with the local office's replicas, rather than communicating long distances via the communal and potentially slower Internet. When users change geographic location, they require a seamless and rapid adjustment to the new environment.

- Connecting multiple portables at a conference, on a plane, or at the beach and operating in a "portable-workgroup" mode. Users want the ability to directly exchange updates and reconcile data.

In addition, since updates generated at second-class replicas are themselves second-class updates, they cannot be handled like an update from a first-class replica. Specifically, the conflict-resolution process suffers as a direct result of the second-class nature. Updates generated at second-class replicas which, if integrated, would create conflicts with the first-class replica are essentially aborted, and must be re-integrated by hand at a later time [KS92]. Such actions can be painful and time-consuming for the user, especially since first-class updates that create conflicts can often be automatically resolved [RHR+94].

2.2.2 First-class

On the opposite side of the replication quality spectrum lies first-class replication. First-class replication, while suffering from additional complexity, provides increased functionality, most notably the ability for any-to-any communication. Allowing any replica to communicate and exchange updates with any other replica is strictly more powerful, and includes the above two examples as possible operational modes. However, it does require that all replicas learn of the existence of each new one, possibly making the addition of a first-class replica a more heavy-weight operation.

Additionally, first-class replication provides guarantees to the updaters that their modifications will not be lost or discarded. This guarantee, while not necessary in a read-only environment, becomes extremely important when postulating mostly-disconnected environments with high update potential.

2.2.3 Different granularity classes of service

Just as different replication granularities were combined above, different classes of service could be combined in a multi-granular replication scheme. Each level of granularity could potentially provide a different class of service. However, the above discussion illustrates that the replication unit needs to be a first-class object, and therefore we have designed a system with first-class replication at the file granularity. Other choices were available, such as first-class volume replication and second-class file replication, but these are not targeted at our hypothesized environments and sharing models.

2.3 Maintaining consistency

The heart of a replicated filing system is the maintenance of consistency across all replicas. Without consistency, file versions diverge, and users see different states depending on the particular replica accessed. Many methods for maintaining consistency exist, including both optimistic and conservative synchronization methods. Conservative schemes reject updates if a majority vote cannot occur or if a primary site cannot be accessed, and for these reasons it has already been argued in Chapter 1 that optimistic replication is required. Yet, there are multiple choices regarding how to maintain consistency in the optimistic case. Furthermore, maintaining consistency actually involves two separable processes: keeping the file data consistent, and freeing the resources as quickly as possible once they are inaccessible by the users. These two processes will be discussed further, as we define exactly how consistency will be maintained.

2.3.1 Keeping file data consistent

Ficus maintains data consistency in two distinct manners. First, Ficus recognizes that in many types of environments, such as a local area network (LAN), machines are almost always operational and accessible. Ficus takes advantage of such connectivity with a process called *update-propagation*, which is a best-effort broadcast of update notification to all available replicas. The propagation process is orthogonal, and therefore unaffected by, the granularity of replication or the class of service. Updates are only propagated to other known replicas of the updated file. In environments that are either always fully-connected or utilized guaranteed message queuing and delivery services, propagation would guarantee data consistency.

Yet, Ficus is not aimed at these environments, and propagation is not a reliable service. Therefore, Ficus has a second process called *reconciliation* which executes at periodic intervals and guarantees consistency. Reconciliation executes independently at each replica by periodically having it select an "appropriate" other replica, communicate with it, and perform a pairwise comparison of each file (using version vectors). Upon termination, reconciliation guarantees that the first replica has learned about all updates known to the second. As the pairwise processes continues at all replicas, all updates are guaranteed to be incorporated and distributed.

The central issue affected by replication granularity is the *reconciliation topology*, the rules governing how an individual replica chooses an "appropriate" remote replica for reconciliation purposes. Many different topologies are possible, and each configuration has different implications on message complexity, adaptability, and time required to distribute updates to all sites. In order for a reconciliation topology to work well in the types of environments postulated in Chapter 1, it will have to exhibit certain properties:

1. The reconciliation topology must adapt to changing network topologies.
2. The reconciliation topology must allow sharing between machines which never directly communicate, but do communicate with intermediaries that also participate in the sharing relation.

Replication granularity affects topology in that the coarser the granularity, the simpler the topology becomes. Simplicity occurs primarily because coarse-granularity creates less objects that need to be topologically-connected. Replication quality (first or second-class) also affects topology in that second-class replicas can only cause limited changes.

Our implementation of a reconciliation topology, which satisfies the above properties, is discussed in Chapter 4.

2.3.2 Freeing the resources

Garbage collection is the process that deallocates system resources held by inaccessible file system objects. Without proper garbage collection, disk space is either prematurely freed, resulting in loss of data, or perhaps even never freed, decreasing the user-usable disk space.

The removal of the last pointer to a given file initiates the garbage collection process. However, since the only action that can be detected locally is the removal of the last *local* link, garbage collection cannot complete until it can be determined that all links *globally* have been removed. The local state of "no links" can be a transient case, as new names are created at remote replicas and learned later via reconciliation; the global state of "no links" is persistent.

Guy developed algorithms in [GPP93] for garbage collection in a distributed environment which ensure that all participants terminate correctly and tolerate long-term communication failures. However, his solution does not apply directly to selective replication, so we modified it, as discussed in Chapter 5.

2.4 Performance

The two above distributed algorithms, reconciliation and garbage collection, impact the system in user-visible ways, and therefore their performance is a key criteria in the system evaluation. Poor reconciliation performance impacts the time to propagate updates to all replicas and the system load while reconciliation executes. Similarly, poor garbage collection performance increases the time required to garbage collect files, and therefore decreases the user-usable disk space for a longer period than necessary. It is therefore important to place performance bounds on these two algorithms.

2.4.1 Reconciliation

We would like to place the following bounds on the reconciliation process:

1. A reconciliation by site A with site B of a single file should be an O(1) operation.
2. The reconciliation by site A with site B of an entire volume replica should be an $O(N)$ operation, where N is the number of files locally-stored at replica A.

These two bounds convey our desire to ensure that reconciliation is as lightweight as possible, and does not adversely interfere with the user's normal operation. Additionally, they imply that replicas which store only a subset of the files within a volume only pay a reconciliation cost proportional to the size of that subset.

2.4.2 Garbage Collection

We would like to place the following bound on the garbage collection process:

> Garbage collection of an r-replica file should involve only $O(r)$ replicas and take at most $O(r)$ replica-steps, even when there are $R \gg r$ volume replicas. A replica-step is defined to be a single action performed by a particular replica.

This bound encompasses the idea that time to garbage collect should be directly related to the number of replicas. Since any possible garbage collection algorithm must involve contacting all file replicas at least once, $O(r)$ replica-steps is in fact the minimum required to garbage collect in environments with possible network partitions and long-term communication delays.

2.4.3 Performance solutions

The solutions which follow in the next chapters meet all the performance requirements we have just enumerated. Specifically, reconciliation is described in Chapter 4 and garbage collection in Chapter 5, and both algorithms satisfy the above bounds.

CHAPTER 3

Selective Replication Architecture

With the key questions of Chapter 2 answered, an implementation that meets those criteria is needed. In this chapter we will explain how our selective replication design and implementation meets two of those criteria, namely how to provide first-class replication, and how to implement the two-level granularity structure. The maintenance of data consistency will be examined in Chapters 4 (reconciliation) and 5 (garbage collection).

We will begin with a discussion of first-class replication, and continue with a discussion of the two-level granularity structure, which will include:

- New data structures necessary to implement the two-level hierarchy
- System invariants enforced by the two-level hierarchy
- Tools for controlling replication factors at both granularities
- Replica selection

3.1 First-class replication

Ficus already provides first-class replication at the volume granularity [GHM$^+$90]. We build on this base to provide first-class replication at the file granularity by forcing all participants to maintain a volume replica for each file under replication control. Storing the volume replica guarantees that all objects within will be first-class objects. However, storing a complete volume replica to provide replication for a subset of files is clearly overkill. *Partial volume replicas* solve the problem. A partial volume replica maintains the volume data structures and replica information, but only for a select number of files within the volume, with no minimum number enforced. Partial volume replicas provide a simple first-class replication mechanism while exhibiting low management cost.

3.2 Two-level granularity structure

Partial volume replicas are the key to the two-level granularity structure. They provide the ability to maintain a volume replica while storing only a few of the volume's files. However, new mechanism becomes necessary to locate all replicas of a given file, because the default mechanism of Ficus — merely locate each volume replica — no longer suffices. Each file can now be replicated at essentially an arbitrary subset of the volume replicas, so a method is required to indicate the particular subset. Our approach is built around a *status vector*, described below.

Additionally, the partial replica by itself lacks a solid definition, and therefore invariants are required to maintain a structure which is fault-tolerant to network partitions and supports transparent file access of remote replicas. We want the two-level structure to ensure that locally-stored files are always accessible, even when completely disconnected from all other replicas. Furthermore, when connected to other replicas, we want to provide transparent, fast access to the files stored there: that is, maintaining the impression of one global namespace regardless of the data's replication factors. The two invariants used to provide these features are the *full backstoring* and *directory-entry* invariants. Both the status vector and the two invariants will now be described.

3.2.1 Status vector

The *status vector* is the mechanism used to indicate which volume replicas store individual files. One status vector is maintained at each volume replica for each locally-stored file. Stored with the file's extended attributes, the status vector indicates the subset of volume replicas that physically store the given object. The status vector is constructed of N elements, where N is the number of volume replicas. Each element indicates whether the corresponding volume replica stores the file by means of storage *status* values. Thus, the basic status vector structure is an array of length N, with each array element being a tuple of the form

```
{volume replica identifier, status value}
```

Given a status vector sv for file replica r, denoted as sv_r, and the index into the status vector i with $sv_r[i].replica == r$, replica r's status value is stored in $sv_r[i].status$.

As far as the user is concerned, a particular status element can have one of two values: the file is either stored or not stored at the given volume replica. These are referred to as the DATA and NOTHING status values, respectively. For example, the status vectors for a given file at each of three volume replicas are illustrated in Table 3.1. The example shows that volume replicas only store status vectors for locally-stored files.

Since the status vectors are independently and optimistically maintained structures, partitioned updates can generate inconsistencies, resulting in *conflicting* status vectors. An example of such a conflict would be if volume replica 1 had in its status vector the tuple $\{2, \text{DATA}\}$ while volume replica 3 had the tuple $\{2, \text{NOTHING}\}$. Without additional mechanism, it is impossible to identify the true status value for replica 2, since not even replica 2 can be the authority on its

Volume Replica	Status Vector
1	{(1, DATA), (2, NOTHING), (3, DATA)}
2	none stored
3	{(1, DATA), (2, NOTHING), (3, DATA)}

Table 3.1: A status vector example. There are three volumes replicas, and the file in question is replicated at only two of them (replicas 1 and 3). The status vector is shown as an array of tuples of the form {volume replica identifier, status value}.

own status value once we allow replicas to modify other replica's status elements on their behalf, as described in Section 3.3. For example, we allow a site A to add file replicas at a site B, on B's behalf and possibly without even B's knowledge.

The conflicts are dealt with by adapting the version vector [PPR+83] technology to the status vector. Each status element is actually implemented as a monotonically-increasing counter, like the elements of the version vector. The status value is determined by applying a mapping function from the set of integers to the set of status values. Conflicts are now trivially resolved by applying the greater-than function, and comparing status values in their integer forms. The version vector concept provides a simple method to determine which of two status values is more recent, and therefore provides automatic resolution for conflicting status vectors.

3.2.2 Two-level granularity structure invariants

The two-level granularity scheme maintains two structural invariants in order to be fault-tolerant to network partitions and support transparent file access to remote replicas. These are the *full backstoring* and *directory-entry* invariants.

3.2.2.1 The full backstoring invariant

The set of all locally-stored files at a given volume replica forms a forest of trees. If the trees are disconnected, connection to at least one other volume replica is required to traverse through the intermediate directories (not stored locally) which logically connect the disconnected trees. Tree walks are normal activities

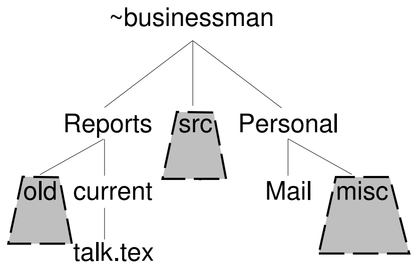

Figure 3.1: An instance of full backstoring. In order to store the `talk.tex` and `Mail` files, the intermediate directories must be stored as well. Shaded subtrees are not stored locally.

for the system (the reconciliation process for example) and the user. Not allowing tree walks when the intermediate directories are inaccessible implies that the system operates incorrectly during periods of disconnection — disconnection is treated as an unusual, rather than typical, occurrence. Therefore, the user is forced to understand more than should be necessary about the physical layout of the tree structure onto the set of volume replicas, in order to operate satisfactorily during disconnection.

A superior mode of operation is achieved by enforcing the invariant of *full backstoring*. Full backstoring means that for each locally-stored file, the parent directory must also be stored locally (the invariant does not apply across volume boundaries). It is enforced by out-of-kernel replication tools, so the kernel is not burdened with unneeded complexity, and so the policy can be easily modified. Figure 3.1 illustrates a simple example of full backstoring. While traveling to a conference, the businessman needs a replica of the invited talk (`talk.tex`), so he must also store the intermediate directories between the volume root and the file. However, he has no need for the source code (`src`) or the old reports (`old`), so these subtrees are not stored locally. He might want to review the week's mail, however, so the `Mail` directory and the intermediate directory `Personal` is also stored.

An obvious alternative to full backstoring are "prefix pointers" which indicate the direct path from the root to each subtree in the forest. Such a method might be a more light-weight solution, as the intermediate directories would not have to be stored locally and the time required to maintain consistency on them would therefore be saved. However, maintaining these pointers and integrating them into the common Unix functions like "pwd" and "cd" would be difficult. Furthermore, prefix pointers are themselves yet another tree-based structure to maintain — the benefit gained by maintaining two distinct but equivalent tree-based structures is questionable. Since directories are relatively small in size compared to files, the added storage imposed by full backstoring is relatively minor. Full backstoring strikes a good balance between functionality and complexity.

Hard links pose certain problems for full backstoring. Hard links are additional pathnames for a file which, in contrast to soft or symbolic links, reference the file directly, as opposed to through an indirection pointer. Even if a file has hard links to it, the replication tools only see one name — the one presented by the user. Given one name, it is impossible to find hard links to the same file without scanning the entire global file system, which is clearly impractical. Therefore, only the path presented by the user gets fully backstored. Other paths will succeed in naming the file when connected to remote replicas that store the particular intermediate directories, but they may not work locally. However, this is not a serious problem, because the local user is the one who selects the path to be backstored at replica addition time. If the user requires the additional paths to be fully backstored as well, the replication tools must be invoked with the additional names.

3.2.2.2 The directory-entry invariant

Selective replication allows files to be stored at different locations than the parent directory. For example, consider the situation at the root of the volume. Users who only want to replicate a particular subtree (or subtrees) will only locally store the directory entries corresponding to that subtree (or subtrees), such as in Figure 3.1. The replica in the example will have no local knowledge about the physical location of the other subtrees, such as the `src` subtree. When a connection exists to other volume replicas, access to the non-locally stored subtrees should be transparent to the user. Selective replication should neither break name transparency nor incur unreasonable access times for files not stored at the local replica.

The question at hand is how to best perform the transparent access. The set of volume replicas could be quite large, and therefore polling on each file access is not an efficient option. Instead, we introduce a new status value, in addition to the basic two from Section 3.2.1. The new status value is for maintaining locally just the status vector for a particular file, without storing the file's data. Since the status vector indicates which volume replicas store data for the file, it provides a fast and easy method of locating file replicas. The new status value is referred to as an ATTRIBUTES-ONLY state, as only the extended attributes (with the status vector) are maintained[1]. Since the status vector is a small number of bytes (20 bytes times the number of volume replicas), this solution consumes little disk space.

The system maintains the invariant that, for all locally-stored directories, all directory entries are also stored locally in at least the ATTRIBUTES-ONLY state (they could be stored in a DATA state). Thus, in the example of Figure 3.1, the files `src`, `misc`, and `old` would be maintained in an ATTRIBUTES-ONLY state at the replica shown.

The status vector is used only as a hint to the physical storage locations. As stated before, since it is an optimistically and independently maintained structure, partitioned updates can create temporary inconsistencies. If the system finds that the local status vector is completely invalid (due to rapid shuffling of storage locations), it must fall back to other policies to find a given file, explained later in Section 3.4.

3.3 Replication controls

Volume granularity replication services have a single policy with regard to file creation — all files are replicated at all volume replicas. As such, the only controls provided are those which create, add, delete, or move volumes. A selective replication service, in contrast, needs to allow the existence of multiple policies regarding file creation, and additionally allow user-configurable policies. Furthermore, regardless of the accuracy of either the default or user-configurable policy, users' minds change and data demand patterns shift. Such changes imply the necessity for tools that provide dynamic addition and deletion of file replicas. This section will discuss the replication policies as well as the selective replication tools.

3.3.1 Default replication policy

Default replication policies are necessary for the novice and unknowledgeable user and in general for users who do not want to be bothered by file storage issues. There are multiple choices for a default, such as files could be stored:

- Only at the local replica
- At all replicas

[1]Currently, not all of the extended attributes are maintained. Only maintenance of the status vector is currently implemented.

- At all replicas where the parent directory is stored

Having only one copy, at the local replica, minimizes the benefits of replication, as file sharing needs to become much more explicit. In contrast, replicating at all replicas quickly approaches a fully-replicated environment, especially given the full backstoring invariant described in Section 3.2.2.

We have selected a default replication factor equal to the current replication factor of the file's parent directory: new files by default are stored wherever the parent directory is stored. This seems to offer the desired flexibility without being overly limiting. However, a potential problem with this scheme is that since the volume root is by definition fully replicated, new files created under the volume root are, by default, stored everywhere. This problem calls for additional mechanism, such as user-configurable replication policies.

3.3.2 User-configurable replication policies

The basic problem with the default replication policy is that it does not differentiate between where the directory is stored and where the children should be stored. User-specified replication *masks*, or mappings onto the set of volume replicas, provide a solution. On a per-directory basis, users can specify where the directory's children should be stored. The full backstoring invariant (Section 3.2.2) implies the children can only be stored at subsets of the parent's storage sites.

3.3.3 Dynamic replica deletion

There are two basic issues concerning the deletion of file replicas. The first is the potential loss of updates. The second is the resolution of the create/delete ambiguity, first noted by Fischer and Michael in [FM82]. Both of these will be examined further below, followed by the replica deletion algorithm.

3.3.3.1 The potential loss of updates

A clear distinction exists between a file removal and a file replica deletion: in the former, the link (and the file itself if it is the last link) is expected to be removed globally, whereas in the latter only the specified file replica is supposed to disappear. Replica deletion leaves the file still in existence globally, and even locally accessible via a remote replica when network-connected. The expected semantics of the file replica deletion are that "no data is lost and the file is no longer stored locally." However, these semantics are difficult to enforce if the to-be-deleted replica contains the most recent data, especially since it is not possible in general to determine if a particular replica has the most recent data given only local information.

A method is required to force another replica to initiate a reconciliation with the to-be-deleted replica, to ensure that data is not lost. This process, called a *reverse-reconciliation*, is not always possible, however. Consider the mobile user on a plane or the user at home working during a phone outage. These users will not be able to initiate reverse-reconciliations due to the lack of communication media. Yet, there are very good reasons for allowing replica deletions in these cases, despite the potential loss of updates — the freeing of vital disk space. When disks become full, all work ceases, and often the possibility of losing an update is less important than the probability of not being able to continue working. Therefore, we allow users to bypass the reverse-reconciliation if necessary.

3.3.4 The create/delete ambiguity

The basic create/delete ambiguity is that object deletion at one site can be indistinguishable from object creation at another site, and resolving the ambiguity requires more information than simply the object's presence at the first site and absence at the second. In the context of file replica deletion, the ambiguity manifests itself as follows. A file `problem-file` has two replicas, A and B. Replica A is deleted, unbeknownst to replica B. When replicas A and B communicate via reconciliation, two possible scenarios are possible. `Problem-file` could be newly-created at site B, in which case it should be stored at replica A. Alternatively, A's replica of `problem-file` could have been deleted, in which case site B needs to update its status vector. This is the essence of the ambiguity. Replicas A and B cannot decide if the file should be added by A or was at some point in the past deleted by A.

A possible resolution of the ambiguity would be to maintain status vectors locally even after the local replica has been deleted. This would in a sense maintain the "history" of all actions that have occurred, and thus resolve the ambiguity. However, such a solution has unattractive properties. There is clearly a point in the future when, after all replicas have learned that replica A deleted its local replica, the status vector need not be maintained any longer at replica A. Maintaining it is unsatisfactory since a file replica deletion causes remnants to forever remain locally.

A superior method is to adapt Guy's solution for resolving the create/delete ambiguity in garbage collection of file system objects, discussed in [GPP93]. Guy keeps a record of the action temporarily and executes a distributed algorithm to guarantee both that all replicas learn of the action and that the record is eventually removed. In our case, the record of the action is a new status value, in addition to the three already mentioned (two from Section 3.2.1 and one from Section 3.2.2). It is called the DROPPING state. When one or more file replicas change state to the DROPPING state, all data-storing replicas participate in a distributed algorithm similar to Guy's garbage collection algorithm. This algorithm, called the *dropping-notification* algorithm, guarantees that all replicas correctly learn of the transition by one or more replicas into the DROPPING state. The algorithm also guarantees that, after all have learned of the DROPPING transition, the state is changed from DROPPING to NOTHING. When the site that actually deleted the file replica changes its local state to NOTHING, its local status vector is removed, thus cleanly completing the replica deletion.

3.3.4.1 Replica deletion algorithm

The actual replica dropping algorithm is as follows. Note that the algorithm allows remote replica deletions, that is deletions which occur on behalf of a specific replica but not physically at the specific replica.

```
/*
 * Procedure Delete_Replica:
 * Delete replica R of file F
 */
Procedure Delete_Replica(file F, replica R)
begin
    attempt to reverse reconcile the file
        (unless we are performing a remote replica deletion)
    if any of F's sibling files are also stored at replica R then
        /*
         * The parent directory must be stored for the sibling files.
         * Therefore, we must preserve the directory-entry invariant
         */
        change replica R of file F's status value to ATTRIBUTES-ONLY
    else
        /*
         * Drop the file, begin the dropping-notification algorithm,
         * and remove any full-backstoring that existed
```

```
         * for the storing of this file
         */
        change replica R of file F's status value to DROPPING
        begin dropping-notification algorithm on file F
        Delete_Replica(parent(F), R)
end
```

The recursive call to delete the file's parent is guaranteed to terminate before reaching the root of the volume. If the parent of the file f is the volume root, then the test to see if any of f's siblings are also stored will always succeed, because the volume root contains special files which must be stored at all volume replicas. Therefore, the recursion is guaranteed to terminate.

3.3.5 Dynamic replica addition

Adding file replicas is simpler than deleting them, because there is no create/delete ambiguity to resolve: the addition of the DROPPING state implies that the default action is to create. The only real issue is to guarantee the full backstoring invariant of Section 3.2.2. If the invariant is not maintained, locally-stored data may be inaccessible during times of disconnection. Therefore, all intermediate directories on the path from the volume root to the to-be-added file must be added as well. Fortunately, the addition of these intermediate directories is relatively simple, because they must be accessible at the time of the file replica addition; if they were not, the file itself would not be nameable.

3.3.5.1 Replica addition algorithm

The actual replica addition algorithm is as follows. Note that the algorithm allows remote replica additions, that is additions which occur on behalf of a specific replica but not physically at the specific replica.

```
/*
 * Procedure Add_Replica:
 * Add replica R of file F
 */
Procedure Add_Replica(file F, replica R)
begin
    /*
     * Ensure full backstoring
```

```
     */
    if parent(F) is not stored at replica R then
        Add_Replica(parent(F), R)

    /*
     * Now add the file F
     */
    change replica R of file F's status value to DATA
    attempt to reconcile file F from another data-storing site
        (unless we are performing a remote replica addition)

    /*
     * If reconciliation fails, there will be no data at replica R
     * for file F until reconciliation completes correctly
     */
end
```

Since all volume replicas locally store the volume root, the recursion is guaranteed to terminate.

3.4 Replica selection

The purpose of replica selection is to locate a data-storing replica for file access. Should the file be replicated locally, the problem is trivially solved. However, if not stored locally, and the machine is connected to other volume replicas via some network, then it becomes possible to transparently locate a remote replica of the file and provide access to it as if the file were local. This is the task of replica selection. Note that replica selection as it is described here only operates within a specific volume. Access to remote volume replicas involves a different process, since the lists of volume replicas are maintained in separate data structures, and is described in [GHM+90].

Replica selection assumes that in the usual case, the user (or replica selection client) desires access to a data-storing replica. Occasionally a specific replica might be requested for a specific purpose. In these cases, replica selection is bypassed, and the requested replica is accessed (if it exists). Only a small number of operations are allowed on replicas which do not store data — these mostly fall under the category of attribute-queries, such as querying the status vector of a replica in an ATTRIBUTES-ONLY state. However, if a specific replica is not requested, the system guarantees to provide a file replica that stores data, or return a special error code if one cannot be located.

There are two main components of replica selection. The first is determining whether a given file replica stores data. The second is the actual replica selection algorithm. Both of these will be examined further below.

3.4.1 Determination of a file replica's state

The status vector of Section 3.2.1 was designed specifically for dynamic determinations of which file replicas store data. Recall that given a status vector sv for replica r, and the local index into the status vector i, the local status value is $sv_r[i].status$. Therefore, when accessing a file replica, replica selection can determine the replica's status by querying the appropriate status value. If and only if this value is the DATA status value has a data-storing replica been identified.

3.4.2 Replica selection algorithm

Without loss of generality, we can assume that all replica selection clients request access to files identified by the form `pathname/filename`. Given this assumption, the replica selection algorithm has three cases to consider:

1. The file `filename` is stored locally (and therefore `pathname` is also stored locally by the full backstoring invariant of Section 3.2.2).
2. The parent directory `pathname` is stored locally, but the file `filename` is not.
3. Some component of `pathname` is not stored locally.

Recall that replica selection as it is described here only operates within the volume boundary, so it is assumed that `pathname` is relative to the volume root (or the current working directory), and does not cross between volumes. Absolute pathnames can have all preceding portions of the pathname which name the actual volume removed for the purposes of replica selection within the volume.

We will now consider each case in turn.

3.4.2.1 The file `filename` is stored locally

When `filename` is stored locally, `pathname` is stored locally as well, given the full backstoring invariant. The lookup on `pathname` completes, returning the local directory object. (The lookup on `pathname` proceeds in the same manner being described). Given this object, the lookup process uses it to obtain a local object for `filename`. `Filename` is tested to determine if the local replica stores data, as described above. In this case it passes the test, and the file is presented to the replica selection client.

The testing process is essentially zero overhead, and is certainly masked by disk access time [2]. Performance data which verifies this conjecture can be found in Chapter 6.

3.4.2.2 The parent directory `pathname` is stored locally

As in the above case, the lookup on `pathname` provides the local directory object, and uses it to obtain the local file object for `filename`. However, in this case the local file object fails the data-storing test, since it is an ATTRIBUTES-ONLY replica. (The directory-entry invariant of Section 3.2.2 guarantees that the ATTRIBUTES-ONLY replica will exist.)

The status vector is extracted from the file object. Using it, the set of replicas believed to store data for `filename` are identified. This set is not reliable, because the status vector is optimistically replicated, and therefore could be out of date. Thus, the members of the set are only potential candidates that are believed to store data for `filename`. Each remote replica candidate is tried until an accessible, data-storing replica is located. Candidate testing can be performed using an intelligent selection policy knowledgeable about network distance and delay time; currently, it is a simple, linear search.

The status vector is treated like a cache. If in fact it is completely incorrect, and none of the candidates actually store data, the local status vector is replaced with another replica's. With the hopefully more accurate status vector, candidate-testing is repeated.

Locating another status vector for `filename` is accomplished using the status vector for the parent directory `pathname`. The set S of data-storing replicas of the parent directory is constructed from the parent directory's status vector. Since all members of S must store at least ATTRIBUTES-ONLY replicas of `filename` (by the directory-entry invariant), we can replace the status vector for `filename` with that found at any replica in S, and repeat candidate-testing.

---

[2] The overhead incurred is a search of the status vector to find the element corresponding to the local replica, and a test on the status value. The test is an integer compare of four bits. The search is currently only a simple linear search, which takes $O(N)$ time on a vector of length N.

Should all members of the set S be exhausted and a data-storing replica of `filename` has still not located (a rare occurrence), we are forced to fall back on a reliable but slow polling scheme. Each volume replica is polled for the existence of the file `pathname/filename` until a replica is found. When no data-storing replica can be located, a special error code is returned to the client.

3.4.2.3 The parent directory `pathname` is not stored locally

The third possibility is that `pathname` is not stored locally. Fortunately, this case requires no special mechanism, as it falls out of the second case above. Since all volume replicas store the root of the volume, all files in a volume must have an initial prefix of the pathname which is stored locally (relative to the volume root). For instance, `pathname` can always be decomposed into two pieces, `first` and `second`, where `pathname` = `first/second` and `first` is stored locally. The lookup on `first` is handled locally under the first case above, and the lookup on `second/filename` is handled under case two above. Once replica selection has obtained a data-storing file object, regardless of whether the object is actually stored locally or remotely, all access to it will transparently be directed to the remote storage site. Thus, replica selection needs no special mechanism in this case. The transparency is due to the vnode interface [Kle86] and the Ficus layered architecture [HP94].

3.4.2.4 Algorithm

The following is the high level implementation of the replica selection algorithm.

```
/*
 * Procedure ReplicaSelect:
 *  Return a data-storing replica of the file F in directory D
 */
Procedure ReplicaSelect(directory object D, file component F)
begin
    let R = the currently-accessing-replica of object D

    /*
     * Attempt "local" replica access, where local is relative
     * to the replica of object D.
     */
```

```
(valid, object-F) = lookup(D, F, replica R)
if (valid) then
    return object-F;

/*
 * "local" (replica R) access failed, but at least we have
 * a status vector.
 * Find a remote replica using the status vector.
 */
for (a = each DATA status replica in status vector except R)
do
    (valid, object-F) = lookup(D, F, replica a)
    if (valid)
        return object-F

/*
 * Status vector failed, must be out of date
 * Obtain new status vector from other replicas of
 * the directory D and repeat.
 */
while (more remote status vectors to obtain) then
    copy remote status vector to local
    for (a = each DATA status replica in status vector)
    do
        (valid, object-F) = lookup(D, F, replica a)
        if (valid)
            return object-F

/*
 * Obtaining new status vectors failed.
 * We must poll all volume replicas (sigh).
 */
for (n = all volume replicas)
do
    (valid, object-F) = poll(volume replica n, F)
    if (valid)
        return object-F

/*
```

```
     * The file must be at an inaccessible volume replica.
     * We cannot access it at this time.
     */
    return "file inaccessible"
end

/*
 * Procedure Lookup:
 * Access replica R of file F in directory D
 * Return the file object and a flag indicating its storage status
 */
Procedure Lookup(directory object D, file component F, replica R)
begin
    (error, object-f) = access(D, F)
    if (error) then
        /*
         * This case should never occur.  The directory-entry
         * invariant states that if a replica stores a directory
         * it must store replicas of all directory entries
         * in at least the ATTRIBUTES state
         */
        panic("Invalid state --- directory-entry invariant failed")

    let SV = status vector of object-f
    test_result = test-status-vector(SV, replica R)
    if (test_result is the DATA status value) then
        return (valid, object-f)
    else
        return (not valid, object-f)
end
```

CHAPTER 4

Reconciliation

Recall from the Ficus overview in Chapter 1 that *reconciliation* is the name of the process which maintains data consistency, and the *reconciliation topology* is the communication pattern used by reconciliation. The underlying reconciliation algorithms are discussed in [Guy91]. While these algorithms are topology independent, meaning that no specific reconciliation topology is required for correctness, different topologies yield different results in terms of performance and *message complexity*, or the total number of inter-replica messages exchanged globally. For example, a quadratic message complexity results from an all-pairs reconciliation topology, but a ring (using gossip-transferral of information) reduces the message complexity to a linear cost. An interesting problem is to avoid the quadratic cost when suitable inter-site communication is available, but still gracefully handle degraded communication.

The Ficus solution is an *adaptive ring* [PGPH90], which is a ring capable of dynamic reconfiguration during network failures and adaptation to volume replica additions and deletions. However, the adaptive ring assumes that all replicas are capable of communicating the sum total of global knowledge, and selective replication breaks this assumption, as we will see. Therefore, a new topology needs to be developed for selective replication: one that operates correctly and shares the same robustness and linear message complexity as the adaptive ring. The solution developed is a *multi-ring topology*.

This chapter will discuss the adaptive ring used by Ficus and illustrate both its robustness and its implicit assumptions and pitfalls for selective replication. The multi-ring topology will be introduced, as well as a discussion of its implementation and performance. The chapter will conclude with some remarks about optimizations to Guy's original algorithms.

4.1 Adaptive ring topology

We will identify the set of volume replicas as the set V. In the adaptive ring topology, participants reconcile with the next member in V as defined by ordering the replica identifiers. This ordering produces a ring topology. However, modifications are made to make the ring adaptive in two ways:

1. Due to the nature of the ordering process, the ring is dynamically reconfigured when the set V increases or decreases in size.

2. During periods of network failures, a given replica $v \in V$ reconciles not with the next replica $v + 1 \bmod V$ but with the next *accessible* replica $v + e \bmod V$, where $e \in 1, ..., |V - 1|$.

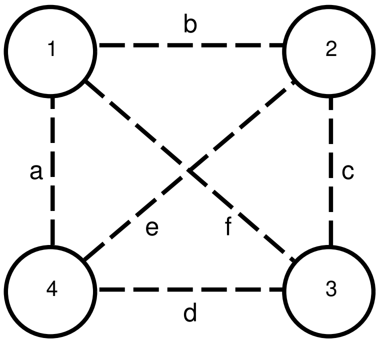

Figure 4.1: An interconnected network of sites. Circles represent individual sites, indicated by number. Dotted lines indicate communication links, indicated by letter.

The adaptive ring requires only a linear message complexity of O($|V|$) messages to propagate a given update to all replicas. As illustrated by point 2 above, it adapts itself to changing network topologies, and is therefore very robust. Since replicas gossip among themselves about the activities of third-party replicas, the ring topology does not require point-to-point links connecting all replicas to each other. Each replica in the ring eventually learns all information by communicating with its neighbor, who in turn learned from its neighbor. Given the gossip-based robustness, a fully-connected network of replicas can tolerate any two link failures, and some combinations of three link failures, and still transfer information successfully between all replicas. For instance, consider the network topology of Figure 4.1. Removing any two links, and some combinations of three links (such as links a, d, and f), still leaves all nodes *gossip-connected*, meaning that any node n_i can learn about the state of node n_j via communication with node n_k. A direct corollary of the above robustness is that the adaptive ring allows sites such as laptops that rarely or never communicate directly to one another to still share data by relying on third-party replicas to gossip on their behalf.

Since it can tolerate network failures and provide sharing between non-communicating sites, the ring is a very desirable topology. Unfortunately, it also has an accompanying cost. The ring operates under the assumption that nodes can obtain all necessary information solely by communicating with one other node — its neighbor. Furthermore, it is irrelevant which "neighbor" is communicated with, because they all store the exact same set of information. Selective replication breaks these assumptions, because since files are only stored at a specific subset of volume replicas, only that subset can be knowledgeable

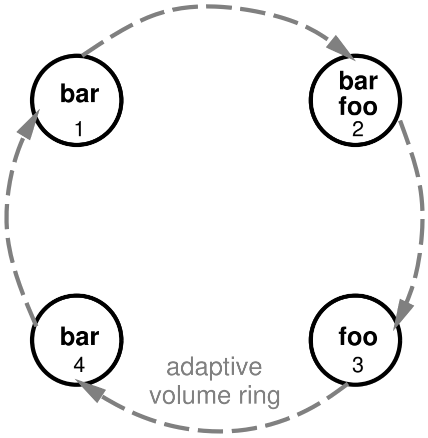

Figure 4.2: An example of two selectively replicated files, `foo` and `bar`. Circles represent volume replicas, indicated by replica number. Gray dotted lines indicate the communication pattern in the standard volume ring topology. Using this topology, reconciliation cannot make progress without forcing the non-data-storing replicas to store file data.

about the state of the file. Forcing other volume replicas to maintain data solely for gossip purposes negates two of the key advantages of selective replication — saving disk space and communication cost.

A simple example, shown in Figure 4.2, illustrates the problem. Using the standard volume ring, replica 2 learns about updates to the file `foo` generated at replica 3 when it reconciles with replica 3, but replica 3 will never learn of updates generated at replica 2. Replica 3 normally reconciles with replica 4, the next site in the ring, and replica 4 does not store data for the file `foo`. Only if both replicas 4 and 1 were inaccessible would replica 3 correctly reconcile with replica 2 using the volume ring. The situation exists similarly for the file `bar` as well. Thus, it is no longer the case that a given volume replica can communicate with an arbitrary neighbor for reconciliation purposes.

The new topology developed to retain the ring's robustness and operate correctly given selective replication is the *multi-ring* topology.

4.2 Multi-ring topology

The central problem with the volume ring is that the reconciliation topology operates at a different granularity from that of the replication service. The multi-

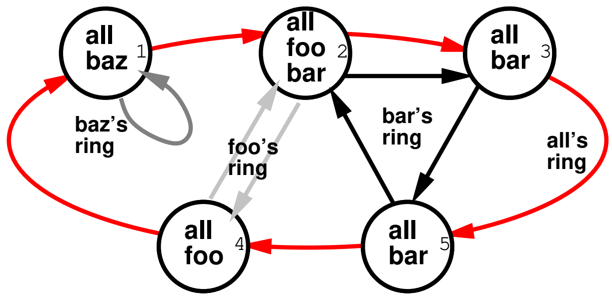

Figure 4.3: An example of the multi-ring topology. Four files are selectively replicated. Files are indicated by name (`all`, `foo`, `bar`, `baz`), and each file's ring is labeled by name and indicated by arrows. Circles represent the volume replicas. Together all four rings comprise the multi-ring topology for the given volume and replication pattern.

ring topology rectifies the problem by associating an adaptive ring with each individual file. By definition, this ring must be a subset of the volume replica ring, since the file exists within the volume. The file ring is also adaptive: it responds to replica additions and deletions, and adapts to changing network topology as the volume ring did before.

Reconciliation for each file proceeds along the file's own per-file ring. Since a volume contains many files, reconciliation of a volume consists of communicating along multiple rings, exactly the same as reconciliation in Ficus operated on multiple volumes by utilizing several volume rings. Figure 4.3 illustrates a multi-ring topology example. The file `all` is replicated at all volume replicas, whereas the files `foo`, `bar`, and `baz` are selectively replicated within the network. All together there are four separate rings for the four files in the volume.

4.2.1 Multi-ring topology performance

In analyzing the performance properties of the multi-ring topology, recall the performance design discussion of Chapter 2, Section 2.3. The multi-ring topology provides that volume replicas only participate in reconciliation for their particular locally-stored files. Therefore, the time to reconcile a given volume replica is

$$O(|\text{locally-stored files}| * \text{Time(one file reconciliation)})$$

which satisfies the constraints we set forth, since the time to reconcile one file is unchanged by the topology, and remains an $O(1)$ operation as in Ficus.

Additionally, the time required for new information to be distributed to all file replicas is

O(|file replicas|)

which, we believe to be adequate. An O(1) time to distribute updates to all file replicas would of course be optimal, and is achieved (as connectivity permits) with the *update-propagation* service previously described.

Nevertheless, the multi-ring topology as described exhibits a serious performance problem, namely the initiation of separate communications for each file, resulting in O(|locally-stored-files|) communication actions. When files are stored at similar subsets of replicas, multiple communications are unnecessary. For instance, at one extreme is full-replication — users replicate all files at all replicas. In such an environment, reconciliation need initiate only one (albeit large) communication action to reconcile all files. Even in a selectively-replicated environment, two replicated files will often be both stored at a similar set of volume replicas. Locality of reference suggests that if they are both stored at the same replica, they are somehow logically-related, so a given volume replica that stores one is likely to store the other as well. In light of such observations, the multiple communication problem must be addressed in the implementation of the multi-ring topology so that performance becomes reasonable. The solution developed takes advantage of the locality of reference by reconciling files in *groups*.

4.2.2 Multi-ring implementation

A *group* is a property of two communicating volume replicas. It is defined to be the set of files that is mutually stored at both replicas, and thus takes advantage of duplication or overlap in individual file rings. An example of the group concept is illustrated in Figure 4.4. The figure uses the same network topology as the multi-ring example in Figure 4.3, and describes the various groups. The group definition is symmetric, so the group between replicas X and Y, indicated by $group_{X,Y}$, is the same as the group between replicas Y and X. Additionally, $group_{X,X}$ is defined to be exactly the files locally stored at replica X.

We implement the multi-ring topology so that reconciliation becomes driven by the group concept instead of the individual file ring. When one volume replica reconciles with another, the two communicate information about all files in the group which they share. Group communication allows all m files whose per-file rings intersect to be reconciled in one communication action, rather than m.

To guarantee algorithm correctness, reconciliation by group must behave ex-

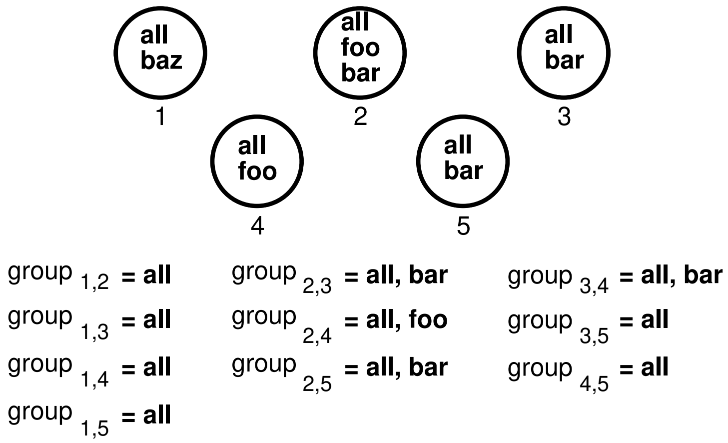

Figure 4.4: An example of the group definition. Four files, indicated by name, are replicated in the network. Volume replicas are indicated by circles and their replica identifier beneath. $Group_{X,X}$ is trivial and not shown. $Group_{X,Y}$ is equivalent to $Group_{Y,X}$.

actly like the multi-ring topology predicts. Specifically, the multi-ring topology guarantees that information correctly propagates around the ring only if each file reconciles with the next available file replica in the ring. Treating the reconciliation topography as a graph, we must guarantee that the graph has no sinks or sources. If the graph has a sink, updates propagate into, but not out of, the sink; the reverse is true for sources in the graph. In both cases, information is no longer guaranteed to be distributed properly to all participants. For instance, updates generated at the sink never reach other replicas.

Therefore, while some files may be members of multiple groups with multiple volume replicas, we must ensure that the topology generated by group reconciliation does not create sinks or sources. Specifically, correctness dictates that each file must be reconciled with the "first" (first accessible) replica in that file's ring. The group associated with the first replica in the file's ring is referred to as the *first-group* for that file. Each file has its own first-group associated with it, and reconciliation must guarantee that each file is reconciled with its first-group. A simple first-group example is illustrated in Figure 4.5. The example is from volume replica 1's point of view. The file `foo` must be reconciled as a member of the group shared between replica 1 and 2, called $\text{group}_{1,2}$, even though it is also a member of $\text{group}_{1,3}$. $\text{Group}_{1,2}$ is `foo`'s first-group (from volume replica 1's point of view) — reconciliation of `foo` *only* as a member of $\text{group}_{1,3}$ would turn

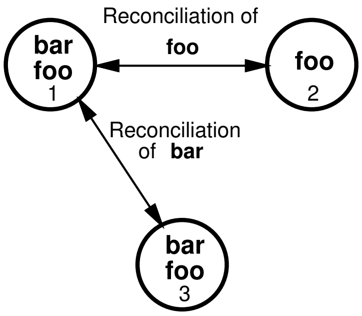

Figure 4.5: A simple example of the first-group definition. Volume replicas are indicated by circles, and their replica identifiers by number within. Files (`foo` and `bar`) are indicated at the replicas which store replicas. The arrows indicate the multi-ring topology from volume replica 1's perspective.

volume replica 2 into a source in `foo`'s reconciliation topography graph.

Once volume replica 1 reconciles `foo` as a member of $\text{group}_{1,2}$, re-reconciliation as a member of $\text{group}_{1,3}$ is not strictly necessary from a correctness standpoint (though this will change when we consider timebased optimizations to reconciliation in Section 4.3). Of course, re-reconciliation, while not necessary, does not violate correctness either.

4.2.3 Multi-ring topology details

With the high-level description of the multi-ring topology, we must now describe the low-level details of the implementation. Prior to reconciliation, each replica R computes its set L of locally-stored files by traversing its local volume replica. Using each file's status vector, R dynamically computes the set G of groups and marks each file's first-group. Groups which are not first-groups cannot be discarded, however, because the first-group is a dynamic property of the current network connectivity, and therefore second-groups could get promoted to first-group status when replicas are found to be inaccessible.

As previously mentioned, the status vector is an optimistically replicated data structure, and can therefore contain out of date information, resulting in incorrect group formation. However, reconciliation is robust to these errors and automatically corrects them, as we will describe.

Each replica independently initiates reconciliation at modifiable time intervals. The *initiator* replica constructs the set G of groups as described above. For each first-group $g_1 \in G$, the appropriate remote replica is tested for accessibility. If it is not accessible, g_1 is discarded, and the first-group property is recomputed for all files whose first-group was g_1.

The accessible replica contacted by the *initiator* is called the *responder*, as it responds to the initiator's requests for new information. Information flows unidirectionally from responder to initiator; the responder learns new information only when it independently becomes the initiator and initiates its own reconciliation action. The responder constructs its own view of the files believed to be in the group shared with the initiator. As already noted above, the set of files constructed by the responder, called the *responder-set* may not be identical to the set of files in the initiator's first-group, or the *initiator-set*. Differences between the two sets are caused by one of three situations:

- The responder-set contains files which the initiator-set does not, but should.

  These include both new files and remote replica additions, performed on behalf of the initiator but not physically at the initiator.

- The initiator-set contains files which the responder-set does not, but should.

  These are file replicas which the initiator has deleted, unknown to the responder.

- Either set S contains files which do not exist in the other set, and should be removed from S.

  These are files which have completed either of the two create/delete ambiguity-resolution algorithms, and have had the record of the file removed at one of the two replicas. The two algorithms are the dropping-notification and garbage collection algorithms.

The discrepancies are detected by the reconciliation process, and the appropriate action taken in each case. Additionally, there can be changes in a file's status vector not reflected by the above discrepancies: changes to status vector elements corresponding to replicas other than initiator or responder. These changes are resolved using a version-vector approach as described in Chapter 3, Section 3.2.1.

Finally, reconciliation performs the "standard" reconciliation actions described in [GHM+90], such as conflict detection, update propagation, and create/delete ambiguity resolution.

4.3 Reconciliation optimizations

We would like to consider what optimizations to the above algorithm can be made to reduce the total reconciliation time from initiation to completion. One chief optimization which can be made to the reconciliation algorithm is the use of timestamps and *timebased reconciliation*[1]. The optimization existed in Ficus before selective replication, and is not dependent upon selective replication; however, selective replication and timebased reconciliation interact in an unusual manner, forcing an algorithmic modification to the multi-ring topology.

4.3.1 Timebased optimization

The basic idea behind timebased reconciliation is that due to locality, typically only a small subset of the files locally-stored at any given volume replica will have been modified in between reconciliation times. Therefore, a method that could identify just the set of modified files (as opposed to the set of all files) would reduce reconciliation time by reducing the number of files needed to be analyzed.

The method implemented records timestamps. Stored at each volume replica S is a series of data pairs {timestamp T, volume replica R} which reflects the last time T that S successfully reconciled with R. To avoid clock synchronization problems, the time T is recorded from R's clock, rather than the clock at S. The next time S reconciles with R, only the files at R which have been modified since time T need to be analyzed: the remainder by definition are unchanged, and thus reconciliation would not take any action on them.

Each file in R's responder-set which has not changed since time T, is tagged with an "unchanged" marker to indicate that the file does not need to be reconciled. The unchanged marker is necessary to differentiate the case that R does not store the file from the case that the file has not been modified at R since time T.

4.3.1.1 Algorithmic difficulties with timebased reconciliation

Unfortunately, the timebased optimization requires a small algorithmic change to the multi-ring topology for correctness. Recall that previously correctness only required that a given file be reconciled with its first-group. With timebased reconciliation, in order to correctly update the timestamp value, the file must be re-reconciled as part of each group. Without the algorithmic change, updates

[1]The original idea and implementation of timebased reconciliation was by John Heidemann.

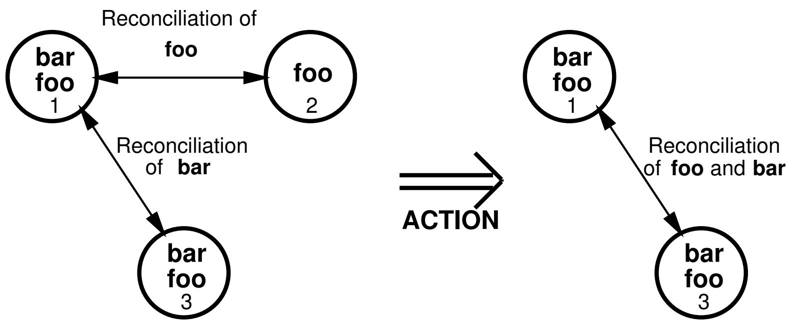

Figure 4.6: An example illustrating why the timebased optimization requires an algorithmic change. Volume replicas are indicated by circles, and their replica identifiers by number within. Files (`foo` and `bar`) are indicated at the replicas which store replicas. The arrows indicate the multi-ring topology from volume replica 1's perspective. The action indicated is a deletion of either volume replica 2 or 2's file replica of `foo`, which causes problems for the timebased optimization.

could be accidently lost, as the following example illustrates.

We will prove that an algorithmic modification is required by means of a proof by contradiction. We will assume that we are using the timebased optimization as described, but that files are only reconciled with their first-group, and illustrate how the lost-update effect occurs.

Figure 4.6 illustrates the multi-ring topology under these assumptions from volume replica 1's reference point. As indicated by the arrows in the figure, replica 1 reconciles the file `foo` with replica 2, as $group_{1,2}$ is `foo`'s first-group, and updates the timestamp $\{T_a, 2\}$. T_a is recorded from replica 2's clock after the reconciliation of `foo` completes. Replica 1 then reconciles the file `bar` with replica 3, and under our assumptions does not re-reconcile `foo` even if replica 3 of `foo` has the most recent data. The time T_b is recorded from replica 3's clock, and the timestamp $\{T_b, 3\}$ is recorded, but note that the most recent modification time of `foo`, $(T_{mtime(foo,3)})$, is less than T_b.

Consider what happens if either volume replica 2 or its file replica of `foo` is deleted. The only remaining replicas of `foo` are at replicas 1 and 3. Replica 1 already stores the timestamp $\{T_b, 3\}$ and $T_b > T_{mtime(foo,3)}$. The next time replica 1 reconciles with replica 3, the file `foo` will be marked as unchanged, even though it actually contains new updates. Barring any future updates at replica 3, replica 1 of the file `foo` will remain out of date indefinitely, causing the timebased lost-update effect.

4.3.1.2 Algorithmic modifications for timebased reconciliation

It is clear from the above discussion that some modification to either timebased reconciliation or the multi-ring topology is required for the two to correctly interoperate. An inspection of the situation reveals that the underlying problem occurs when the timestamp for a site R is updated and not all of R's data is processed. A modification to solve this would be to change the granularity of the timestamp from volume to file, but this solution does not scale, since one timestamp would need to be stored per file per replica, which is clearly impractical for large numbers of files or replicas.

A closer analysis of the situation illustrates that the lost-update effect arises only when the above site R stores new information, which later becomes lost by the nature of the timestamps. By modifying the multi-ring topology (when using the timebased optimization) to force a "re-reconciliation" of each file as a member of each group, the lost-update effect can be avoided, at a relatively cheap cost. The re-reconciliation process is inexpensive for two reasons:

1. If in fact the re-reconciliation causes new data to propagate or the modification of file attributes, these actions would have had to eventually occur at some future reconciliation time. No "extra" work is being performed in this sense. In addition, performing the re-reconciliation in this case has the advantage of maintaining stronger consistency.

2. The main cost of reconciliation occurs when actions must be performed or information must be transported. If in fact the re-reconciliation does not generate changes to the file system, very little time has been wasted. Additionally, the decision process occurs entirely in user space, so the kernel did not have to perform any additional work.

4.3.1.3 Benefits of timebased reconciliation

An analysis of timebased reconciliation shows that, with selective replication, only a small improvement is made in reducing total reconciliation time. All file records must still be transported from responder to initiator, so no communication time is saved. The responder saves CPU time because records it marks as unchanged are left as incomplete objects, since their internal data fields will not be queried. The initiator saves CPU time because the records marked as unchanged bypass the tests to determine if reconciliation needs to take action. However, the tests already occur entirely in user space, so the savings generated at the initiator are not large.

CHAPTER 5

Garbage Collection

Garbage collection is the process responsible for "cleaning-up" the file system after files are removed. The file's resources, namely the disk blocks occupied by the data, must be returned to the general system pool, so that they can be reallocated. Resource deallocation must be done in centralized as well as distributed systems — this is not a unique problem in this aspect — but distributed issues make the problem significantly more difficult. Files have pointers, or *links*, which name them and provide access, and garbage collection must occur strictly after all links to the file have been removed. Otherwise, user data could be lost as the the file's disk blocks gets prematurely re-allocated to another client. While a determination in a centralized system that all links have been removed is trivially solved with a simple counter, the determination is made more difficult in distributed, replicated services by dynamic naming. Dynamic naming allows users to create new links, implying that the removal of the last local link at replica R is not an indication of the state of all links. Other links could be active at other replicas, which R will learn about when it initiates reconciliation. Data could be lost if R has the most recent version and decides to garbage collect the file before learning of the new link.

On the other hand, waiting longer than necessary to garbage collect causes problems for the user. The disk space remains allocated to the files waiting to be garbage collected, and is therefore unusable by other clients, effectively reducing the size of the user's disk. Garbage collection must attempt to free the disk space as soon as possible, but not before all links globally have been removed, making it a difficult problem.

This chapter will discuss how garbage collection works in Ficus, and why its solution is not applicable to selective replication. It will then present a solution for selective replication, and discuss its performance and implementation, concluding with an informal proof of correctness.

5.1 Garbage collection in Ficus

Guy, in [GPP93], discusses how garbage collection of files operates in Ficus. However, his results assume full-replication: that is, Guy assumes that each file is replicated at each volume replica. For this reason, Guy's algorithm does not directly apply to selective replication.

Instead, let us consider the analogous operation to a file removal in selective replication — a volume removal in Ficus. These are analogous operations because they both are removals at the granularity of replication, and therefore the objects being removed can exhibit arbitrary replication patterns. Ficus handles the removal of volumes by forcing the replication factor of the volume to zero, using the volume-deletion utility. When the replication factor of the volume is zero, all replicas have been removed and garbage collected. Note that such a process requires either all volumes to be in the same partition at destruction time or user-intervention to destroy replicas in other partitions. In essence, Ficus does not really solve the problem.

The Ficus volume removal strategy is not well-suited to environments where volumes have hundreds of replicas spanning multiple network partitions, but it works adequately in limited-replica environments for two key reasons. First, volumes are heavy-weight objects by definition, and as such their removal is essentially a "global" decision. Due to the severity of the operation, volume removals are often preceded by out-of-band communication. Second, volumes are large objects, and therefore are not frequently removed, allowing for simple but inefficient procedures when removals do occur.

However, the Ficus volume removal solution does not apply well to either file removals or selective replication. First, files are light-weight objects, and their removal often occurs without a "global" thought-process and out-of-band communication. Second, files are created and destroyed much more frequently than volumes. High frequency implies that entirely automatic processes (involving no user intervention) are required when garbage collecting and that the execution time is an important issue. Finally, dynamic naming occurs much more frequently with files than volumes. A garbage collection process that alters replication factors must have additional functionality and complexity to restore the original replication factor in cases where it terminates prematurely due to new link discovery, because the garbage collection process should be invisible to the user.

Therefore, the Ficus solution does not apply to selective replication, and we must design our own. Additionally, due to the analogy between file removals in selective replication and volume removals in Ficus, such a solution would be directly applicable to the Ficus problem as well.

5.2 Garbage collection for selective replication

As previously mentioned, Ficus uses algorithms developed by Guy in [GPP93] to garbage collect files, but these algorithms assume the files are fully-replicated. We will expand the algorithms to remove the assumption.

To briefly review Guy's algorithms, Guy developed a fully-distributed, two-phase algorithm which executes independently at all volumes replicas. When a file has no links at replica R, replica R initiates Guy's algorithm. The algorithm terminates when either a new link is discovered (abnormal termination meaning garbage collection is halted), or all replicas agree that no links exist globally (normal termination meaning garbage collection can proceed).

Guy's algorithm assumes full-replication, implying that the garbage collection participants are well-known prior to initiation. In order to remove this assumption, we construct additional mechanism to identify and inform all replicas which need to participate in the garbage collection process, and then execute Guy's algorithm among those participants. These processes can be identified as:

1. Identifying the subset of volume replicas that participate in garbage collection

2. Informing all sites in the above set that they are in fact members

The new mechanism will handle file replica additions and deletions like Guy does at the volume granularity, but the problem is significantly more difficult. The volume provides a simple inheritance mechanism, which makes determining set membership for objects within the volume much easier. Here, the object at the granularity of replication is in fact the object being garbage collected.

The two key new processes, identifying and informing the participants, will now be discussed in detail.

5.2.1 Identifying the participants

Identifying the garbage collection participants means deciding the subset of sites that need to participate in the algorithm. At file removal time, we can trivially identify the set of sites that were supposed to store it: that is, those volume replicas that would have eventually stored the file had it not been removed. This information is maintained in the status vector. Although the local status vector may not reflect recent replica changes, such as additions or deletions, assume for the moment that we can detect such actions in parallel (as will be shown later). In this way, identifying the set of storage sites that were supposed to store the file is straight-forward. We shall call this set the *eventual storage sites* $= E$.

However, at file removal time, not all members of E necessarily know that they are supposed to store the file, and identifying which members of E actually store the file is more difficult. We shall identify this smaller set as the *actual storage sites* $= A \subseteq E$. Since replicas gossip about new information, it is impossible for a site S to construct the set A using only local information. An individual replica does not know which other replicas have learned of the file by gossiping with third parties. Thus, other replicas must be contacted to construct the set A, and synchronization barriers must be used to ensure that once a member $e \in E$ is deemed not to be in A, e does not later learn of the file and become a member of A.

We would like to identify the smallest set possible which can be easily used for garbage collection — the smaller the set, the less time to contact all members and complete garbage collection. Using the set A to identify the participants is only advantageous if $|A| \ll |E|$ and the time and work necessary to construct A is less than the extra time to garbage collect using the set E. If we assume that network partitions are rare and temporary events, then reconciliation will be able to converge all replicas to a common state in a reasonable amount of time: it follows that in general $|A| \simeq |E|$. On the other hand, if network partitions occur rapidly and often, or if file replication factors change rapidly, then intuition suggests that often $|A| \ll |E|$.

However, one must also balance the benefit of using the set A with the cost of computing it, which is in general not cheap. Computing A involves either querying all sites in E, which nullifies much of the advantage in using A over E, or maintaining gossip-lists to determine the flow of information regarding the file, and therefore which replicas actually store it. Both methods involve some sort of distributed algorithm to contact remote replicas and gather information regarding the set A as well as to erect synchronization barriers to stop the information flow to replicas deemed not in A. Due to the complexity involved, we use the set E, the eventual storage sites, as the set of participants for garbage collection.

Replica additions and deletions are handled in parallel by allowing the set E to fluctuate dynamically. As Guy's garbage collection algorithm contacts members of E, the set E could expand or contract, depending on each member's individual knowledge. Since Guy's algorithm consists of two phases, we can achieve consensus on the size of E before garbage collection completes. Replica additions and deletions are not allowed on files being garbage collected, and therefore the final set E can be constructed in one phase, and distributed to all participants in parallel during the second phase.

5.2.2 Informing the participants

Now that we have identified which sites should participate, it remains to inform them that they should participate. Informing all sites in the set A is trivial, because reconciliation will inform each of them of the removal of the link. The removal of the file's last local link triggers the initiation of garbage collection, and as data-storing replicas they will know to participate.

The difficult task is informing the members of the set $E - A$, called the *unknowledgeable storage sites* $= U$. Volume replicas in the set U do not yet store the file, because they are unknowledgeable about the file's existence. The problem arises in guaranteeing that they will learn of the file's existence now that the file has been marked as removed.

Recall that information propagates in a pull-only manner in the multi-ring topology as discussed in Chapter 4. Had the file not been removed, members of the set U would discover the new file by learning of the new directory entry pointing to it. Due to full backstoring, each member of U must store the parent directory — learning this fact involves applying the above argument, only one directory higher in the tree. Since the root of the volume is by definition always stored at all sites, the induction argument holds. Basically, full backstoring ensures that the creation of the per-file ring for the parent directory precedes that of the file itself, and therefore the existence of the file can be learned by discovering the new directory entry.

However, once the file has been marked deleted, the directory entry is marked as well. The two become logically divorced, and garbage collection proceeds independently and possibly at different rates on each. Thus, the discovery of the deleted directory entry is not enough for a member in the set U to learn that it should be a participant in garbage collection on the file previously-referred to by the link. A site $u \in U$ can only know to participate is if discovers it should store the file, which will occur only if u happens to reconcile directly with a replica $a \in A$. Unfortunately, there is no guarantee that such a reconciliation will ever occur. The members of the set U may never learn that they should be participants in garbage collection, and therefore garbage collection will never terminate. An example showing members of the set U never learning that they should be participants is illustrated in Figure 5.1. In the example, replica 1 believes that replica 2 stores the file `f1`, and cannot complete garbage collection until replica 2 participates. However, replica 2 neither stores the file nor will learn that it should, because it reconciles all of its local files with replica 3, the next replica in the multi-ring topology[1]. The `f1` ring does not exist, and therefore garbage collection on `f1` cannot complete.

The underlying problem is that garbage collection is initiated on a file `f` before `f`'s ring has been constructed, and the event that initiates garbage collection (the

[1]Note that had `f1` not been removed, then replica 3 would have learned of the new directory entry, and replica 2 would have learned of the file's presence in its reconciliation with 3.

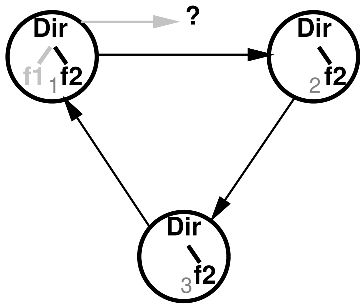

Figure 5.1: An example illustrating the problem with garbage collecting selectively-replicated files. Volume replicas are indicated by circles, and their replica identifiers by number within. Files in black reconcile along the black per-file ring. Files in gray (`f1`) are files waiting to be garbage collected, using the gray per-file ring not yet constructed. All volume replicas store all files, except that `f1` is to be stored at only replicas 1 and 2, and replica 2 has not yet learned of `f2`'s existence.

removal of the link) removes the only marker that provides for ring creation. Since information is passed in a pull-only manner, it cannot flow to everyone without the ring existing.

Modifying the information-transfer method to use a *push-pull* hybrid mechanism solves the problem. The idea is that in situations where the ring is broken and cannot be constructed by a standard pull of information, knowledgeable sites in the set A push the information to the sites in U. The push guarantees that the set U becomes empty, the ring correctly forms, and garbage collection can complete.

Utilizing a push for information transferal is only feasible when each site stores a list of knowledgeable and unknowledgeable sites. Pull-style mechanisms do not require this added complexity, and as such are generally simpler to implement. However, since the garbage collection algorithm itself requires verifying that each participant has performed a specific action, the list of sites already exists, and a push mechanism can be easily applied.

5.3 Garbage collection performance

Since only the data-storing replicas of a file participate in its garbage collection, and each replica participates a fixed number of times, O(r) steps are required to contact all replicas of an r-replica file. Guy's algorithm requires three phases for garbage collection to complete at all replicas (two phases for the algorithm itself and one phase for all replicas to learn of the completion) [GPP93]. Therefore, the total running time of garbage collection on our r-replica file is $3 * \mathrm{O}(r) = \mathrm{O}(r)$, which satisfies the design constraints of Chapter 2.

Garbage collection now consists of two parts — participant informing and Guy's algorithm. Fortunately, these two pieces execute in parallel: the push action can be performed concurrently with the execution of Guy's algorithm. For example, if during garbage collection, a site s communicates with a site $u \in U$ and determines that the ring is broken, the file is pushed to u. Only a file record for garbage collection purposes needs to be pushed, not the complete file data. Site u becomes a member of the set A, and the set U decreases in size by one. Additionally, the participant set is dynamically modified if, during garbage collection, new file replicas are discovered or existing replicas are deleted. In this way we handle file replica additions or deletions in parallel, and the modified garbage collection algorithm has the same performance properties as Guy's original algorithm.

5.4 Garbage collection implementation and correctness

The actual implementation of the new garbage collection algorithm is very much like Guy describes it in [GPP93], with the modifications to identify and inform the garbage collection participants executing in parallel. Each replica initiates the algorithm when the number of local links for a file drops to zero. If a new link is discovered, garbage collection is terminated. When all replicas complete Guy's two-phase algorithm, then each replica independently removes the file and de-allocates its resources.

Guy showed in [GPP93] that his garbage collection algorithm operates correctly. In order to show that the new algorithm is also correct, it remains to prove that the informing-phase eventually terminates, and that all sites that are supposed to participate do in fact participate. Proving this is equivalent to proving that U becomes ϕ, as when U is empty, $A = E$, and we simply execute Guy's original algorithm. The proof that U eventually becomes empty follows:

> If U is non-empty, then $\exists$ at least one site $u \in U$ such that u's "ring-neighbor" in the multi-ring topology is a site $a \in A$—that is, a reconciles by communicating with u. When a discovers the site u, it will push the file being garbage collected to u, thereby causing u to remove itself as a member of the set U and become a member of the

set A. After this action, the set U is decreased in size by one. This process repeats as long as U is non-empty. Upon termination, U is empty, and therefore by definition $A = E$.

CHAPTER 6

Performance Evaluation

In order to evaluate any new service, it needs to be demonstrated that the implementation does not degrade performance, except possibly under justifiable conditions. Regardless of the features, users will not use new services if their performance characteristics cannot be understood and justified. Therefore, we need to evaluate the performance of selective replication.

We would like to compare our service to that of a volume granularity replication service. Comparing against non-replication services, like standard UNIX, is not necessarily valid, because we are assuming that users require replication and will therefore use some sort of replication service. Nevertheless, a comparison against UNIX is presented as a base case for performance numbers.

We hypothesize that selective replication does not affect performance relative to a volume granularity service: that is, users will not be adversely affected by the selective replication implementation. As such, users will desire selective replication due to its new features and replication flexibility. This chapter will attempt to prove the hypothesis by discussing the experiments and benchmarks used for the evaluation as well as the results and conclusions which can be drawn from them.

6.1 The experiments

We wanted to study the relative performance of selective replication against a volume granularity replication service. For the volume granularity service, we used unaltered Ficus (without the selective replication functionality). We performed two experiments. First, we compared the cost of local data access in the two services. Second, we compared the cost of remote data access, accessing remote data in unaltered Ficus via a remotely-stored volume, and in the selective replication case via a remotely-stored file replica.

We also wanted to have an idea of the cost of selective replication — the overhead encountered when users save disk space and choose not to replicate specific files locally. Our belief is that the cost of the replication mechanism should also be insignificant when the file being accessed is remote. That is, the elapsed time to reference a remote replica may well exceed that of a local one due to inherent communication overhead, but the replication mechanism itself should not add further cost. To study the selective replication cost, we compared local data access times against remote data access times, using the selective replication service in both cases.

Furthermore, we compared selective replication performance against standard UNIX (SunOS 4.1.1) to serve as a base for performance numbers.

Finally, we wanted to understand the performance characteristics of the multi-ring topology compared to the simpler topology used in Ficus. We qualitatively analyzed both topologies to reach our conclusions.

6.2 The benchmarks

Six benchmarks were used in this evaluation. The first test is the modified Andrew Benchmark (`mab`) [HKM+88, Ous90], which is intended to model a normal mix of filing operations and hence be representative of performance in actual use. The second is a recursive copy tests within the replicated namespace, operating on the `/usr/include` hierarchy which on our system contains 4.2 Mb of data and 1191 files, of which 60 are directories. The third through fifth tests are combinations of "find" and "grep" tests: one find, one grep, one find and grep. `Find` and `findgrep` operate on the same `/usr/include` tree; `grep` operates on the `/usr/include/sys` subtree, which is 336Kb of data and 104 files. To the extent that one believes that the frequency of read operations greatly dominates writes in a typical filing environment, read performance is a critical measure of file system performance. The final test is an `ls` test, which operates on the same `/usr/include/sys` subtree. Many of the tests execute multiple repetitions, to extend the total running time, highlight potential performance problems, and achieve measurable results. The iteration counts for each benchmark are listed in Table 6.1.

All measurements are performed on Sun IPCs, each with a 207 Mb SCSI disk and standard Ethernet connection. All replicas and sites involved are part of the same Ethernet segment, and in all cases the location of the tests was performed in the same place in the namespace.

6.3 The results

We performed the experiments described in Section 6.1 and obtained the following results:

6.3.1 Local data access cost

We measured the local data access cost for Ficus and selective replication, and the results are shown in Figure 6.1. Illustrated are both the system and elapsed time for all seven benchmarks. In the Ficus case, data is stored in a local Ficus volume. In the selective replication case, data is stored in a selectively-replicated

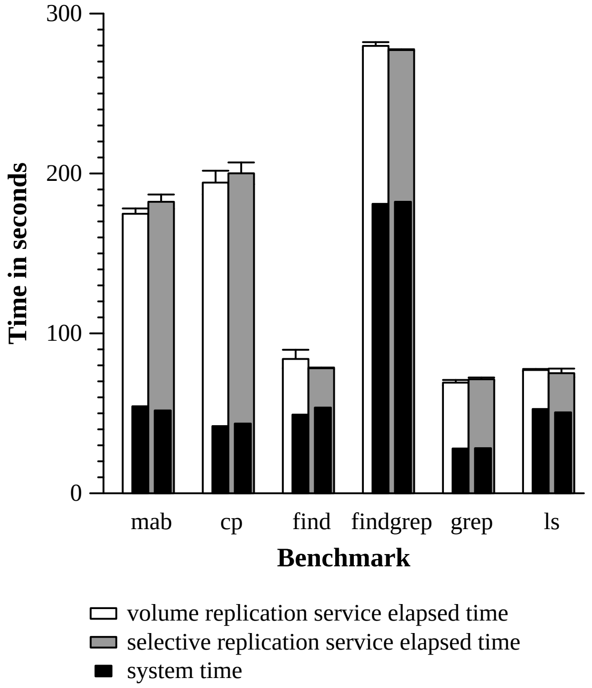

Figure 6.1: Local data access performance for the volume and selective replication services. Elapsed time is indicated by total height, system time by the black bar within. Each data point is the mean of five runs. 95% confidence intervals are shown, except where the pixel granularity obscures them.

Benchmark	Number of Iterations
`mab`	1
`cp`	1
`find`	5
`findgrep`	5
`grep`	100
`ls`	500

Table 6.1: The number of iterations for each benchmark described in Section 6.2.

volume with a locally-stored replica. As expected, there is basically no overhead imposed by the selective replication service on local data access. We can conclude from this that users accessing local data will not be visibly affected.

6.3.2 Remote data access cost

Figure 6.2 displays remote data access cost for Ficus and selective replication. System time and elapsed time are indicated for the seven benchmarks. In the Ficus case, remote data is accessed via a remote Ficus volume, which Ficus grafts into the local tree structure. In the selective replication case, the remote data is accessed via the selective replication mechanism: the data is accessed through a directory that is only stored on a remote machine. In both cases the data is stored on the same remote machine, which is on the same LAN as the local machine.

The results again show that, as expected, selective replication does not cost more than a volume granularity replication service, even when accessing remotely-stored data. Some of the benchmarks are even faster with the selective replication service, which can be explained by two factors. First, some of the key functionality in the critical path of operations was rewritten in the selective replication implementation, and this rewrite could have resulted in better and faster code. Second, the mechanism by which Ficus locates and accesses remotely-stored volumes is different from that which selective replication uses to locate and accessing remotely-stored files.

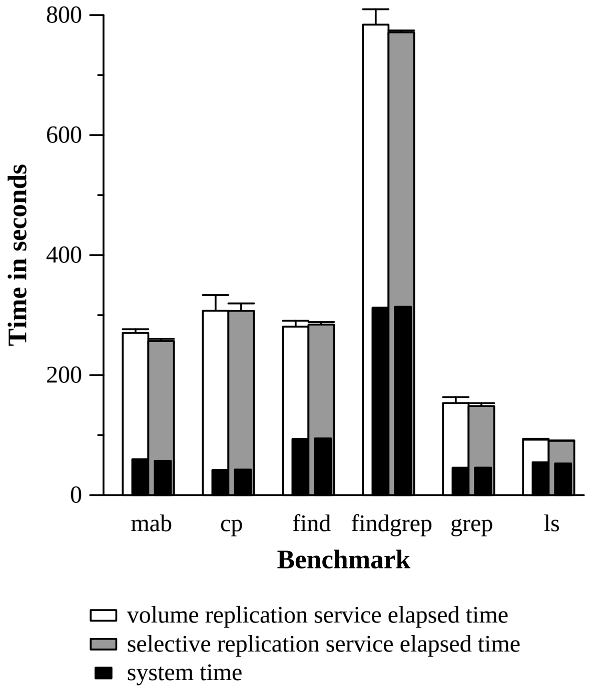

Figure 6.2: Remote data access performance for the volume and selective replication services. Elapsed time is indicated by total height, system time by the black bar within. Each data point is the mean of five runs. 95% confidence intervals are shown, except where the pixel granularity obscures them.

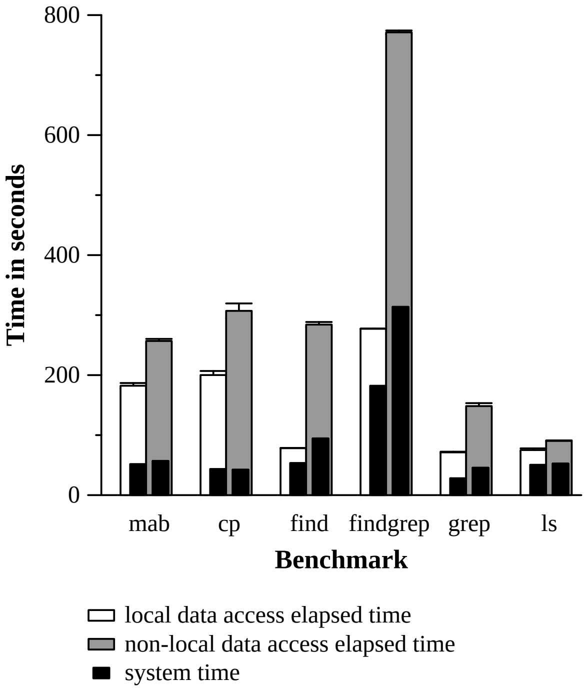

Figure 6.3: Local and non-local data access performance for the selective replication service. Elapsed time is indicated by total height, system time by the black bar within. Each data point is the mean of five runs. 95% confidence intervals are shown, except where the pixel granularity obscures them.

6.3.3 Selective replication cost

We wanted to measure the cost of selectively-replicating files only on remote machines, as we believe that the replication mechanism should not produce significant overhead above and beyond pure transport costs. Figure 6.3 illustrates these measurements. It displays the elapsed and system time for data access in the selective replication service, accessing both local and remotely-stored data. Unfortunately, it cannot show the savings in disk space when not replicating the files locally.

Three of the tests — `mab`, `cp`, and `ls` — show only a moderate overhead in elapsed time, and almost no overhead in system time. The majority of the overhead in these three tests is the waiting and communication time to transfer the remote data over the wire: a price readily accepted and paid by the user. In the case of the `ls` benchmark, we would expect identical results, because the directory-entry invariant guarantees that the names of the directory entries are all maintained locally. The slight performance degradation (6%) for this benchmark is because the top-level directory (`include`) is remotely-replicated, and therefore `include/sys` and the underlying directory entries are only stored remotely as well. If `include` were locally replicated and all data underneath were replicated remotely, we would expect identical results.

The `find`, `findgrep`, and `grep` tests show a much larger performance degradation in both system and elapsed time. All three of the tests are very dependent on `stat` time. (The grep benchmark is run on a large number of small files, so its total time becomes dominated by the `stat` time as well). `Stat` has historically performed poorly in Ficus due to a few early design problems, and selective replication has inherited these problems. Additionally, local `stat` time is insignificant compared to remote `stat` time, because at least one round trip per individual file is required in the remote case, and our current implementation performs two round trips, as the data cannot be cached due to cache-consistency issues. We are currently resolving the caching issues and analyzing methods to decrease the remote `stat` cost. Users operating in this intensive-`stat` mode would probably want to perform the computation at a replica which stores data for the files in question. However, for typical filing operations, such as those modeled by the `mab` test, the overhead for remote data access is within acceptable limits.

6.3.4 Base comparison against Unix

As previously stated, the comparison against standard Unix is slightly unfair because we are assuming that users require some form of replication (and optimistic

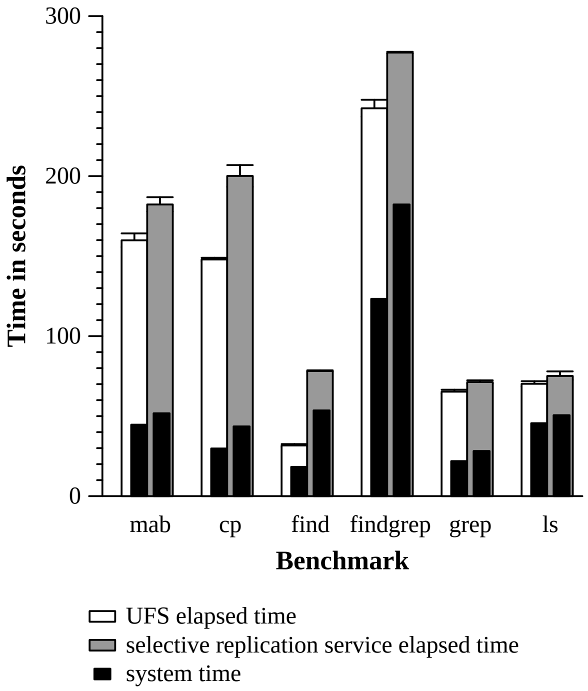

Figure 6.4: Local data access performance for UNIX and the selective replication service. Elapsed time is indicated by total height, system time by the black bar within. Each data point is the mean of five runs. 95% confidence intervals are shown, except where the pixel granularity obscures them.

replication additionally). Nevertheless, the performance comparison of selective replication and UNIX (SunOS 4.1.1) is presented in Figure 6.4. Many tests show only a slight overhead in elapsed time: `mab`, `findgrep`, `grep`, and `ls` are all under 14% overhead, with `ls` being the lowest at 6.9%. The `cp` performance is barely acceptable at 35% overhead, but the `find` test measured a 153% overhead in elapsed time. These results are unfortunately due to the Ficus implementation used by selective replication, described in [GHM+90]. The Ficus design and implementation accrues four I/Os beyond the normal UNIX overhead on each file open in a non-recently opened directory. The extra I/Os are due to the way Ficus currently interacts with the UFS layer and maintains its replication information. The overhead is neither necessary in Ficus, nor due to the selective replication service.

6.3.5 Reconciliation topology

We would like to understand what effect if any the multi-ring topology has on the performance of reconciliation relative to that of the simpler topology used in Ficus. A careful observation of the multi-ring implementation shows that, by using groups (discussed in Chapter 4), the multi-ring topology behaves exactly the same as the simpler ring topology when files are fully-replicated. In the fully-replicated case, the group shared between sites S and R contains all files in the volume, and therefore the multi-ring "collapses" into the simple adaptive ring. Therefore, when users choose not to selectively-replicate files, there will be zero performance impact on reconciliation.

We must also consider the performance impact on reconciliation when users do in fact selectively-replicate files, since this mode of operation is assumed to be the primary case. From the point of view of a single replica, two factors are pitted against each other. While reconciliation performance improves because selective replication implies there are less files to reconcile, performance degrades because the number of remote replicas to communicate with will often be greater than one.

If we assume that the savings generated by analyzing less files are larger than the penalties incurred by contacting more remote replicas, we can conclude that reconciliation performance will improve. In calculating the penalties, it is important to note that while there may be multiple communications, each one transfers only a subset of the files in the volume (which is the subset mutually stored at the two communicating parties), so each communication is shorter in real time than the one large communication in the non-selective replication case. However, some files (file meta-info, not file data) will be transferred multiple times as a result of the multi-ring topology.

Without real data regarding typical replication factors with selective replication, we cannot determine for certain the affect on reconciliation performance. We can assume that locality of reference will play a major role in determining replication factors — if two files are replicated at one site, they will often be replicated as a pair at other site — and this locality will help reduce the total number of communications needed to reconcile a volume replica. Nevertheless, we can only speculate until selective replication is truly used in practice (see Chapter 8).

6.4 Performance conclusions

We can conclude from the above experiments and observations that selective replication does not severely degrade performance above and beyond that of a volume-granularity replication service. Replicating at the file granularity therefore is not more expensive than at the volume granularity, contrary to what one might initially expect.

CHAPTER 7

Related Work

Previous chapters have examined our selective replication design and implementation, as well as key features of the Ficus system. Nevertheless, many other related services exist. Sun Microsystem's NFS is an important example of a service which provides access to non-local files. Deceit, LOCUS and Coda are all different examples of replication services which have made different design choices and assumptions about their target environments.

7.1 NFS

SUN Microsystem's NFS [SGK+85] allows remote access to non-local files using a special mount system call. The remote file system is "mounted" into a specific location in the local file system. However, NFS does not support replication of the data, only remote access to it, and operates strictly at the volume granularity. In addition, the volumes must be in well-known locations, and the system must be manually changed when the location changes.

7.2 Deceit

The Deceit file system [SBM90, Sie92], places all files into one "volume" and allows each individual file to be replicated independently with varying numbers of replicas. In this sense it provides selective replication. However, Deceit employs a conservative approach to replication, namely a writer-token mechanism, and as such is unable to provide the high availability offered by optimistic mechanisms. In addition, Deceit cannot tolerate long-term network partitions; instead, only a related failure known as a *virtual partition* [SBM90], which eventually corrects itself, can be tolerated. Furthermore, Deceit uses a vastly-simplified garbage collection algorithm, one not guaranteed to operate correctly in environments with common network partitions. These factors make the Deceit system unsuitable for the modes of use discussed in Chapter 1.

Nevertheless, the Deceit system has several nice features, among them simple user controls for specifying replication factors. Users indicate the minimal number of replicas desired, and the system guarantees at least that replication factor. The system is free to change a replica's physical location and modify the number of replicas (observing the minimum) at any time. Replicas are free to be moved by the system. However, users are not able to request that replicas reside at specific hosts, which is a significant disadvantage in our hypothesized environments, where specific replicas simply must exist on specific laptops.

Our selective replication implementation solved the local availability problem by enforcing the system invariant of full backstoring (Chapter 3). Deceit solves it in a different way. Deceit implements the directory structure using an in-memory binary tree of all hard links, backed up to non-volatile storage. Similar to the previously-discussed "prefix-pointer" mechanism, Deceit's solution has the advantage that it saves the system from locally storing the intermediate directories, but has the disadvantage of added complexity, both to maintain the list and to garbage collect from it. Furthermore, the in-memory tree could potentially absorb much of main memory.

7.3 LOCUS

The LOCUS operating system [WPE+83, PWC+81] essentially provides the volume notion and allows selective replication within the volume. In LOCUS, replication is provided at the volume granularity. The file system is separated into "logical filegroups", analogous to the volume, and replication is made possible by allowing multiple physical "containers" for each filegroup, one container existing for each replica. Files belonging to a specific filegroup X may be stored at any subset of the sites that store physical containers for X. In this way, LOCUS provides selective replication.

However, the approach LOCUS takes toward replication is not strictly optimistic, and additionally has serious problems with network partitions. Within each partition, strict synchronization among the file replicas is maintained, so that conflicts can never occur inside a partition. A token-based approach is used, and processes must contact the current synchronization site, or CSS, to access (read and write) a file in a particular filegroup. The CSS ensures that all accesses are always directed to a file replica with the latest version (within the partition) — the CSS does not actually need to store file data, but must store enough information to essentially act like the primary site. As in any primary-site replication scheme, the process of reconciliation is simplified to one of update detection and propagation.

Between partitions, LOCUS allows concurrent updates, and provides a special mechanism to merge partitions together. The mechanism, however, is overly complex and does not scale, because it attempts to always maintain a correct picture of partition membership.

7.4 Coda

The Coda file system [KS91, SKK+90] uses two separate replication policies. First, Coda provides first-class replication at the volume granularity on a backbone of servers, aptly named server replication. The degree of replication and identity of the replication sites must be specified at volume creation time, unlike in Ficus. Although the replication degree and site locations can be changed, it is not clear that this is an easy process. Conflicts occurring between volume replicas are reliably detected, using a version vector scheme similar to that in Ficus.

Second, Coda provides second-class replication at the file granularity. Recall that second-class replicas are not peers; instead, they are basically cached-copies of a first-class replica, and as such do not have the authority to generate independent updates or communicate with any given replica. Clients of the servers cache whole files using a local cache-management process called Venus. Since the files are cached individually without other pieces of their volume, Coda emulates selective replication only at the clients. On file access, Venus creates a second-class replica of the file on the client locally. Future accesses and updates are directed to this local copy.

While connected, Venus is guaranteed to get a *callback* notifying it when a cached file has been modified. Furthermore, local updates to the cached copy are propagated in parallel back to all available volume replicas, called the available volume storage group or AVSG. The propagation of updates is performed by Venus using a two-phase, conservative protocol, with the client acting as initiator and coordinator. Venus assures that only the latest replica of the file within the AVSG will be accessed and cached at the client. Venus must detect whenever the AVSG changes size, and issue callbacks on files whenever the AVSG enlarges (newer data may be available) or whenever the AVSG shrinks and the preferred server (the server from which the local copy was generated) cannot be contacted.

While disconnected (the AVSG is empty), conflicts can occur when the local cached copy is updated, since these updates do not leave the local machine. When communication is restored, *reintegration* begins. If the reintegration of the cached copy with the first-class servers creates no conflicts, the operation is performed. However, should it create a conflict the operation is aborted (the data is saved in a *covolume*), and the user must manually integrate the changes. A failure to reintegrate one cached file can also cause the failure to reintegrate other cached files as well, due to the manner by which reintegration proceeds.

7.5 Summary of related work

While a number of distributed, replicated systems are in existence today, none are able to adequately provide solutions to the problems and sharing models discussed in Chapter 1. Deceit and LOCUS were not designed for the types of environments we are contemplating. The Coda system does not allow selective replication among the servers, which limits the scalability and use of server replication. In addition, Coda does not provide first-class replication at the clients, limiting their effectiveness as discussed in Chapter 2.

However, the Ficus selection replication service has some disadvantages relative to the above services, particularly with regard to Coda. For instance, the peer-to-peer file replication service is a more heavy-weight solution than the Coda second-class mechanism. There are clearly scenarios where the Coda solution performs adequately and true selective replication is not required. Such scenarios might include read-only caching, and possibly even replication of personal volumes where a human write-token exists, mediating potential problems caused by disconnection. However, we feel that the sharing models and environments discussed in Chapter 1 require the full utility of selective replication.

CHAPTER 8

Conclusions

8.1 Future Work

With a working selective replication service, there is much future work to be done in this area, such as:

1. Usage studies in real and varied population contexts
2. Studies into the relationship between replication and caching
3. System applications

These will be discussed briefly.

8.1.1 Studies into use

It will take time to study and analyze how selective replication is actually used in practice. Such studies are necessary to identify areas of the system that need enhancement or tuning. For instance, what is the average replication factor of most files? In general, are files replicated at almost all volume replicas, or at only a select few? How often do replication factors change? The answers to these questions will impact possible optimizations to the algorithms and implementation of the system.

In addition, recall that replication masks are the selective replication mechanisms for specifying where files should be stored at creation time. We will need to study the appropriateness of the default replication mask and the usability of the user replication mask. For example, is there a need for more complex controls on the replication factors of files at file creation time? How often does the user replication mask change? The answers to these questions might suggest that users require more complicated utilities to control where files become replicated.

8.1.2 Replication and caching

Selective replication seems an ideal platform for tasks such as whole-file caching of first-class replicas and predictive caching for primarily disconnected operation. Such caching services need to dynamically create and destroy local file replicas, and they often require these replicas to be first-class.

Whole-file caching could be used in Ficus so that accesses of remotely-stored files automatically generate local file replicas. This would be similar to the service Coda provides, only with first-class replication, making it strictly more powerful. After the file is no longer being accessed, the replica could be deleted.

Predictive caching could be utilized in mobile environments to automatically create replicas of "interesting" files on users' laptops prior to network disconnection. The predictive caching system in essence knows each user's typical work patterns, and generates first-class replicas of all files the user will want to access on the laptop.

With the selective replication service in place, all that is required for these caching services is software design and implementation. Relying on selective replication as a base for replication reduces much of the complexity of these problems.

8.1.3 System applications

Selective replication could be used to dynamically control the locations of file replicas within the network, according to high-level, user constraints. For example, a user might desire, for availability and performance reasons, three replicas of a given subtree; however, the actual storage locations of these replicas is irrelevant, as long as access time is consistently fast. The system becomes responsible for dynamically shifting replica location within the network in order to optimize communication patterns and keep the system load at a minimum. Replica location shifting would be simple selective replication addition and deletion commands. Such a system is similar to what the Deceit system provides, but in an optimistically-replicated environment. This would enable the system to scale in ways Deceit cannot, and to properly handle long-term communication failures.

8.2 Summary

In Chapter 2 we discussed design questions and performance bounds which must be adequately solved in order to conclude that a good service has been designed. The selective replication service discussed in this thesis satisfies the outlined qualitative performance constraints, and quantitative data illustrates that the system performs well. Nevertheless, such a service is useful only if the implementation's solutions to the design questions satisfy user requirements and are targeted at the correct environments.

Recall the initial hypothesized environments of Chapter 1. We desired a system which would provide a high degree of data availability, both for read and write access. We required the system to function correctly during long-term communication delays and intermittent communication failures. Finally, we needed a service which would provide first-class replication at a granularity small enough to enhance sharing opportunities, not hinder them. Selective replication meets these goals. It offers peer-to-peer optimistic replication at the file, granularity, and is ideal for such data-sharing modes as the home-office mode, the traveling businessman mode, and the geographically-separated collaboration mode. In all these working environments the system provides dynamic replication control while guaranteeing the maintenance of data consistency and correct execution of all distributed algorithms (like garbage collection) even when communication is at a minimum. We believe selective replication to truly satisfy user-requirements in these operational modes, opening the door to new and more productive sharing environments.

References

[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.

[GHM+90] 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.

[GPP93] Richard G. Guy, Gerald J. Popek, and Thomas W. Page, Jr. "Consistency Algorithms for Optimistic Replication." In *Proceedings of the First International Conference on Network Protocols*. IEEE, October 1993.

[Guy91] Richard G. Guy. *Ficus: A Very Large Scale Reliable Distributed File System*. Ph.D. dissertation, University of California, Los Angeles, June 1991. Also available as UCLA technical report CSD-910018.

[HKM+88] 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.

[HP94] John S. Heidemann and Gerald J. Popek. "File-System Development with Stackable Layers." *ACM Transactions on Computer Systems*, **12**(1):58–89, 1994. Preliminary version available as UCLA technical report CSD-930019.

[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.

[KS91] James J. Kistler and Mahadev Satyanarayanan. "Disconnected Operation in the Coda File System." Technical Report CMU-CS-91-166, Carnegie-Mellon University, July 1991.

[KS92] James J. Kistler and Mahadev Satyanarayanan. "Disconnected Operation in the Coda File System." *ACM Transactions on Computer Systems*, **10**(1):3–25, 1992.

[KS93] Puneet Kumar and Mahadev Satyanarayanan. "Supporting Application-Specific Resolution in an Optimistically Replicated File System." In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pp. 66–70, Napa, California, October 1993. IEEE.

[Ous90] John K. Ousterhout. "Why Aren't Operating Systems Getting Faster As Fast as Hardware?" In *USENIX Conference Proceedings*, pp. 247–256. USENIX, June 1990.

[PGPH90] Gerald J. Popek, Richard G. Guy, Thomas W. Page, Jr., and John S. Heidemann. "Replication in Ficus Distributed File Systems." In *Proceedings of the Workshop on Management of Replicated Data*, pp. 20–25. IEEE, November 1990.

[PPR+83] 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.

[PWC+81] Gerald Popek, Bruce Walker, Johanna Chow, David Edwards, Charles Kline, Gerald Rudisin, and Greg Thiel. "LOCUS: A Network Transparent, High Reliability Distributed System." In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pp. 169–177. ACM, December 1981.

[RHR+94] Peter Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. "Resolving File Conflicts in the Ficus File System." In *USENIX Conference Proceedings*, pp. 183–195. USENIX, June 1994.

[SBM90] Alex Siegel, Kenneth Birman, and Keith Marzullo. "Deceit: A Flexible Distributed File System." In *USENIX Conference Proceedings*, pp. 51–61. USENIX, June 1990.

[SGK+85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. "Design and Implementation of the Sun Network File System." In *USENIX Conference Proceedings*, pp. 119–130. USENIX, June 1985.

[SHN+85] Mahadev Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector, and Michael J. West. "The

ITC Distributed File System: Principles and Design." In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pp. 35–50. ACM, December 1985.

[Sie92] Alexander Siegel. *Performance in Flexible Distributed File Systems*. Ph.D. dissertation, Cornell, February 1992. Also available as Cornell technical report TR 92-1266.

[SKK+90] 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.

[WPE+83] 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.