University of California
Los Angeles

View Consistency for Optimistic Replication

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

by

Ashvin Goel

1996

The thesis of Ashvin Goel is approved.

Mario Gerla

Rajeev Bagrodia

Eli Gafni, Committee Co-chair

Gerald J. Popek, Committee Co-chair

University of California, Los Angeles

1996

Most people are thermometers

that record or register the temperature of majority opinion,

not thermostats that transform and regulate the temperature of society.

- Rev. Martin Luther King, Jr.

Table of Contents

List of Figures

List of Tables

Acknowledgments

This work would not have been possible without the support of my advisor, Professor Gerald Popek. I am most grateful to him for urging me to review, understand, and think as widely as possible, while still maintaining my focus. I would also like to thank Professor Peter Reiher whose constant guidance inspired me to do my best. Perhaps the best ideas came from members of the Ficus Project at UCLA. I have greatly enjoyed the numerous discussions (about everything) with John Heidemann, Dave Ratner, Geoff Kuenning, Mike Gunter, and all the others. Janice Martin and Apu Gupta proofread this thesis and provided immensely valuable help.

I would also like to 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.

My parents have trusted and supported my views and work; they understand why it takes so long to conduct good research; I wish to thank them. Finally, my thanks to Nadia, who stood by me as I took this long path to finishing my research, and supported me during those hard times that every graduate student comes to know.

Abstract of the Thesis

View Consistency for Optimistic Replication

by

Ashvin Goel

Master of Science in Computer Science

University of California, Los Angeles, 1996

Professor Gerald J. Popek, Co-chair

Professor Eli Gafni, Co-chair

An important objective of distributed and replicated systems is providing *highly available data* while preserving *data consistency*. *Conservatively replicated* systems maintain data consistency but severely limit the availability of data during network partitions. This makes them unsuitable in environments where network partitions occur frequently, or for long periods of time. *Optimistically replicated* systems allow updates to *any* available copy of the data. This provides high data availability. However, updates to older copies of data (or *conflicting updates*) can be made, thereby causing data inconsistency. Such inconsistencies are resolved when the network partitions heal, and copies exchange the new updates that have been generated. The time lag between an inconsistent update and its resolution depends on the length of the network partition. During this period, users may access intermediate states of data. This problem of accessing intermediate data states arises because optimistic systems provide consistency guarantees when copies communicate with each other and resolve their differences, and *not* when updates are generated.

We examine the consistency guarantees that can be provided when updates are

made in optimistic systems. These are called *immediate consistency* guarantees. A *view consistency* model that provides consistent views of data on a *per-user* basis is proposed. Maintaining data consistency is the responsibility of the clients, rather than the data servers in the view consistency model. The consistency criterion is enforced at each client based on information stored at the client. This consistency model is scalable to large numbers of data replicas since no global information is required for maintaining view consistency. It is well integrated with the optimistic consistency model, and provides high data availability. Optimistic systems that incorporate the view consistency model offer enhanced immediate consistency guarantees, at a low cost, and without significantly affecting availability.

CHAPTER 1

Introduction

Many distributed applications require *highly available* data. Data is highly available if it can be accessed at any time. The connectivity of the network affects the availability of data. For example, data can become unavailable when network partitions occur. To increase availability, data is often replicated. This allows an application to access data replicas that are present locally, or are present in a local partition.

Data must be *quickly accessible* in addition to being highly available. If several replicas are available, data should be provided from the best available replica. The choice of the best replica depends on factors such as bandwidth and latency of access to the replica.

Accessing distributed or replicated data introduces the *data consistency* problem. Data accesses are consistent when they do *not* reflect the intermediate states of data between accesses. For example, if replica A has been updated and the update has not reached replica B, then accesses to replica B may be inconsistent.

An important objective of distributed and replicated systems is providing highly available (and quickly accessible) data while preserving data consistency. The goal of maintaining data consistency conflicts with providing highly available data. Consistency allows access to consistent replicas only; availability (and accessibility) improves if any replica can be accessed.

Although numerous replicated systems have been built, most are suited for small numbers of nodes that are tightly connected together. The focus of this work is providing both availability and consistency to large numbers of nodes that are weakly and intermittently connected. Examples of such an environment include mobile, or highly geographically distributed systems. The next two sections present the *conservative* and *optimistic* consistency schemes that lie at opposite ends of the availability-versus-consistency spectrum. Both schemes are shown to be inadequate in a weakly connected environment. Then we present motivations for *view consistency.* View consistency uses a limited form of conservative consistency to provide improved consistency guarantees in an otherwise highly available optimistic system.

1.1 Instantaneous Data Consistency

A consistency scheme provides *instantaneous data consistency* if each access provides consistent data. In this section, we discuss some schemes that provide instantaneous data consistency. Section 1.2 below describes *eventual data consistency* where each access may not provide consistent data but data eventually converges to a consistent state.

Strong Consistency Traditionally, *one-copy serializability* has been used as a consistency criterion for replicated data [Pap79, BG81]. One-copy serializability ensures that the execution of a program does not affect or is not affected by the execution of other programs. In other words, each program (transaction) is isolated from the effects of other programs. One-copy serializability uses a *strong consistency* replica control algorithm to map logical data into physical replicas, and a *concurrency-control* algorithm (such as two-phase locking) to implement program isolation.

Strong consistency ensures that programs access the *latest data*, and all data copies see the *same order* of updates. It provides instantaneous data consistency since each access yields consistent data. Moreover, an access yields the same data as is obtained in a non-distributed or a *single copy* system. This makes strong consistency an attractive consistency model.

Unfortunately, strong consistency requires reliable communication links for high data availability [FM82, All83]. When a network becomes partitioned, data cannot be accessed in more than one partition. For example, suppose the network is partitioned into two sets of replicas, A and B. If updates are allowed in partition A, then access must be denied to replicas in partition B. Otherwise these accesses may operate on old copies of data. Note that strong consistency is conservative and does not allow accesses to partition B, even if partition A is never updated. This consistency model considers partitions to be failures. In some cases, these failures are rare and can be corrected quickly. Under such conditions, strong consistency is a useful consistency constraint.

Often communication links are inherently unreliable in mobile and highly distributed systems. In such situations, an application using the strong consistency criterion will not be able to access data, although the data is available in its partition. For several applications such as banking applications, reservation systems, and personal applications like appointment calendars, design documents, meeting notes, and in general, mobile file accesses, continued access is critical even during network partitions [PWC81, FM82, Sat89, GHM90, TTP95]. In mobile environments, network partitions are a normal mode of operation, rather than a failure condition that must, or can be immediately corrected [HPG92, KS92]. Mobile computers may not frequently connect with stationary networks, either because of the cost or the bandwidth restrictions of wireless communication. With strong consistency, either none of the disconnected mobiles will be able to access locally-stored replicated data, or at most, one of the mobiles will be allowed access. Similarly, communication links may by unreliable in highly geographically distributed systems. Moreover, the synchronization cost is high in such systems. Since strong consistency severely restricts availability when communication is unreliable, it is unsuitable as a consistency criterion in mobile or highly distributed systems.

Weak Consistency Strong consistency always provides the latest data version. However it does so by sacrificing availability. Several consistency schemes that enhance availability have been proposed. These schemes trade availability against consistency. One such scheme, *weak consistency*, is similar to strong consistency except that it ignores read dependencies [GW82]. Thus weak consistency provides instantaneous data consistency with respect to writes only. Writes update the latest copy of data, but reads can return older copies. This increases read availability of data. However, writes can only occur in a single partition. If write availability is essential, this consistency scheme suffers from the same problems as strong consistency. Moreover, in dynamic partitioning situations, even worse is possible: weak consistency will allow a reader to switch from a newer version to an older data version.

1.2 Eventual Data Consistency

The purpose of providing consistency guarantees is to maintain the semantics of data. Data accesses do not have to be serialized for maintaining the semantics of several types of data [FM82, All83, GAB83]. For other data types, some minimal inconsistency, or external intervention to restore the semantics of the data is often considered more acceptable than entirely inhibiting data accesses when communication is not possible [Sat89, GHM90, TTP95]. Consider a calendar that is shared by several users and replicated on each user's portable. A user can schedule meetings in his personal replica of the calendar, even when other replicas are not available. Later, if it is found that a conflicting meeting has been scheduled, one of the users can manually change the meeting time. Manual intervention may be preferable to disallowing accesses to the calendar until all (or a majority) of the replicas are available. This example shows that for effective use of weakly connected systems, update generation *must be made independent* of update integration (at other sites).

This approach is taken by optimistically replicated systems. An access can be made to any file replica at any time in optimistic systems. Updates are generated at a site, and integrated at other sites as data propagates to them. Unlike instantaneous data consistency schemes, no consistency is enforced during accesses. This can lead not only to reads returning old data, but also to conflicting updates, or updates being made to old copies. These inconsistencies are detected and resolved when data is integrated, rather than when it is generated. The resolution is done either automatically or manually depending on the semantics of the data type.

The integration and resolution of data at other sites ensures that replicas eventually reach identical states or become mutually consistent if no new updates are generated in the system [WB84, HHW89]. This consistency approach, which guarantees eventual consistency of data, is called *eventual data consistency*. Eventual data consistency provides high availability by allowing updates to be made in any order. This consistency approach is attractive when it is essential to provide access to data, and the network connectivity is unreliable or not present. Thus, optimistic or eventual data consistency schemes solve the availability problem of the instantaneous data consistency schemes.

However, eventual consistency schemes raise a different problem; they do not provide any consistency guarantees during accesses. It has been observed that this does not necessarily cause serious problems for shared files [KS92, RHR94]. Often, files that are heavily shared are not frequently updated by multiple users. Since the level of write-sharing is low, few consistency conflicts are generated. Unfortunately, this argument ignores read-dependency issues for shared files. Moreover, it does not take non-shared files into account.

When *read-dependencies* are ignored, older versions of files can be read. Moreover, data that is older than the last version accessed by a user may also be read. Since reads do not generate conflicts, eventual consistency schemes do not detect old reads, or provide any guarantees when the data is read. Users may remember the last version they had accessed in the past and use that version for future accesses. An example illustrates the problem. Suppose a user edits a replicated file and checks-in it using a source code control system. The replica crashes before it can propagate its changes to other replicas. The user gets an old version of the file from another replica when the file is checked-out again. Often, users do not update this old copy of the file. They must remember that this copy of the file is old and should not be updated. Moreover, if they do update the file, a conflict will necessarily happen. Worst of all, the user gets no hint that these problems might arise.

Eventual consistency ignores *write-dependencies* also. Thus writes can be made to older copies of data. This is uncommon for many shared files [KS92, RHR94]. However, often a single user may update multiple copies of data and thereby cause conflicts. This is different from the old read problem because the writes may be done without reading the data. Suppose a user is updating a replicated file on replica A. Replica A becomes heavily loaded after the first update has reached it. The system switches to using replica B, and writes are sent to replica B. This causes conflicts since an older version of the file at replica B gets updated. Both the problems of old reads and old writes occur because consistency guarantees are not provided when the access is done.

1.3 View Consistency

The optimistic or eventual data consistency model provides high availability, but does not provide instantaneous consistency guarantees. We propose the *view consistency model* that is built above an optimistic model, and provides instantaneous consistency guarantees. Moreover, it maintains most of the availability benefits of the optimistic consistency model. The instantaneous consistency guarantees are provided during accesses.

View consistency provides guarantees to the user that depend on the *actions* of the user and the *type* of the user. For example, consistent views can be provided to a process, or a machine, or a group of processes. The implementation of the consistency scheme may change with different user types. Thus this model consists of suite of consistency schemes. The next chapter describes this model in detail.

The aim of this work is to provide *instantaneous consistency guarantees* at a *low cost* in a *highly available* large-scale replicated environment. The view consistency model provides instantaneous consistency guarantees to individual users based on the past actions of those users. It is built above an optimistically replicated substrate that provides high availability and eventual data consistency.

Switching access to the fastest available replica is integrated into this model. The performance overhead of the view consistency model is small compared to other instantaneous consistency models since no global information is required for maintaining view consistency.

We describe two examples that illustrate the advantages of using view consistency for replicated data. The first example presents consistent *reads*. Consider a *web* or *FTP* site that replicates its data (for example, by mirroring) around the world for fault tolerance and faster access. Suppose a user down-loads a new version of his mailer from one of the sites. While installing the mailer, he discovers that the new mailer requires a new mailsend program, and he does not have this program. He tries to down-load the mailsend program, but the site from which he down-loaded the new mailer is experiencing heavy loads. The user's web or FTP client automatically switches accesses to another replicated site. View consistency ensures that the new mailsend program is as consistent as that which the user had seen at the previous site, or else denies access. This ensures that the user does not get an old copy of the mailsend program. The web or FTP client checks each replicated site until a consistent replica is found.

The next example presents consistent *writes*. Suppose a user edits a replicated file and he checks it back to his version control system. The site storing the file replica crashes before the check-in can finish. The check-in therefore occurs on another replica. If this replica is older than the replica on the crashed site, view consistency denies the check-in since this would necessarily cause a conflicting update.

In Chapter 2, view consistency issues are discussed. Chapter 3 describes the system on which view consistency has been implemented, and following this Chapter 4 presents the view consistency implementation in detail. Replica switching issues for improving availability and for providing view consistency are discussed in Chapter 5. Chapter 6 describes an algorithm for garbage collecting the version information that is stored for implementing view consistency. The performance and overhead of providing view consistency on an optimistic system are shown in Chapter 7. Chapters 8 and 9 discuss related and future work.

CHAPTER 2

View Consistency

Consider a set of cooperating processes accessing a replicated file. Each process is guaranteed access to some data replica. If the first data replica is not available, the second data replica is accessed, and so on. The replica access order can be chosen in several ways. For example, each process may access replicas in order of geographic distance, or all the processes may access replicas in the same order. Writes are propagated in the background on a best-effort basis and integrated at other sites so that replicas eventually reach consistency. This scheme allows high data availability since data is available as long as at least one replica is available.

There are several problems with this scheme. Inconsistent data can be accessed when the current replica becomes unavailable. The next data replica may not have data that is consistent with what the process has seen previously. Second, if a process splits into or spawns multiple subprocesses, these subprocesses can access different replicas and see inconsistent data. Finally, multiple concurrent writer processes (or concurrent reader-writer processes) cause conflicting updates (or stale reads). These problems motivate the need for consistent views.

The view consistency model provides a *consistent view* of replicated data to a *logical entity*. The condition that must be satisfied for a view to be consistent is called the *consistency criterion*. As we will see, different criteria can be used to define consistent views. One such criterion is that a view is consistent if it provides a data version that is *not older* than what an entity has seen previously.

Intuitively, a view is consistent if the data is consistent with the actions of the entity or a set of entities.

In the previous example, a process is not guaranteed consistent views when replicas become unavailable. Second, sub-entities of the process group entity do not see consistent views. Finally, different entities see non-consistent views.

Although each entity observes a consistent view of data in the view consistency model, different entities may see different views or different data. View consistency does not guarantee consistent views across entities. For example, each entity may see data that is the same as or newer than what *it* has seen previously. Thus an entity that has never seen a particular file may be provided any version of the file. However, an entity that has just created a new version of the file only sees the new version.

Entities in the view consistency model can be of different types. Examples of entities that see a consistent view include:

- a process, a group of processes
- a user on a machine, a user login session
- a machine, a group of machines, etc.

View consistency provides consistent data to an entity immaterial of the definition of an entity. The view consistency algorithm must take the entity type into account, since different entities have different properties and this affects the implementation. Properties of entities are further discussed in Section 2.2.

Let us illustrate a view consistency scheme with an example. Let a *process* be the entity. The consistency criterion is that the process will always read or write a version of a file that it had read or written previously. It will not be allowed access to a version that is older than or newer than what it had seen previously (it can of course update and create newer versions itself). Notice that the consistency criterion depends on the actions of the process: the process will read or write what *it has seen* previously. Thus two processes reading from or writing to a replicated file might see different data. Suppose a process accesses a file, and then another process writes to the same file. The former process will no longer be allowed access to the file. This consistency criterion does not ensure that the data itself is consistent, since different processes may access and write to different replicas, and yet obtain view consistent data. Previous work on this type of consistency criterion has also referred to it as a *session guarantee*.

Recall that the instantaneous data consistency schemes provide consistent data on each access. View consistency is similar to these schemes since it also provides consistent data on each access. However it differs from these schemes in four important ways. First, while the data consistency model keeps the data in a consistent state, and thus provides guarantees about the state of the data, the view consistency model provides guarantees to the entity immaterial of the state of the data. Even though the data may conflict, view consistency will attempt to provide data until the consistency criterion is not satisfied. Second, the consistency criterion is dependent upon the actions of the entity. For example, a consistency criterion can be to provide data versions that the entity has seen previously. If the data version does not exist, the criterion is not met, and data is not made available. This criterion depends on the past actions of the user. Third, the view consistency criterion is enforced by the entity, and not by the data servers. It is this difference that makes view consistency significantly cheaper to implement than data consistency schemes that require consensus among the data replicas for each access. Finally, although the consistency criterion may be the same, different entities may see different versions of a file. Again, this happens because the

	Instantaneous Data Consistency (*Conservative*)	**Eventual Consistency** (*Optimistic*)	**View Consistency**
Data Guarantees	Consistent on each access	Eventually consistent	Consistent on each access per entity
Guarantees depend on actions of entity	no	no	yes
Consistency Enforcement	Server-based	Server-based	Entity-based
Availability	Low	High	High

Table 2.1: Differences in the instantaneous data, eventual and view consistency models

criterion applies to each entity, and not to the data. Table 2.1 shows these differences. It shows the relationship among the data, eventual, and view consistency schemes.

There are two factors that characterize a consistency scheme in the view consistency model.

Entity: The entity is the smallest unit that observes a consistent view of data. In the previous example, each process is the entity. Other examples of entities include a user login process, a machine, etc.

Consistency Criterion: The consistency criterion is the specific guarantee that is provided to each entity. The consistency guarantee in the previous example is that each process sees a version of a file that it has seen before.

The view consistency model consists of a suite of consistency schemes that differ in the consistency criterion and the entity to which they provide consistency. The rest of the chapter investigates different types of useful consistency criteria and entities in the view consistency model.

2.1 Consistency Criteria

A view is consistent with respect to a consistency criterion when it satisfies the consistency criterion. A criterion can be a local, neighbor, or global one. A *local criterion* allows an entity to access data that is consistent with its own actions. A *neighbor criterion* ensures consistency with the actions of some small set of entities. A *global criterion* allows an entity to access data that is consistent with the actions of all entities that access that data.

The local criterion only depends on the actions of the entity. If an entity has never accessed any version of the data, then any version satisfies this criterion. It is relatively simple and cheap to implement since the criterion can be checked and enforced by the entity itself. Entities do not have to coordinate among themselves to ensure that they access data that is consistent across entities.

The neighbor criterion provides data to an entity that is consistent with the accesses of a small group of known entities. Assuming that each entity (not just some of them) requires consistency with all the entities in the entity group, then this small set of known entities can be grouped together to form a single entity. The neighbor criterion then becomes a local criterion for this single entity-group. For example, a process may be neighbor consistent with all the processes in its process group. If all the processes in the process group require neighbor consistency, the process group can be considered a single entity, and local consistency can be enforced for the process group.

The global criterion requires that data is consistent with the actions of all other entities. Since the number of entities that can access data and the data replicas that these entities access is arbitrary, this criterion requires examining all the replicas to see if any accesses have been made. In particular, it will only allow access to the latest data. A global criterion is thus no different from a strong consistency scheme that provides the latest data on each access, and will not provide high availability. We will not consider it any further.

As discussed above, the neighbor criterion is equivalent to the local criterion when the granularity of the entity is changed to include its neighbors. Thus we only study the local criterion. We have suggested that it is cheap to implement because the criterion can be checked and enforced by the entity itself. The obvious disadvantage of this scheme is that different entities may access different data. If concurrent writes are not common, this scheme provides high data availability as well as good consistency guarantees.

The differences in the consistency criterion that are discussed above result from providing consistent views with respect to different sets of entities. Another dimension along which consistency criteria can differ is in the data that is provided. In the *later-version* criterion, each read or write access provides data that is the same as or newer than what the entity has read or written previously. For brevity, we call the same or a newer version of data, a later-version. This definition of view consistency serializes all the reads and writes of the entity. If there are multiple sub-entities of an entity (for example, processes of a process group entity), read-read dependencies can be excluded from the definition of a later-version.

Alternately, a consistent view can be defined in terms of a *same-version*, where each read or write access provides data that is the same as what the entity had accessed previously. This definition reduces availability as compared to the later-version definition because newer data versions that satisfy the later-version criterion do not satisfy the same-version criterion. An example where the same-version consistency criterion is very useful is during the loading of an executable. If successive reads during executable loading provide newer (and different) versions of the executable, the executable will probably not run, if not crash the application.

These examples of a consistent view have been defined in terms of the same or later versions rather than in terms of the latest version. Both these definitions are dependent on the actions of the entity. A latest version definition requires a conservative consistency policy such as strong consistency. We will not study any scheme that requires a conservative policy because it provides low availability. For this same reason, we will only focus on the local criterion, and not the global criterion. The local criterion can yield either the same-version or the later-version of the data. The implementation of the two criteria does not differ significantly. As discussed above, the same-version criterion reduces availability as compared to the later-version criterion. We will therefore focus on the later-version criterion in the rest of the thesis, mentioning the differences between the two criteria when needed.

2.2 Entity Definition

Until now, consistent views have been defined for a generic entity. In this section, we describe a representative set of entity types that benefit from consistent views. The algorithms for implementing view consistency depend on certain attributes of an entity. We will describe these attributes before discussing the different entity types. The attributes determine the algorithm and thus the cost of implementing view consistency.

2.2.1 Entity Attributes

Each entity exists for a certain period of time. Long-lived entities are *persistent* entities while short-lived entities are *transient* entities. The distinction between long and short-lived entities is arbitrary. One way to make this distinction is by considering entities that survive machine reboots to be long lived. Extra effort is needed to provide consistent views to a persistent entity. For example, the information that is used to decide whether the consistency criterion is satisfied for a persistent entity must be kept on secondary storage.

An entity may be composed of sub-entities that can exist by themselves, and that can access data independently of other sub-entities of the entity. Such an entity is called *complex*. Entities that do not have sub-entities associated with them are *simple*. A complex entity can access copies of the data via multiple streams of execution. Thus complex entities need to coordinate their accesses to see consistent views. For example, a process group entity is complex because it consists of multiple threads of execution. These threads must coordinate their accesses so that each process in the group always accesses (say) later versions of data.

A complex entity is *centralized* if all its execution streams are confined to a single machine. If the execution streams originate from different machines, the complex entity is *distributed*. It is simpler to coordinate the accesses of centralized entities than distributed entities.

A distributed entity is *identifiable* if the execution streams originate from a known or identifiable set of machines. If the distributed entity consists of

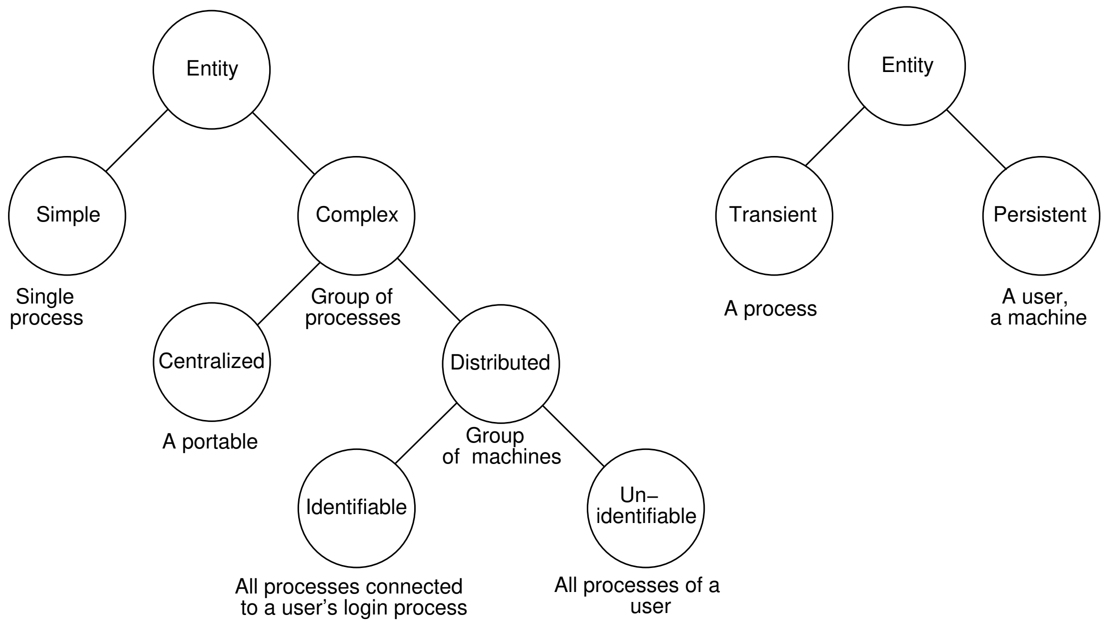

Figure 2.1: The entity hierarchy

sub-entities whose origins cannot be identified, it is called *unidentifiable*. The distinction between identifiable and unidentifiable entities is subtle. An identifiable entity is denied access to data only when the same or newer (later) versions of data are unavailable. An unidentifiable entity can be denied access either because later versions of data are not available or because the sub-entities of the entity cannot be coordinated at a particular time. Mechanisms such as primary coordinator, token passing, or voting are needed to synchronize the distributed accesses of an unidentifiable entity.

This entity hierarchy, along with examples of different types of entities, is shown in Figure 2.1 and Tables 2.2.

Examples	**Transient**	**Persistent**
Simple	Single process	Single machine A user on a single machine
Complex	Group of processes	Group of machines All processes of a user

Table 2.2: The persistence and divisibility attributes of an entity

2.2.2 Entity Types

The entities shown in Figure 2.1 can be categorized into three generic entity types: processes, machines and users. The attributes of these entities are discussed below. In the examples presented in this section, we assume the local later-version criterion. As discussed at the end of Section 2.1, we believe that this criteria is most useful for applications and provides the highest availability.

Process entity At each file access, each process sees a later-version of the file. A process is a simple, transient entity, i.e., it is single and does not survive machine reboots. A set of processes (for example, subprocesses of a process) is a complex transient entity. This set of processes sees later-versions of the file. The subprocesses can belong to the same machine (centralized entity) or to different machines (distributed entity). An example of a distributed-process-group entity is the set of processes (across all machines) that belong to a user in a login session. If all these processes have a single controlling terminal, the set of machines on which these processes are running is known, and this entity is identifiable. The processes can be controlled from this single controlling terminal. This is an example of the primary coordinator approach to synchronizing distributed accesses of an entity. Other solutions such as token passing or voting can also be used to coordinate the accesses of the sub-entities.

Machine entity A machine is another type of an entity. The machine entity is simple and persistent. All processes or users accessing data from a single machine see a later version of the data. This is not equivalent to "all the processes on a machine" entity because unlike processes, the machine entity is persistent. A consistent view to the machine entity ensures that processes see later versions even when the machine is rebooted.

A known group of machines is another type of entity. Here, all processes on any of the machines in the group see later versions of data. This entity is persistent, but distributed. Since the data access streams originate from a known set of machines, the entity is identifiable.

User entity A user sees a consistent view when each access by any process belonging to the user provides later-versions of data. The user, unlike the process, is a persistent entity and crosses login session boundaries and machine reboots. It is a complex and distributed entity since it can own multiple processes which access the same data, and these processes can originate from multiple machines. Moreover, it may be an unidentifiable entity since these processes may not be easy to locate and the data access streams can originate from an unknown set of machines. As discussed earlier, unidentifiable entities require conservative consistency schemes. Thus we will only consider a user entity on a single machine or a known set of machines. All processes of the user accessing data from this set of machines access later-versions of the data.

This chapter defines the view consistency model and shows its relationship to other consistency models. View consistency can be characterized by an entity and a consistency criterion. The next chapter describes an optimistic replicated file system on which view consistency has been implemented.

CHAPTER 3

System Model

Recall from Chapter 2 that we want to focus only on the local view consistency criterion because it allows high availability of data. However, the local criterion allows conflicting updates. View consistency must therefore be implemented over an optimistic system that detects and resolves conflicting updates.

We have implemented view consistency on the Ficus optimistically replicated file system. This chapter presents the features of Ficus that are relevant to the view consistency design or were changed for the view consistency implementation.

3.1 Stacking in Ficus

The Ficus file system has been developed using a stackable layers approach for modularity. Stackable layers allow third-party vendors to deliver shrink-wrapped file-system software modules that contribute file-system functionality, and do not require replacing or rewriting the existing parts of the file system. The stackable layers approach allows vertical composition of file-system layers. Each layer must adhere to a well-defined operation interface when making calls to lower layers. Moreover, it can assume that calls from upper layers also follow this same interface. This mechanism of fixing the interface between different functionality layers allows each layer to be developed independent of the other layers.

Figure 3.1 shows the layers in Ficus that are important for replication and

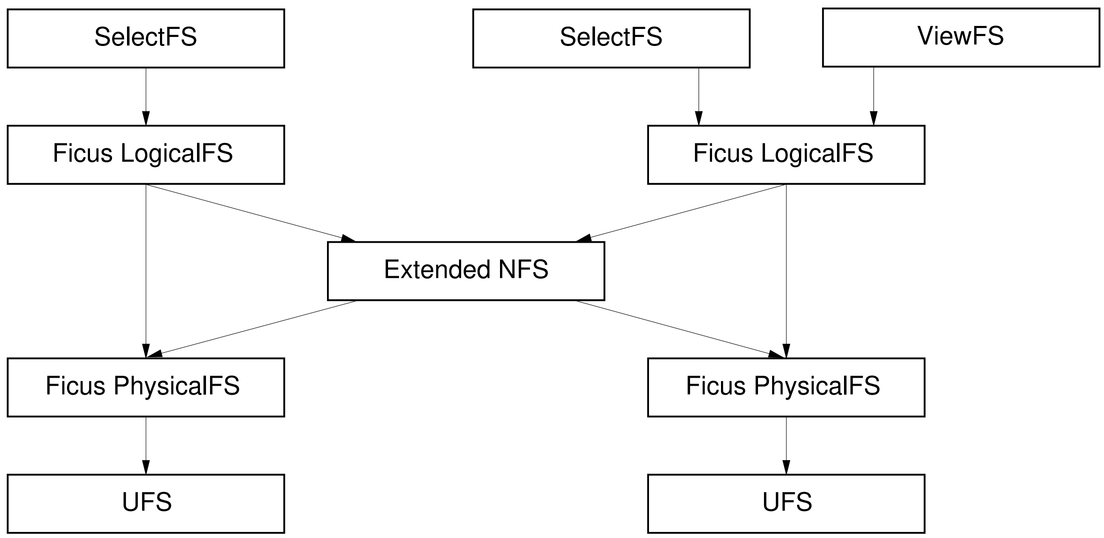

Figure 3.1: The basic layers in the Ficus stackable file system

view consistency. The view consistency implementation resides in the *ViewFS* layer. Either little or no changes were needed in other layers to support view consistency. The implementation of the ViewFS layer is discussed in Chapter 4. Here we briefly describe the function of each Ficus layer and discuss some overall features of the stackable layered Ficus implementation.

The *Ficus physicalFS* implements the Ficus directory structure and the Ficus attribute storage service. It implements the functions for accessing each file replica. The *Ficus logicalFS* knows about replication and maintains a mapping between a file and the file replicas implemented by the Ficus physicalFS. Mounting and unmounting of Ficus volumes and replica switching (the ability to switch access from one replica to another) are implemented in the *SelectFS* layer. The SelectFS layer functionality is discussed in more detail in the next sections.

Ficus stacks are *dynamically composable* at boot time. The position of the layers in the stack does not have to be specified at compilation. The layers that must be stacked, and the order in which they must be stacked, is specified during boot time. The Ficus stack is then built bottom-up. Layers can be switched without affecting other layers. For example, the UFS layer can be replaced by a log-structured file-system layer (if its implementation out-performs the UFS implementation) without affecting any other layer. There are some semantic limitations to the stacking order. For example, the logicalFS and the physicalFS layers in the Ficus stack cannot be interchanged because the logicalFS layer requires the services of the physicalFS layer.

The ViewFS layer that implements view consistency is an enhancement of the SelectFS layer. Today, both layers can be stacked in parallel in different parts of the Ficus name space. Files in the ViewFS name space are view consistent while those in the SelectFS are not. This has several advantages, such as allowing easy debugging of new file-system code, and performance testing where both the old and the new code can run in parallel.

Another feature of the Ficus stacks is *fan-in* and *fan-out*. In the figure above, Ficus stacks in different address space boundaries are connected by stack fan-in and fan-out. Fan-in and fan-out can occur within the same machine boundary also. Fan-in allows clients on different file systems to access the same file and fan-out allows a single client to access files in multiple file systems. Ficus uses an extended NFS layer to transport data between machines. This layer sits between the logical and the physical layers and provides the logical layer at each site with information about other replicas. No extra mechanism is required for adding the NFS layer to the stack since fan-in and fan-out are part of stacking.

Another important feature of the Ficus stacking is that it is *extensible*. New layers with custom operations can be supported anywhere in the stack. Earlier, we had stated that the operation interface between layers is fixed. How is it possible to have a fixed interface and allow new operations be part of the interface? The solution is to provide a catchall bypass operation in each layer that passes operations to the lower layer if it does not understand the operation. An error is returned if an operation is bypassed to the lowest level of the stack since the operation cannot go down any further. The bypass operation allows, for example, the extended NFS layer to pass Ficus operations (like getting Ficus attributes) from the logical to the physical layer across machine boundaries. Layers that do not add new functionality to an operation (such as getting UFS attributes in the SelectFS) can simply bypass it down to the lower layers until the appropriate layer (UFS) handles the operation. In general, the nature of the bypass operation depends on the semantics of the layer. For example, the SelectFS layer may do replica selection immaterial of the operation that is passed down to it. Thus, replica selection must be part of the bypass code in the SelectFS layer. Other layers may also have layer-specific code in the bypass routine of that layer.

3.2 Reconciliation and Conflict Resolution

Ficus provides highly available data by allowing unsynchronized updates to different replicas of a file. This can cause conflicting updates or updates being made to older copies of data. While a conservative system disallows conflicting updates, an optimistic system detects and resolves, or rolls back conflicting updates. The detection of conflicting updates is done by using version vectors in Ficus.

There are six basic operations that can cause version vectors to change in Ficus. At arbitrary times a user will update a file. This action results in a *file replica update* which is immediately followed by *Ficus update notification.* The update notification sends a message to all other currently accessible file replicas. This message is a one-shot, best-effort attempt to trigger *Ficus update propagation* by other replicas. When a site receives an update notification message, it invokes update propagation from the updated replica to the old replica. This propagation will fail if the versions conflict or if some replicas are not currently available. Since the update notification and propagation operations can fail, we cannot depend on these operations alone for files to reach eventual consistency. To encourage files to reach eventual consistency, pairs of file replicas are periodically compared using *Ficus reconciliation*. Replicas pass information directly or indirectly through a gossiping protocol [GPP93], ensuring that information exchanged in pairwise reconciliations eventually reaches all replicas. Replicas compare their data by using vector timestamps. A vector timestamp A is later than another timestamp B, if each of A's components is greater than or equal to the corresponding component of B. Suppose A and B are the vector timestamps of the two replicas that are reconciling. If A is not later than B, and B is not later than A, a conflicting update has been made to the file. If conflicting updates have been made to a file, the conflict is reliably detected during reconciliation and *Ficus automatic conflict resolution* is attempted. When two file replicas conflict, the file name and its type are used to search for an applicable conflict resolver [RHR94] that attempts to merge the replicas into one version and distribute the new version. If a resolver is not found or the merge is unsuccessful, the user must manually merge file versions.

The view consistency implementation does not directly modify the reconciliation and resolution operations in Ficus. However these operations are essential because view consistency allows conflicting updates and these must be resolved. Ficus reconciliation is used for the garbage collection of view consistency information. It is described in more detail in Chapter 6.

3.3 Volumes and Location Information

A file must be located before it can be accessed. Since files are replicated in Ficus, the location information consists of one or more tuples of host and pathname information. Each host and pathname tuple uniquely determines a replica of a file. Ficus is a distributed file system, and like other distributed file systems, it attempts to achieve name transparency, or provide the same view of the name space from any site. Thus, a file is named identically immaterial of the host from which it is accessed. This requires the location information to be replicated, or the site that stores the location information becomes a bottleneck and a single point of failure. The replicated location database must be kept consistent like other data.

NFS ignores the replicated location database problem by requiring the system administrator to know the location of files. The location information is kept at a file-system level rather than per-file since there are large numbers of files in a system as compared to much fewer file systems. System administrators know the location of each individual file system in the network and statically attach it to individual hosts using a mechanism that is very similar to *mounting* in UNIX. However, if an administrator decides to mount a file system at a different point in the name space, this information is not propagated to other sites that mount this file system. Ficus has been designed to scale to large numbers of hosts, replicas and users. Clearly, a mechanism that requires agreement by human convention is infeasible in such an environment.

Ficus has a novel solution to the replicated location database problem. To scale well, Ficus maintains the location information at a mini file-system level that is called a *volume*. Each volume lies within a file system and consists of a connected set of about 100 to 10000 files and directories. In a conventional single-host

UNIX system, a single mount table contains the mappings between the *mounted-on* directories and the *roots of the mounted* file systems. Ficus maintains a similar location database for the root of the mounted volume in each of the mounted-on directories. For example, volume A may be replicated at five sites and volume B that is mounted on one of the directories of A may be replicated at seven sites. The location information for B is then stored at the five sites at which volume A is stored. If volume A must be traversed before B can be accessed, the location information for B will always be available. Ficus assumes a global name space and typically enforces the above restriction.

We have discussed where the replicated location information is kept. However, the information must also be kept consistent. Typically, systems have maintained the consistency of the location information by using a mechanism separate from the one used for keeping the file-system data consistent. This adds complexity to the system. The Ficus location information is stored in files in the mounted-on directory. Thus the *same* mechanism that updates, reconciles and resolves file and directory information in the mounted-on volume can be used to maintain the consistency of location information. Note that the location information is maintained optimistically just like data. Once the location information is known, switching to and accessing consistent replicas can be implemented as discussed in the next section.

3.4 Replica Selection

The volume location information is optimistically maintained at mount points in the Ficus volumes. This information contains a unique volume identifier and the hosts at which the volume replicas are stored. In order to access a file in a volume, the volume is first mounted using the location information. Initially, a single replica is mounted to speed up the mounting process. Later, other replicas are mounted on demand.

Most file operations do not specify the replica that must be accessed. The system must choose and provide access to a specific replica of the file. This section provides an overview of the *default* replica selection process. Consistency and availability criteria are not taken into account in the default replica selection process. These criteria modify the default process and are described in detail in the next two chapters.

Replica selection occurs for the first time during a kernel `lookup` operation. The `lookup` operation is similar to a file open but is also invoked for several other file operations. For example, a `stat` operation in UNIX that gets file attributes invokes `lookup`, but does not open the file. It takes a pathname of the file as input and returns a *vnode* or a file handle for the file. Two simple policies are used for choosing a replica of the file during `lookup`. First, the fastest available replica is chosen when a *volume root* (root of the volume) is opened. The determination of the fastest available replica is described in Chapter 5. Second, the replica of a file (or a subdirectory) is chosen to be the same as the currently accessed replica of the parent directory. These rules ensure that the same replica is accessed for all files within a volume, and this replica is the fastest available replica. This simple replica selection policy performs well for many applications; the number of conflicting updates that are generated is extremely low as reported in [RHR94]. A complication occurs when files are *selectively replicated*. Replicas of a selectively replicated file are stored at some subset of sites at which the volume is replicated. Modifications to the replica selection process for selective replication are described in [Rat95] and are not discussed here any further.

Default replica selection guarantees that file operations will keep accessing the same file replica until the replica becomes unavailable. In that case, the replica is switched to some other available replica. After switching to the available replica, the file operation is attempted again. This process is done until either the file operation is successful, or there are no available replicas. Switching is not attempted if a file was open when the replica became unavailable. Instead, the next operation on the open file fails. This is done because the default replica selection scheme does not guarantee anything about the consistency of the next available replica. Here, availability is sacrificed to avoid generating potentially conflicting updates.

When a file replica becomes unavailable, it is added to a volume-specific *down-replica* cache. There are two types of file operations that are possible: operations that create new file handles (such as `lookup`), and operations that access existing file handles. Operations that create file handles ignore the down-replica. If an operation is invoked on an existing file handle that references the down-replica, and the file has not been opened (as is the case with the `stat` operation explained in the example above), the file handle replica is switched to an available replica even before the operation is attempted. This avoids long NFS timeout delays for an operation that is very likely to fail. Eventually the volume-root replica also gets switched to an available replica. After that, the rules for replica selection described above ensure that all files will access the available replica. A timeout feature prevents a replica from being in the down-replica cache forever. Chapter 5 discusses a generalization of the down-replica cache.

A list of replicas is kept for each volume in the kernel. It is used to find available replicas and is a copy of the location information stored in the mount points of volumes. Recall that the location information is kept optimistically. This implies that there may be times when a replica has been added, but the replica addition has not been noticed by a host. In that case, although the replica may be available, switching to it is not attempted.

Although most file operations do not specify the replica that must be accessed, there are some operations that require accessing a specific file replica. For example, file reconciliation between pairs of replicas requires accessing specific file replicas. For such accesses, replicas are not switched even if the replica becomes unavailable.

CHAPTER 4

View Consistency Implementation

This chapter presents the view consistency algorithm for a generic entity and then discusses the issues in the implementation of the algorithm for a centralized entity. The distributed entity case has not been implemented, and issues related to it are described in Section 9.2 on future work.

Recall that we only study the local view consistency criterion since it provides high data availability and can be implemented relatively cheaply. The local criterion allows an entity to access data that is consistent with its own actions, or provides data versions that are *later* than what the entity has seen *previously*. This requires keeping track of the data version that was last read or written by an entity.

The basic view consistency algorithm involves tracking the last version of data that was accessed and using this version to ensure that the next access yields a later data version. Figure 4.1 shows this simple algorithm. The `viewMediator` function is called by each file operation that returns data as a result of the operation. The version and the replica (collectively called the *view-entry*) that was last accessed for a particular file is obtained from `readViewEntry`. This information in the viewEntry is used to switch to a later file replica in `switchToLaterReplica`. If the view-entry for a file does not exist, the file has never been accessed.[1] In

[1] Even if the file has been accessed, the view-entry may not exist because it has been garbage collected, as discussed in Chapter 6. Such a file can be considered as never having been accessed.

```
viewMediator(file, entity, fileOperation)
{
    viewEntry = (viewVersion, replica) = readViewEntry(file, entity);
    if (viewEntry != NULL) {
        replica = switchToLaterReplica(file, viewEntry);
    } else {
        replica = switchToAnyReplica(file);
    }
    (data, newViewVersion) = fileOperation(file, replica);
    writeViewEntry(file, entity, newViewVersion, replica);
    return data;
}
```

Figure 4.1: The general view consistency algorithm

this case, `switchToAnyReplica` is invoked, since any replica can be accessed. The switching mechanism in the "switchTo" functions is described in detail in Chapter 5 when we discuss the issues involved in switching for availability. The file operation is enhanced to return the data and the version of the data that was accessed. This version and the replica that were accessed are stored by `writeViewEntry` as the new view-entry for the file, and are used for the next access.

Note that *only* the `readViewEntry` and `writeViewEntry` routines require the entity type. View consistency can be provided to different entity types by appropriately modifying these routines. The cost and the complexity of providing a consistent view to an entity thus depends on the cost of reading and writing view-entries.

4.1 Reading and Writing Version Information

The view-entry information must be stored and read as shown in Figure 4.1. Several factors must be taken into consideration before designing a view-entry storage and retrieval service. First, these operations are in the critical path of a file operation. For the view consistency overhead to be low, the cost of these operations must be small compared to the cost of a file operation. Second, this information must be stored persistently for a persistent entity. Even for a transient entity, this information may become large enough for it to be stored on disk. Third, the storage and retrieval frequency of the view-entries are roughly the same, since `readViewEntry` and `writeViewEntry` are invoked for each file operation that goes through the `viewMediator`. Finally, the view-entry information must be stored for every file that is ever accessed. The number of files that have been accessed and thus the number of view-entries grows with time. Therefore, the storage service must be able to handle large numbers of view-entries.

The system must delete view-entries that are no longer required, or else the number of stored view-entries will increase with time. We note that view-entry deletion is not in the critical path of a file operation, and its execution can be made relatively independent of the file operation and the view consistency algorithm. The truncation of this information is thus described separately in Chapter 6.

4.1.1 View-Entry Storage Service

The view-entry storage requirements are large storage, fast read and write access and persistence. A good database satisfies all these requirements. We chose the *db* package built at UC Berkeley by Margo Seltzer [Sel91]. The db package is relatively small, provides compact persistent storage and caches large chunks of the database in memory for efficient access.

The view consistency implementation has been done within the Ficus kernel. There were two choices regarding the placement of the database package. The package can be run as a user-level daemon process, or be made a part of the kernel. There are several advantages of using a daemon process. First, the database package does not have to be ported to run in the kernel. A port involves considerable effort since the file-system interface exported by the kernel is very different from the interface within the kernel. Second, the kernel size is not affected since the database package is a separate user-level process. This is especially important because the package caches large parts of the database in memory. Third, bugs in the database either do not affect the kernel or at most affect the file-system name space that is view consistent. The main disadvantage of using a separate process is that at least one kernel-to-user and one user-to-kernel context switch is needed every time the database is accessed. Since the view-entry must be read before, and written after a file operation, four extra context switches are done for any file operation that goes through the `viewMediator`. Since this is unacceptable, kernel caching is used in our implementation to solve this problem.

4.1.2 View-Entry Caching

The view-entry returned by `readViewEntry` can be cached in the kernel. The user-level daemon does not have to be invoked as long as the view-entry of a file exists in the cache. If the cache hit ratio is high, the amortized per-file-operation overhead of the `readViewEntry` routine will be significantly reduced. This approach can potentially provide an efficient solution to the view-entry access problem with little extra effort.

Many file operations require a file handle or a *vnode* rather than the file name. The view-entry is cached with the file handle and is available in the kernel as long as operations occur on the file handle. It is cached the first time the `viewMediator` code is invoked on a file handle and can be removed from the cache when the file handle is destroyed. In practice this implies that the view-entry is read from the view-entry database on each file open. After that the current value of the view-entry is cached in the kernel until the database is updated with this cached view-entry.

The `readViewEntry` operation involves context switches as well as reading from the disk (assuming that the view-entry is not cached in memory in the database). Thus each file open requires an extra disk operation.[2] Assuming that normal opens require a single disk operation, the `readViewEntry` operation adds 100 percent overhead to an open. This is very expensive in normal UNIX environments where opens can constitute 15 percent of the total time spent in file

[2]We assume that the key of the disk block is in memory.

operations [FE89].

The number of `readViewEntry` operations that go to disk can be reduced by caching the view-entries more aggressively. UNIX kernels already cache file handles in the *dnlc* or the name lookup cache. A mapping is maintained between the file name and its file handle. Since we keep the view-entry with the file handle, it stays in the kernel as long as the file handle stays in the cache. The effectiveness of this cache depends on the locality of file accesses. A more aggressive solution caches the view-entries independent of the dnlc cache. However, we have not implemented such a cache since it is not clear that a separate cache will significantly improve performance compared to a dnlc cache.

Another method of reducing the delay in reading view-entries is to perform the file operation *optimistically*. The file operation can be done in *parallel* with reading the view-entry. If the file operation yields a data version that is later than the version in the view-entry, the operation is successful. Otherwise, the operation must be tried again with a consistent replica. We see in Chapter 5 that this optimism pays off since the file replica that will normally be accessed by the file operation in Figure 4.1 (even if `switchToLaterReplica` has not been invoked) is the same replica as the one in the view-entry. Thus the replica will be view consistent. The optimistic file operation is especially useful for remote data because the view-entry can be obtained from the local disk while the data is obtained in parallel from a remote disk. Unfortunately, the `viewMediator` function is in the Ficus kernel, which does not support kernel threads. Although it is possible to implement parallel operations, the overheads of the implementation are significant.[3] We have therefore not included optimistic file operations in the current implementation.

[3] A separate user-level context is needed. Its overheads include context switching, data communication and synchronization.

4.1.3 View-Entry Update and Storage

Figure 4.1 shows that both the data and its version are obtained from the file operation. The version information is then stored by `writeViewEntry`. Normally file operations return only the data. Obtaining both the data and its version from a file operation requires changes to the file operation at the data servers. If the file operations cannot be changed at the data servers, the version information can be obtained *separately* after getting the data. Such a version is the same as, or newer than[4] (later than) the data version that is obtained. Since the version information is later than the accessed data, the next access also yields a later version. There are two problems with getting the version information separately. First, the same-version criterion can not be supported. The version information may be newer than the data retrieved, and thus the next access does not yield the same data version. Second, it necessitates going over the wire twice instead of just once, since the version information must be obtained separately. We have modified some of the file operations (such as `lookup`) in Ficus to return the data and the version information. For other operations, the version information is obtained separately. As an aside, some operations (e.g., `lookup`) can return cached data. In this case, the version information does not have to be obtained from the remote servers again since the cached data version is already known. The performance difference between getting version information along with the data versus separately is shown in Chapter 7. These figures show that obtaining the version information together with the data drastically reduces the view consistency overhead. We therefore plan to change all the relevant Ficus operations to return the version information and data together.

Once the data version is obtained, it must be stored in the view-entry. If

[4]The version is newer if the file is updated by another entity after the data is retrieved, but before the version is obtained.

the file has never been accessed, the view-entry does not exist, and it must be initialized with the new data version. If the view-entry does exist, but the version has changed (it can only have increased), or a new replica has been added or deleted, the view-entry in the kernel is updated with the new version. A dirty bit is set when the version is updated in the cached view-entry.

For view consistency, the updated view-entry (as determined by the dirty bit) must be stored on disk before the data is returned to the user. This ensures that the system records a file version on persistent storage that is either the same as or newer than (later than) what the user has seen. The next access yields data that is later than this recorded version and is therefore view consistent. Note that recording the view-entry before returning the data is similar to write-ahead logging for providing atomicity in transactional systems.

Unfortunately, it is very expensive to write to disk after each version update, since each file write then incurs an extra disk operation. Moreover, many directory operations also update versions, and this requires extra disk operations. Further, in UNIX kernels, writes are generally deferred for efficiency. Thus immediately writing an updated version does *not* necessarily achieve its desired effect of recording a later version to disk before returning the data to the user.

We decided to write updated versions to disk on a file `close`. However, there are certain operations (such as directory operations) that update versions without doing an explicit open. All such operations have a file handle associated with the file. Therefore, updated versions are also stored to disk when file handles are destroyed. Since files may remain open, or file handles may exist for a long time, updated versions are also periodically stored back to the view-entry database every 30 seconds. The view-entry database flushes its own updated memory entries to disk every 30 seconds. These mechanisms together ensure that every view-entry that was updated more than 60 seconds ago has been stored on disk. Thus a 60-second *window of vulnerability* exists during which users may see a data version that is not recorded persistently. After a machine crash and boot, users may see older versions of those files that they had been accessing a minute before the crash.

4.2 Operations That Need Consistency

Until now, we have not specified the file operations that require view consistency. Data operations like file reads and writes clearly require view consistency since only later file data versions must be provided. However, file operations also read and modify file attributes and the name space. Many users and applications store important information in the file attributes. Thus file attributes must be consistent with the file data. For example, the UNIX `make` command requires consistent file times or else it can miss file dependencies.

Directory Consistency Directory operations modify the name space of the file system. For example, a `rename` changes the name of a file or a directory. View consistency is also useful for directory operations. For example, suppose a user renames a file. The replica that was being accessed then becomes unavailable, and the system starts accessing an older replica of the directory. The new file name is no longer visible, and it causes confusion. View consistency on directory operations disallows accesses to the older directory. Only later directory versions are accessible.

Directory operations occur frequently in UNIX environments. Thus directory view consistency can add significant overhead in a view consistent system. Chapter 7 presents the cost of providing file and directory view consistency. The costs were measured separately for the file and the directory operations.

Attribute Consistency View consistency is optionally provided for file attributes in Ficus. This ensures that file attributes are as consistent as file data. It is optional because of the heavy cost of providing view consistency for attributes. There are several commonly used UNIX operations such as `find` and `ls -l` that invoke large numbers of attribute operations without opening any files. View consistency for attributes imposes a heavy overhead on these operations. Without attribute view consistency, users may see older attributes although the data version is view consistent.

A problem that occurs with file attributes in the later-version criterion is that attributes may be from a later version of the file than the file data that has been accessed. This is also possible in a single copy system. For example, a user reads a file and then reads the time at which the file has been updated. If another user updates the file between these two operations, then the file update time that is read by the first user is later than the file data that he has seen. In other words, file data and file attributes are effectively separate objects. This is normally accepted in most environments.

4.3 Local Replica Optimization

An optimization can be done in the view consistency algorithm for a centralized entity when a file replica is stored on the local host (a machine associated with the centralized entity). If a replica exists on a host, the default replica selection procedure ensures that entities on that host *always* access the local replica. If the host crashes, the entities either die (such as processes on the host), or don't function until the host comes up again (a user working from a portable). Thus entities never need to access remote data.

We assume that a replica does not undo the effects of an update that it has incorporated. Therefore, the version of a file increases monotonically at each replica. Since entities always access the local replica, they access later versions of the file, or *always* obtain view consistent data. Therefore the view-entries do not have to be accessed or kept at all. Thus, when a local replica exists, view consistency can be provided to centralized entities at no cost.

Replica Addition and Deletion The discussion above assumes that a replica is already present on the host at which the centralized entity resides. Some special actions must be taken when the local replica is added or dropped. Suppose a user decides to add a local replica of a file that he has been accessing remotely.[5] The local replica must be view consistent. This is not hard to ensure since the remote replica from which the data was copied is view consistent. After the local replica has been created, the view-entry can be removed from the database and all accesses directed to the local replica.

A local file replica deletion must record the version of the deleted data. The next access to a remote replica uses this version to ensure view consistency.

[5]Replication agents have also been built at Ficus that automatically add (or drop) replicas of files [Kue94].

CHAPTER 5

Replica Switching for Consistency and Improving Availability

A distributed system provides data consistency as well as highly available data. Chapter 4 describes the view consistency algorithm and several efficiency issues for a centralized entity implementation. However, it does *not* discuss replica selection and switching issues related to providing consistency in the function `switchToLaterReplica` shown in Figure 4.1. In this chapter, replica switching issues for improving availability and for providing view consistency are discussed. We will see that there are tradeoffs in providing high data availability and consistency at the same time. Normally, view consistency does not significantly reduce availability. Moreover, non-view consistent data is very confusing to the user. Therefore, we take the position that data must always be view consistent, even if it implies reducing availability.

Section 5.1 presents replica switching to improve availability. It does not take view consistency into account. There are three goals of switching for availability:

- Data is accessible as long as any replica is available.
- Data is accessed from the fastest available replica.
- The overheads of replica selection and switching are minimized.

Suppose the access latency and bandwidth to replicas A and B are roughly the same. If all accesses are being made from replica A, it may become overloaded and appear to be slower than replica B. Suppose all accesses are then switched to replica B. Replica B then behaves exactly the same as replica A. Thus, the system pays a high switching cost, and accesses switch back and forth between A and B without necessarily improving availability. Obviously, replica switching should avoid this ping-pong effect and its associated costs. Ideally, replica switching should occur when the availability benefits of switching to a faster replica outweigh the costs of selecting the fast replica and switching to it. This discussion does not take data consistency into account. Sometimes switching may not be possible because the faster replica is inconsistent. Section 5.2 describes replica switching for consistency and shows how it affects availability.

5.1 Switching to Improve Availability

Figure 5.1 shows the view consistency algorithm in a block diagram. The previous chapter discussed the `readViewEntry` and `writeViewEntry` modules. This chapter discusses the shaded modules that perform the replica selection and switching functions. The `switchToReplicaOnError` module was not shown in Figure 4.1 in Chapter 4. This module switches accesses to an available replica when the current replica becomes unavailable. Section 3.4, which presented the default replica selection policy, discusses this functionality briefly. The aim of the `switchToAnyReplica` module is to provide the fastest available replica. The `switchToLaterReplica` module is similar except that it provides view consistency also. It is discussed in the next section.

The `switchToAnyReplica` module is misnamed. Although it allows switching to *any* replica (as opposed to only a *later* replica), its main purpose is to select

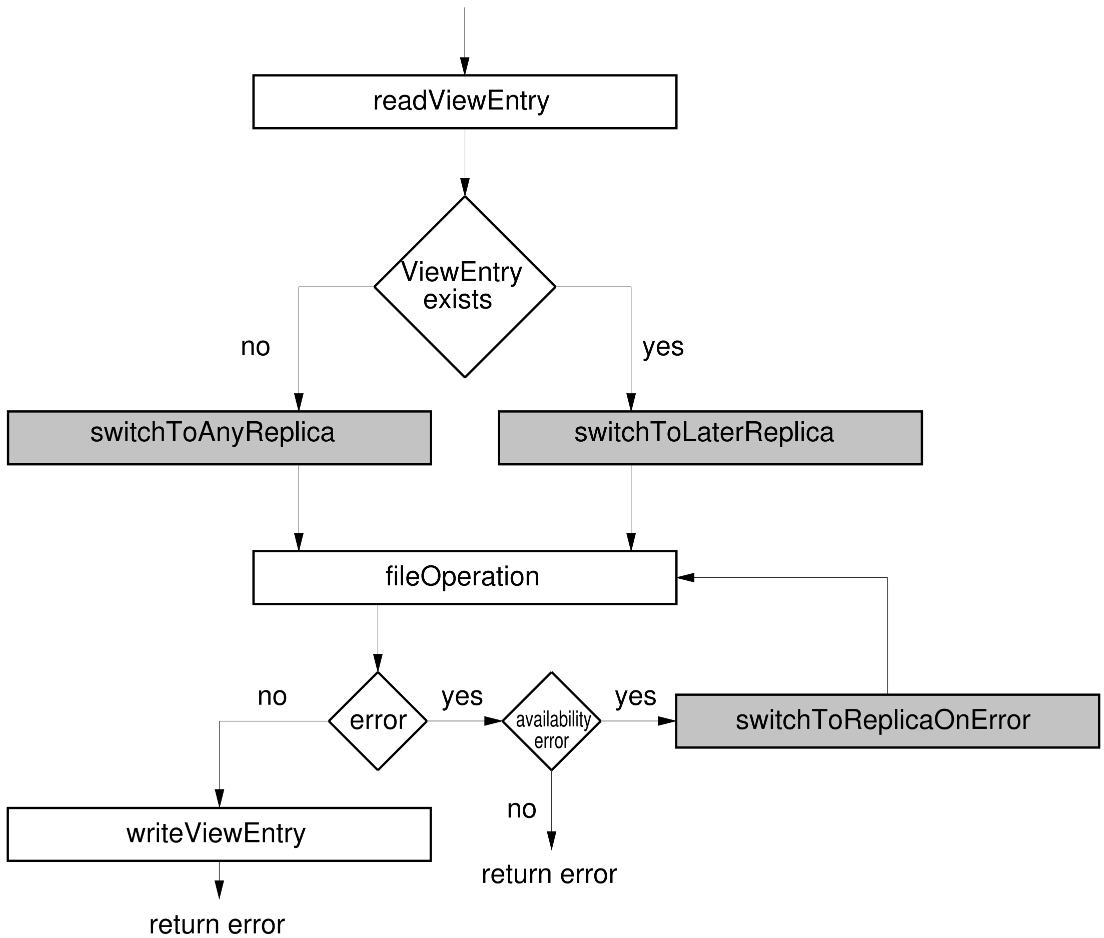

Figure 5.1: A block diagram of the view consistency algorithm. Replica switching is done in the shaded modules.

and switch to a highly available replica. It selects a highly available replica using two criteria as shown in Figure 5.2. The first criterion, *accessibility*, selects a replica that has a high probability of being available. Such a replica is called an *up-replica*. The second criterion, *optimality*, selects the fastest available replica, called the *fast-replica*. The precise definition of a fast-replica is given later. The choice of a highly available replica is made using only one criterion at a time, i.e., either the up-replica or the fast-replica is chosen (also, neither may be chosen).

It may seem that the fastest available replica should also be highly available and thus the optimality criterion should include the accessibility criterion. However, a distinction is made between the two criteria because they are used under different conditions. The accessibility criterion is used for short-term availability. When the replica from which data was being accessed becomes unavailable, accessibility ensures that data is provided from an up-replica. Therefore, the up-replica must be highly available in the short-term failure mode of the system. The optimality criterion is used for long-term availability. If the current replica that is being accessed is relatively "slow," optimality selects and switches accesses to the fast-replica. This switching is performed for overall long-term throughput and efficiency, and not for failure correction. Thus the fast-replica may not necessarily be highly available in the short term period. Each criterion optimizes for a different set of operating conditions. If the two criteria are combined into a single one, the system will not be as effective. The next two sections describe the two availability criteria in detail.

5.1.1 The Accessibility Criterion

The accessibility criterion accomplishes the first goal of switching for availability, i.e., providing data as long as any replica is available. There are two steps in the process:

- Notice that a replica has become unavailable.
- Pick a replica that is available (the up-replica) and switch to this replica at the appropriate time.

Nothing needs to be done as long as a replica is being successfully accessed. The `switchToReplicaOnError` module notices that the current replica has become unavailable when a file operation returns with an availability error. The current replica is marked a *down-replica* (a replica that has a high probability of *not* being available). An up-replica must be chosen at this time. There is no replica that is known to be highly available. Thus the replicas are tried in order until an access to a replica is successful. The access order depends on several factors such as knowledge of an up-replica (if it had been chosen earlier and is not the current down-replica) and replica access times that are determined for the optimality criterion. The chosen replica is marked the new up-replica and the file operation is attempted with this new replica. Note that the down-replica cache has just one entry per volume. A replica is not marked a down-replica, even if it is not available when different replicas are tried. This would mask the original (current) replica that was put in the down-replica cache.

When a file operation fails, the `switchToReplicaOnError` module switches accesses to an available replica, and sets an up-replica and a down-replica. Presumably other files in the volume have also been accessing the down-replica. Accessibility disallows any further accesses to the down-replica. The `switchToAnyReplica` module uses the accessibility criterion at this time as shown in Figure 5.2. If a down-replica is going to be accessed for some other file in the volume, it switches the file operation to the up-replica. The switching is done aggressively before the

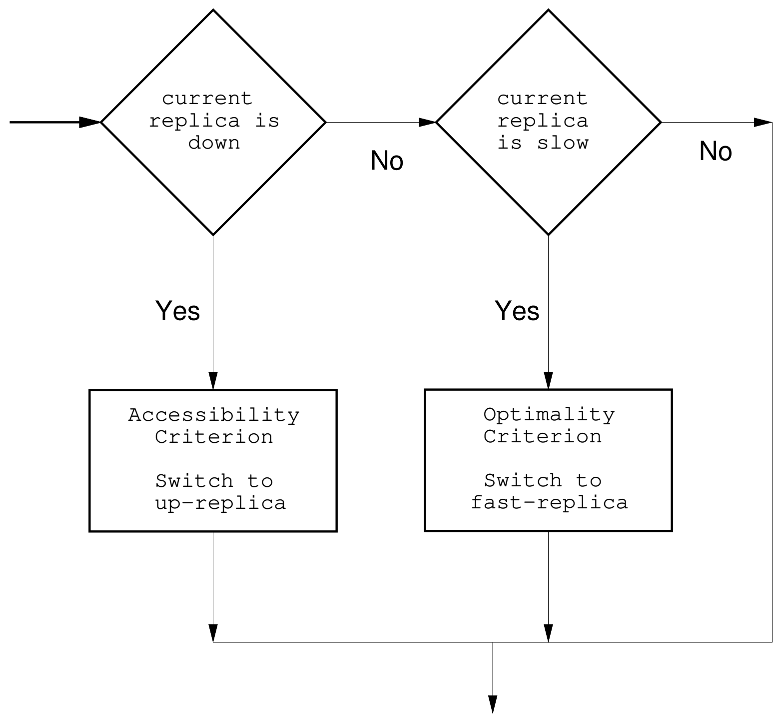

Figure 5.2: The **`switchToAnyReplica`** module switches to a highly available replica based on the *accessibility* and the *optimality* criterion. The **`switchToReplicaOnError`** module marks an unavailable replica as being a down-replica and finds an up-replica. An external availability server finds a long-term fast-replica. A replica that is *switching-factor* times slower than the fast-replica is considered a slow-replica.

file operation can commence.

We had mentioned earlier that the accessibility criterion is used for short term availability. A replica cannot be considered an up-replica if it has not been accessed for a long period. Similarly, a replica is not considered a down-replica for long periods, since machines generally go down for short periods. For example, the replica that is typically the fastest available replica may currently be down. We do not want to disallow access to it forever. Thus there is a user-changeable timeout (default value is 5 minutes) after which the down-replica cache is cleared. If the down-replica is still unavailable after the timeout period, the next attempted access to it will put it back in the down-replica cache.

5.1.2 The Optimality Criterion

The optimality criterion accomplishes the second goal of switching for availability, i.e., providing data from the fastest replica. This improves the overall long-term throughput of the system.

The definition of the fastest available replica has not yet been made clear. Is it a replica with the highest bandwidth connection to the client accessing the data, or is it a replica with the lowest latency connection? The fastest available replica may have to take both these factors into account. For example, a high bandwidth connection is preferable for large file transfers, while a low latency connection is useful for small file transfers. We assume that both these factors (and maybe others) can be combined into a single *availability value*, and the replica with the smallest positive such value is the fastest available replica. An interface and a default method for calculating the availability values is provided in Ficus, and is discussed below. It allows the user to easily modify the availability values depending on the system requirements.

The `switchToAnyReplica` module uses the accessibility criterion when the replica that is going to be accessed is a down-replica. Otherwise the optimality criterion is tried (Figure 5.2). If the availability value of the current replica is greater than *switching factor* times the availability value of the fastest replica (or the current replica is switching factor slower than the fastest replica), switching to the fastest replica is attempted. If the switching factor is close to one, replica switching occurs when the current replica is slower than the fastest one. This ensures that the fastest replica is accessed all the time. Unfortunately, a slight (and perhaps temporary) change in the availability value can induce switching between replicas. If this occurs frequently, it adds excessive switching costs to the system. Thus the value of the switching factor should take bandwidth and switching costs into account. In Ficus, the switching factor is set to 2. Performance figures in Chapter 7 suggest that this factor can be set to a smaller value since the cost of switching is very low in our system.

If switching to the fastest replica is successful, nothing else has to be done. However, if the fastest replica is not available, then it is marked as a down-replica and its availability value set to an undefined value of zero. An undefined availability value signifies that either the replica availability has not been measured or the replica is unavailable. Switching (for optimality) to such a replica is never attempted.

Maintaining Availability Values The availability values must be obtained before they can be used to determine and switch to the fastest available replica. Note that availability values are *relative values* since they are only compared among themselves. The following should be kept in mind while calculating availability values:

- Availability values should take their past values into account since sudden fluctuations in replica availability should not affect the values severely.

- The calculation of the values can be relatively complex because it does not have to be done in the critical path of a file operation.

- The values should be calculated for each replica in the same way as much as possible.

- The values should be calculated with the same frequency for each replica. This ensures that the availability values for different replicas are equally accurate or reliable.

- Increasing the calculation frequency improves the accuracy of the availability values, but increases the calculation overheads also.

- Different types of file operations should be used for calculating the values, and they should be weighted in proportion to their use during real system operation. A single operation may emphasize a particular aspect of availability (such as latency).

- The interface should allow the user to easily modify the values for individual systems.

The availability calculation code attempts to address these issues. Many of these issues suggest that a separate process (and not the kernel) should be used to calculate the availability values. This availability daemon calculates the values independent of processes in the kernel and can be made as complex as required. Also, it can easily be made user changeable.

Availability values are calculated in two steps. First the *current availability value* (CAV) of a replica is determined based on the round trip times of `open`, `read` and `write` since these form a representative set of file operations. The current availability value is calculated every ten minutes, and is an estimate of the instantaneous replica quality. It takes both the bandwidth and the latency into account. The replica bandwidth is determined by invoking `read` and `write` operations on a relatively large-sized file,[1] while replica latency is measured by invoking `lookup` on a zero-length file. The file operations are done several times to get better averages.

After CAV has been determined, a smoothed (or mean) availability value (SAV) is calculated based on CAV and its past values. The smoothed availability value is an estimate of the long-term average quality of a replica. The kernel uses the smoothed availability value to determine the fast-replica. The calculation of the SAV uses exponential means. Therefore SAV depends on the previous SAV, the current availability value (CAV) and a constant reactivity factor Rf that lies between 0 and 1. The relationship can be expressed with the following formula:

$$SAV = SAV * Rf + CAV * (1 - Rf)$$

This mean is appropriate for most systems because it captures past values reasonably well and requires little space or time overhead. Unfortunately, the exponential mean depends very heavily on the choice of the reactivity factor. The mean responds to the current value sharply when the reactivity value is small, but becomes unresponsive when it is large. Instead of a single reactivity factor, we use two reactivity factors (with values 0.8 and 0.66) for calculating the mean. The smaller factor is used when $CAV > SAV$, and the larger factor is used otherwise.

[1] Since we do not want to store a large file in each volume, a standard system file that is stored at each site is used. This assumes that the access times to different volume replicas at the same site are roughly the same.

This ensures that the mean availability value responds quickly when the current availability value is larger than the mean. Otherwise the mean is not affected as much. Studies have shown [KP87] that this mean is sufficiently responsive to the current value (especially when it is large) and yet not too responsive.

A new vnode operation has been implemented that transfers the availability value for each replica in the volume from the daemon process to the kernel. The kernel uses these values to update its view. If the kernel knows that a new replica of a volume has been added or deleted, or volumes have been dynamically mounted or unmounted, the return status of the vnode operation indicates that the daemon should reread the volume information. The addition of the new vnode operation is simple in the Ficus stackable layers architecture.

Note that the kernel only reads the availability values. It is solely up to the daemon to generate these values. However, there is one case in which the kernel modifies these values. The kernel resets the availability value to be undefined when it tries to switch to the fastest available replica, and the fast-replica is not available. This ensures that the kernel does not keep trying to switch to this replica. The kernel may learn about sudden changes in the quality of a replica before the daemon notices the change (especially when availability decreases suddenly). However, only one process should determine the availability values, or else the different values may not be comparable because of the different overheads involved in the calculation. For example, the total time to perform a read operation as measured within the kernel will be less than the time measured at the user level. Thus, in the case discussed above, the kernel sets the availability value to be undefined, and does not use it for any further comparison until this value is set by the daemon again.

5.2 Switching for Consistency

The `switchToReplicaOnError` module shown in Figure 5.1 provides available replicas when the current replica is down. Similarly, the `switchToAnyReplica` module provides both available replicas (when the current replica is unavailable) as well as fast replicas when the current replica is slow. Both these modules increase data availability, but do not provide any view consistency guarantees. They are useful either for data that does not require consistency guarantees, or for files that do not have view-entries (files that are being opened for the first time). In this section, modifications to these modules for supporting view consistency are described. We also show how these changes affect availability.

Only a minor change is necessary in `switchToReplicaOnError` to support view consistency. Recall that when a replica fails, the replicas are tried until an available replica is found. Accesses are then switched from the failed replica to the available replica. An entity has a view-entry for each file that it has accessed in the past.[2] For view consistency, the available replica's data version must be later than the version in the view-entry of the file. Since Ficus stores replica versions as vector timestamps, this comparison is done in exactly the same way as described in Section 3.2. If the replica's data version is not later than the view-entry version, further accesses cannot be made from this replica. Thus replicas must be tried until a replica that is both available and view consistent is found. This replica can replace the failed replica. Note the reduction in availability; an older replica is not provided although it may be available.

Since major modifications are needed in `switchToAnyReplica` for supporting view consistency, a separate `switchToLaterReplica` module handles files that

[2]Unless the view-entry has been garbage collected, in which case the file can be treated as if it has never been opened (described in Chapter 6).

have an associated view-entry. Several issues must be considered before building this module. While the `switchToReplicaOnError` module is invoked only after an availability failure, the `switchToLaterReplica` module lies in the critical path of most file operations. Thus its overheads should be low. Moreover, it must always provide view consistent data. Finally, within the constraints of consistency, it must try to provide available as well as optimal replicas.

From the consistency standpoint, the common case in `switchToLaterReplica` has low overhead. The data is view consistent as long as it is accessed from the same replica because this data is either the same as or newer than the data version last seen. However, this data replica may be a slow replica. In that case, it may be beneficial to switch to a faster replica. For example, suppose a replica that is nearby goes down, and a distant replica starts being accessed. After some time the nearby replica comes back up. Although accesses to the distant replica are view consistent, there are availability advantages of switching to the closer replica.

The `switchToLaterReplica` module is shown in Figure 5.3. There are four decision functions and three sub-modules A, B and C that perform the switching tasks. The common path is shown with thick lines. In this path, the current replica, or the replica that is going to be accessed, is the same as the replica in the view-entry and thus is view consistent. Moreover, this replica is neither down nor slow, and therefore it is highly available. No replica switching is required in this case, and the decision functions 1 and 4 can be executed quickly.

Suppose the current replica is not the same as the view-entry replica. This can happen, for example, if the `lookup` returns a file handle for a replica that is different from the view-entry replica.[3] If the view-entry replica is not slow, the

[3] Since the view-entries contain the file handle rather than the file name, the `lookup` has to be done before getting the view-entry for the file. Thus the replica in the view-entry can be different from the replica returned by the lookup.

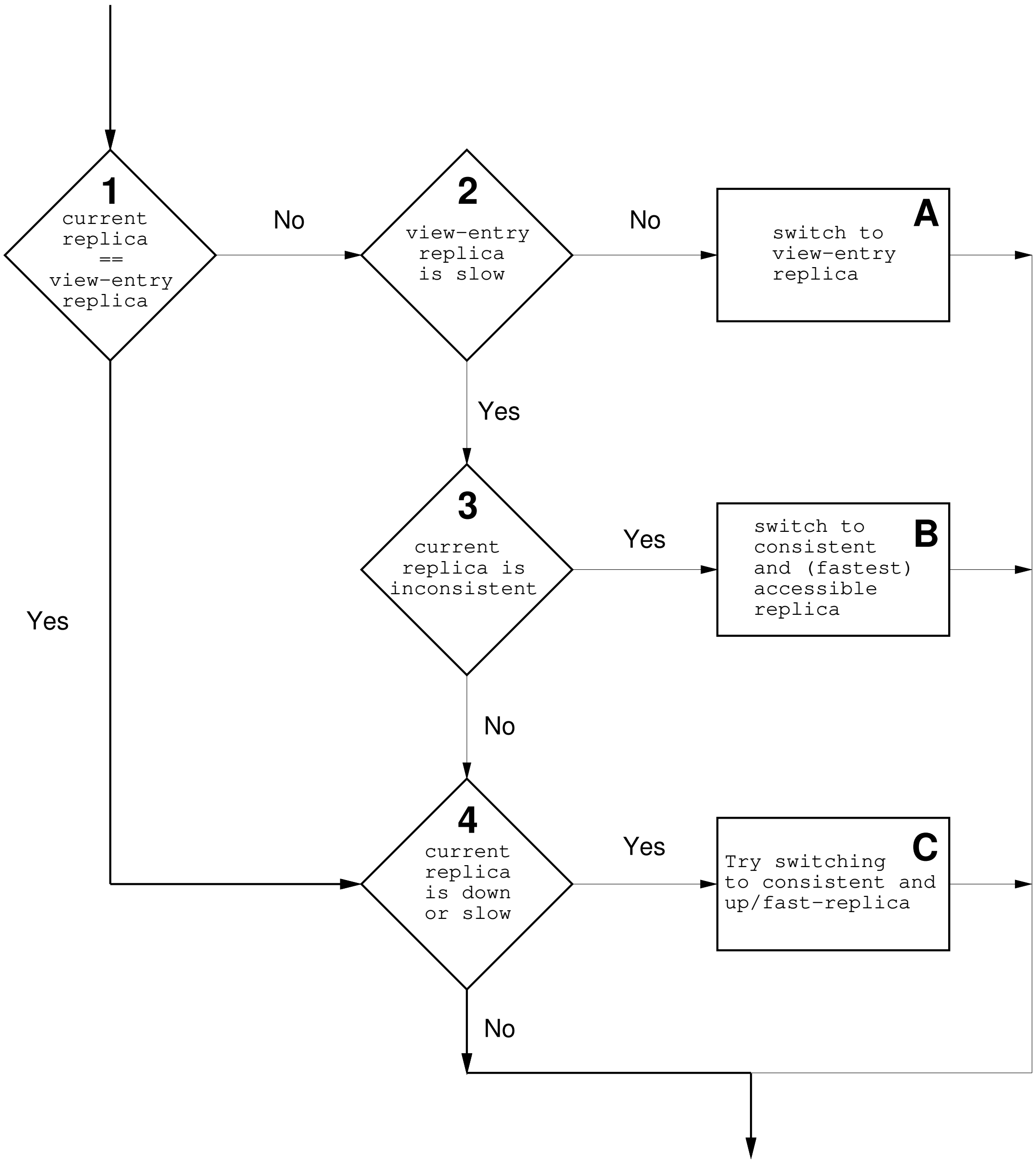

Figure 5.3: The `switchToLaterReplica` module

current replica is switched to the view-entry replica. This is the first switching step that is tried because the view-entry replica is known to be view consistent.

If the switching is unsuccessful, or the view-entry replica is very slow, the current replica is checked for consistency. The current replica's version is obtained now if it has not been obtained earlier. Since the current replica may be stored at a remote site, this operation can be expensive.

If the current replica is not view consistent (it is older than the version in the view-entry), the sub-module B is invoked. This module does an exhaustive search of all replicas (in availability value order) to find another consistent replica. Note that module B is very similar to the `switchToReplicaOnError` module (that has been modified for view consistency as discussed above). The only difference is that while `switchToReplicaOnError` is called on an availability failure, module B is invoked on a consistency failure. However, from the consistency standpoint, both these errors are very similar since they both prohibit access to data.

Finally, if the current replica is either down or slow, switching to the up-replica or the fast-replica is attempted. The decision function 4 and module C together are similar to the `switchToAnyReplica` module. The only difference is that the up-replica or the fast-replica must be checked for consistency in sub-module C. If the fast-replica is available but not consistent, it is marked as being inconsistent. Switching to an inconsistent fast-replica is not attempted for a certain period (default 5 minutes). It is assumed that if a single file at a particular replica is inconsistent, other files at that replica (in the same volume) are also inconsistent. If this (early) assumption is not supported by future experience, we will make appropriate modifications to our current switching algorithm.

Each of the sub-modules returns a consistent replica. Preference is given to the view-entry replica (unless it is much slower than the fast-replica) because it is known to be view consistent, and therefore switching to it does not require an exhaustive search of all replicas. The difference between sub-modules B and C is that while the former does an exhaustive search, the latter tries to switch to known replicas. Moreover, an error returned by B implies that data is unavailable, whereas the current replica is still usable if there are switching errors in C.

CHAPTER 6

View-Entry Garbage Collection

View-entries are kept for each file that has been accessed by an entity as described in Chapter 4. Each entity has a logically separate view-entry database that contains these view-entries. This view-entry database is stored at the client site associated with the entity.

The view-entries for persistent entities must be stored persistently because these entities can exist for long periods of time, across login sessions and machine reboots. The database for transient entities may also have to be stored on disk if its size becomes very large. Since view-entries are kept for each accessed file, their number grows over time. These view-entries must be garbage collected or deleted so that the database size does not increase indefinitely. The database for transient entities can be entirely removed when the entity terminates. A different criterion is needed to delete the view-entries of persistent entities.

View-entry deletion can be done by using the following rule: the view-entry is removable when all the file replica versions are later than (newer than or equal to) the file version in the view-entry. The view-entry determines the replicas that satisfy the consistency criterion. Since all the replicas satisfy the later version criterion, the view-entry is not required anymore. Any replica that is next accessed will yield a later version. This idea is used in the design of the view-entry deletion algorithm.

Recall from Chapter 4 that a view-entry for a file contains the version and the replica of the file that was last accessed. The view-entry version determines whether the next access satisfies the consistency criterion. In Ficus, a file version is a vector timestamp.[1] The view-entry also contains the last access time. This will be used for the garbage collection algorithm. Thus, a view-entry has the form (f, v_f, s, t_f), where v_f is the last version of the file f accessed at time t_f from site s.[2]

6.1 Simple View-Entry Deletion Algorithm

A simple view-entry deletion algorithm can obtain the current version of each of the file replicas. If all the replicas have versions that are later than the view-entry version, the view-entry can be deleted. Unfortunately, this algorithm has two problems. All replicas may not be simultaneously available. In mobile computing environments, replicas may not frequently communicate with each other. When several mobile replicas exist, view-entries may never be deleted. Moreover, this deletion mechanism is very expensive. All the file replicas must be consulted on the fly to delete a view-entry.

This simple algorithm obtains all the file replica versions and then deletes a view-entry based on this version information. The version acquisition operation can be separated from the deletion operation. This would solve the problems mentioned above. The replica versions can be obtained when replicas communicate with each other. The view-entry deletion algorithm can use this information, at some later time, to decide which entries can be deleted. A deletion algorithm that separates these two operations in presented below.

---

[1] A vector timestamp can be newer, older than, or incomparable (conflicting) with another vector timestamp.

[2] The terms *site* and *replica* are used interchangeably.

6.2 Deletion Algorithm

For view-entry deletion, every replica must have a version that is later than the view-entry version. This information is distributed across all replicas. The task is to collect this information at each replica. Recall that a version is a vector timestamp of length n, where n is the number of replicas of a file. If each replica version is acquired and stored at each site, the storage requirements will be n versions at each site, or $O(n^2)$ values per file at each site. This storage requirement is prohibitive.

An equivalent deletion criterion that leads to a more efficient version acquisition algorithm is the following: the view-entry can be deleted when the view-entry replica has propagated the view-entry version to all other replicas. Note that for this criterion, a site must learn what other sites know about itself, i.e., have the other sites seen a file version that it had stored (in the past)? We call such information an *acknowledgment*. Suppose A^i denotes an acknowledgment timestamp that is stored at site s_i. It is a vector timestamp, and its j^{th} component (or $A^i[j]$) states that site s_i knows that site s_j has learned about every version that site s_i had until time $A^i[j]$. Each site that stores a replica stores such an acknowledgment timestamp. The deletion algorithm uses this acknowledgment information to decide whether a view-entry can be deleted. The deletion algorithm is described here. The following sections present efficient algorithms for obtaining acknowledgment information.

Consider the view-entry (f, v_f, s_i, t_f) on some client site, where v_f is the last version of the file f accessed at time t_f from site s_i. If $t_f \leq \min_i(A^i[j])$, site s_i knows that all other sites must have seen the version v_f of the file f, or all sites have a version later than v_f. This is the view-entry deletion criterion. In other words, the client can obtain the minimum component of A^i from site s_i (the site from which it last accessed the file) and delete the view-entry if this value is greater than t_f.

Figure 6.1 shows the algorithm for deleting view-entries. The deletion algorithm uses the acknowledgment timestamp which is stored at the site from which the file was last accessed. The timestamp is used to determine if the entry can be deleted.

This algorithm solves both the problems of the previous deletion algorithm. The view-entry deletion information (or the acknowledgment timestamp) is acquired independently of the view-entry deletion process. Thus, all replicas do not have to be available during deletion. The previous algorithm accessed all the replicas for the version information during the deletion process. This algorithm accesses only one replica during deletion.

The acknowledgment information states what a site knows that others have learned about itself. Obtaining acknowledgment information requires propagating file and version information between replicas. This file and its version propagation are performed by the reconciliation process in Ficus as described in Section 3.2 earlier. The basic reconciliation algorithm is presented below since the acknowledgment algorithm uses this algorithm.

6.3 Reconciliation Algorithm

Reconciliation propagates file replica data and version information between pairs of sites. It occurs at a *volume* granularity for ease of replica maintenance and efficiency reasons. Let a volume consist of a set of files that are replicated at sites $s_1, s_2, \ldots, s_n$. When site s_j reconciles with site s_i, it gets updates from s_i. These updates may have been generated at any site. An update generated at site s_k at

```
//
// Returns true if entry can be deleted.
//
bool
deleteableViewEntry(entry)
{
    bool deleteEntry = False;

    (f, v_f, s_i, t_f) = entry;
    if (s_i ≠ localMachine) {
        // Go to the machine that stores replica s_i
        // to check if the entry can be deleted.
        On (site s_i) {
            deleteEntry = deleteableViewEntry(entry);
        }
    } else {
        if (t_f ≤ min_j(A^i[j])) {
            deleteEntry = True;
        }
    }
    return deleteEntry;
}
```

Figure 6.1: Algorithm for deleting view-entries

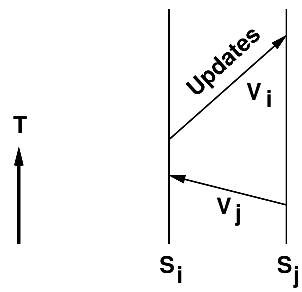

Figure 6.2: File data and version propagation in the reconciliation algorithm in real time

time t is denoted by $U^k(t)$ (or by $U^k_k(t)$). When this update has been propagated to site s_i, it is denoted by $U^k_i(t)$. An efficient propagation mechanism transmits those updates that s_i has generated or incorporated (from other sites) but s_j has *not* yet received.

Let site s_j reconcile with s_i successfully at time $V^j[i]$ (more accurately, $V^j[i]$ is the starting time of reconciliation). The significance of $V^j[i]$ is the following: after site s_j successfully reconciles the volume, each file in the volume at s_j has a version that is the same as or newer than (later than) the file version that existed at s_i until time $V^j[i]$. Site s_j records the last successful reconciliation times for all replicas in the n component reconciliation vector V^j. Thus, V^j records what site s_j has learned about others. $V^j[i]$, the i^{th} component of the reconciliation vector V^j indicates that site s_j knows about all versions that site s_i has seen until this time.

Figure 6.2 shows the reconciliation algorithm in real time. The reconciliation vector V^j is sent from site s_j to site s_i. This vector (that describes what site s_j knows about others) is used by site s_i to efficiently transmit file data that site s_i has seen but site s_j has not.

$s_i \rightarrow s_j$

1. Update Propagation:

At site s_i :

For each site s_k

if $(V^i[k] > V^j[k])$ then (A)

For each update $U_i^k(t)$

// Update is generated at site s_k at time t

// and present at s_i

if (t> $V^j[k]$) then (B)

Transmit $U_i^k(t)$

At site s_j :

Receive updates from s_i and incorporate them

2. Timestamp Propagation:

At site s_i :

Transmit reconciliation-vector timestamp V^i

At site s_j :

$V^j = \max(V^i, V^j)$

Figure 6.3: The reconciliation algorithm

The reconciliation algorithm shown in Figure 6.3 propagates information from s_i to s_j. The reconciliation vector V^j is assumed to be at site s_i already. In the update-propagation step, updates at site s_i that are generated at some site s_k (or $U_i^k(t)$) are sent to site s_j if the update has not been seen by site s_j (step B). As an optimization, updates generated at site s_k do not have to be considered (or sent) when s_i does not know more than s_j about s_k as shown in step A. The reconciliation-vector timestamp V^i is also sent to site s_j. It contains timestamps at the start of update propagation. The timestamp V^j is updated to be the item-wise maximum of V^i and itself at site s_j. This step ensures that s_j knows about all versions that s_k had seen until the timestamp $V^j[k]$. The acknowledgment algorithm uses this reconciliation-vector timestamp information.

6.4 Acknowledgment Algorithm

The timestamp information in the reconciliation algorithm tells a site about the file versions that it has received from other sites. Thus, site s_j knows whatever site s_i knew until the timestamp $V^j[i]$. Similarly site s_i knows whatever site s_j knew until the timestamp $V^i[j]$. Recall from Section 6.2 that we need acknowledgment timestamp information, or the vector A^i at site s_i for view-entry deletion. The acknowledgment-vector timestamp A^i records what site s_i has learned about what other sites know about itself. A client can delete a view-entry for a file that was last accessed from site s_i once site s_i has learned that the file version has reached all other sites. The reconciliation algorithm by itself does not provide acknowledgment information. In particular, acknowledgments require that s_i learn about the timestamp $V^j[i]$ from each site s_j, since s_j knows about s_i until this time. Unfortunately this timestamp information is needed at s_i, but is spread across all replicas.

If pairwise communication is guaranteed between every pair of sites, then s_i can obtain $V^j[i]$ from s_j while communicating with it the next time. The acknowledgment-vector timestamp $A^i[j]$ can then be updated to be the value of $V^j[i]$. The significance of the acknowledgment timestamp for view-entry deletion, as discussed above, is that site s_i knows that every other site has learned about any version that it knew until $\min_j(A^i[j])$. Thus, a client can delete a view-entry that contains any such version.

Direct Communication The acknowledgment algorithm as presented here is a minor variation of the reconciliation algorithm. Note (from the previous paragraph) that when site s_i reconciles with site s_j, only the j^{th} component of A^i (or $A^i[j]$) is incremented.[3] Conversely, the acknowledgment time $A^i[j]$ at site s_i increases only if site s_i reconciles with s_j. Thus, the forward progress of the acknowledgment algorithm needs each replica to periodically communicate with all the others.

In a mobile replicated system, this is a very strong requirement. The progress of the mutual consistency algorithm should only require indirect communication. For example, two mobile sites may communicate with a stationary site at different times and thus never come in contact with each other directly. The acknowledgment algorithm must allow information between the two mobile sites to be propagated through the stationary site. The reconciliation algorithm that is shown in Figure 6.3 allows such indirect communication. The reconciliation-vector timestamp $V^i[j]$ at site s_i can increase even though site s_i may never directly communicate with site s_j. We would like the acknowledgment algorithm (that increments A^i) to support indirect communication also.

[3] This is unlike the reconciliation-vector timestamp V^i, where each component can get incremented.

6.5 Indirect Communication Acknowledgment Algorithm

Gossiping allows indirect communication between replicas and can be used to update the acknowledgment-vector timestamp. Consider the reconciliation-vector timestamp $V^j[i]$ at s_j. Site s_i must learn about $V^j[i]$ in order to update the j^{th} component of its acknowledgment vector A^i. Since site s_i and s_j are not guaranteed to communicate, this timestamp must also be stored at other sites and propagated indirectly. Since no pair of sites are guaranteed to communicate directly, site s_j must send $V^j[i]$ to all sites, so that the timestamp eventually reaches site s_i. Site s_j does the same for every site. Thus, it gossips its reconciliation-vector timestamp V^j to all sites. Other sites store a copy of the timestamp V^j. Each site stores a copy for every other site and thus must store n vector timestamps. Thus, the indirect communication acknowledgment problem requires a matrix timestamp. Effectively, a site stores a copy of what each site knows about others and thus what each site knows about itself (the acknowledgment information).

The following notation is used in the figure shown below:

1. $M^i[j,k]$: The matrix at s_i is M^i. $M^i[j,k]$ is the $(j,k)^{th}$ element of matrix M^i. It states that s_i knows that s_j has learned about all updates from s_k until time $M^i[j,k]$.

2. $M^i[j]$: The j^{th} row of M^i. It records what site s_i knows about the updates that site s_j has seen. This value is a copy of V^j (or what site s_j has learned about others). Its value is always less than or equal to the current value of V^j.

3. V^i: The reconciliation-vector timestamp at s_i is the i^{th} row of M^i or $M^i[i]$.

$s_i \rightarrow s_j$

1. Update Propagation:

   Same as in Figure 6.3.

2. Timestamp Propagation:

At site s_i :

Transmit matrix timestamp M^i

At site s_j :

(Vector propagation) $V^j = \max(V^i, V^j)$

(Matrix propagation) $M^j = \max(M^i, M^j)$

Figure 6.4: The indirect communication acknowledgment algorithm

The acknowledgment-vector timestamp A^i at s_i now becomes the i^{th} column of the matrix timestamp M^i. Elements in this column record what site s_i knows that others have learned about it. Thus, if a file was last accessed from site s_i before $\min_j(M^i[j,i])$, or before $\min_j(A^i[j])$, then s_i knows that the version of the file last accessed has reached all other sites and the view-entry can be removed.

The acknowledgment algorithm is combined with the reconciliation algorithm and presented in Figure 6.4. The only new step in this combined algorithm is the acknowledgment matrix propagation step. This step is similar to the reconciliation-vector timestamp propagation step. Consider the k^{th} row of the matrix M^j, or $M^j[k]$. It records what site s_j knows about the updates that s_k has learned. Taking the maximum of the matrices in this step ensures that site s_j learns as much as site s_i knows about the updates that others have seen. This algorithm updates the acknowledgment information (A^i at s_i) using gossiping. Thus, $A^i[j]$ (or $M^i[j,i]$) can increase even if site s_i never communicates with site s_j.

The algorithm requires storage for an n^2 matrix per site (where n is the number of sites and each entry of the matrix is a four-byte integer) and communication of the n^2 matrix for each pairwise interaction. However, each pairwise interaction involves all the files in a volume. Normally there are 100-10000 files in a volume. Thus, the overhead of matrix timestamp storage and propagation is not necessarily large.

6.6 Local Timestamps

Global timestamps cannot be assumed since the replicas are stored on a distributed set of machines. We use local timestamps of each machine (assumed to be monotonically increasing) in the deletion, reconciliation and the acknowledgment algorithm. These timestamps can be meaningfully compared only if they are issued from the same machine.

When a file is accessed from site s_i, the timestamp t_f in the view-entry (f, v_f, s_i, t_f) is obtained from site s_i. Consider the timestamp $V^j[i]$ in Figure 6.3. The timestamp value is generated at site s_i and stored at site s_j in the timestamp-propagation step. The acknowledgment timestamp $A^i[j]$ is a copy of an older value of $V^j[i]$ and thus is also generated at site s_i. Similarly all values in A^i are generated at site s_i. In Figure 6.1, the timestamp t_f is compared with the acknowledgment timestamps in A^i for view-entry deletion. All these timestamps are generated on site s_i, so their comparison is meaningful. Similarly, the timestamps that are being compared in the reconciliation algorithm and the acknowledgment algorithm can be shown to be generated on the same machine.

The timestamp t_f is *not* used to decide whether a replica meets the consistency criterion. Since t_f is generated on the replica that was last accessed, this timestamp cannot be compared with a file version on some other replica. We use the vector timestamp v_f for comparing replica versions. Thus, view-entry deletion and version comparison (for checking the consistency criterion) use two different timestamps. The deletion algorithm requires a separate timestamp because version vectors are file-specific, whereas the deletion timestamp is compared with a volume-wide acknowledgment timestamp.

6.7 View-Entry Deletion Conclusions

View-entries for all the files that have been seen by an entity are stored in a database for view consistent operation. This database grows over time and must be truncated so that its size does not increase indefinitely.

View-entries can be deleted when all the file replica versions are later than the view-entry version. A simple view-entry deletion algorithm can obtain the versions of each of the file replicas and delete the view-entry if all the replicas have versions that are later than the view-entry version. This is inefficient because all replicas must be contacted for the version information. Moreover, it requires all the replicas to be available simultaneously.

Instead, the version information can be obtained independently of the deletion process. This is done at a volume granularity. Each site learns about the file versions it has seen from others (the timestamp propagation step in the reconciliation algorithm) and what others know about itself (the acknowledgment algorithm). A client can remove a view-entry for a file (the deletion algorithm), if the site from which the file was last accessed knows that the version last accessed has been seen by all the other sites. This information is obtained from the acknowledgment algorithm.

CHAPTER 7

Performance Measurements

This work has two main components: supporting view consistency, and providing highly available and quickly accessible replicas. Both the components have been implemented on Ficus, a SunOS 4.1.1-based kernel, and have been running at UCLA since October 1995. The implementation uses a stackable layers framework [HP94] within the kernel, as described in Section 3.1, and a user-level view-entry database server. A deletion server obtains the acknowledgment information from the reconciliation (and acknowledgment) process and uses it to delete view-entries from the database. Finally, replica switching for higher availability is done within the kernel, with the help of an availability server that determines the availability values.

The performance of the system depends on several factors, such as the type of the database being used for storing the view-entries, effectiveness of view-entry caching within the kernel, size of view-entries, cost of replica switching, and the cost of determining the best available replica and deciding when to switch to it. The effectiveness of view-entry deletion depends on the policy for choosing pairs of replicas for reconciliation so that the acknowledgment information rapidly propagates between replicas, leading to quick view-entry deletion. These factors can be grouped into view consistency performance, replica switching for availability measurements, and view-entry database deletion measurements.

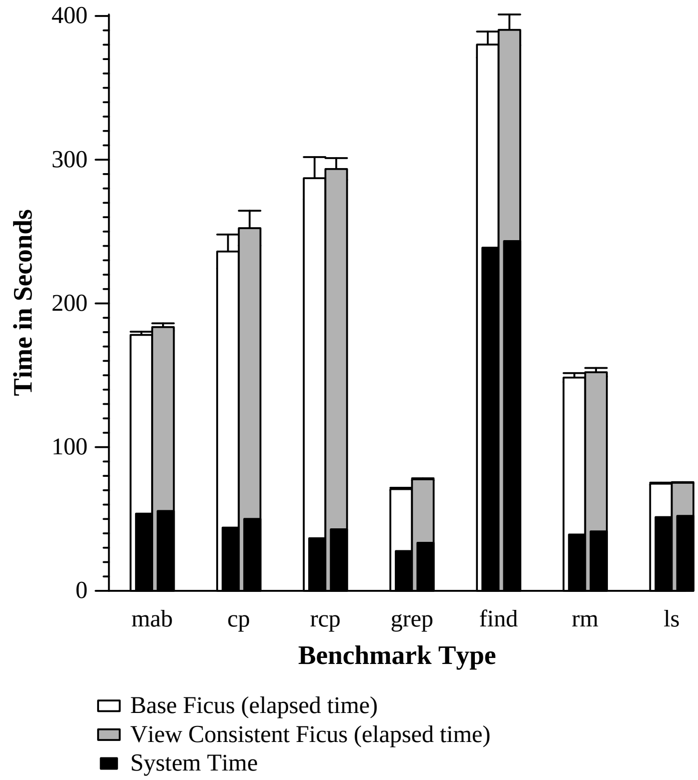

Figure 7.1: Local access times of base Ficus and view consistent Ficus

7.1 View Consistency Measurements

A careful assessment of the benefits of providing view consistent data involves measuring the reduction in the number of reads and writes to stale (not the latest) data versions when compared to a purely eventually consistent system. The costs of view consistency are the reduction in availability (again as compared to an eventually consistent system) and the system cost of providing view consistency. This assessment raises several problems. Although stale data writes (conflicting updates) are detected by using version vectors in Ficus, there is no easy mechanism for detecting *reads* to stale data versions. Moreover, very few conflicting updates are generated in current Ficus [RHR94]. This is partly because most machines that run Ficus are in a well connected LAN environment. This makes it very difficult to measure the reduction in the number of stale reads and writes, and also the reduction in availability. Therefore, only the system cost of providing view consistency has been measured. We hope to get better estimates of the benefits and costs of view consistency in a fully mobile and widely distributed environment in the future.

The cost of view consistency is measured by comparing view consistent and non-view consistent accesses. This is done for locally stored as well as for remotely stored data. All the measurements are taken on the same kernel while running the non-view consistent and view consistent code in different parts of the name space. Four Sun IPCs connected by a standard Ethernet connection within the Ficus LAN are used. SCSI disks are used for persistent storage, and each machine has 12 MB of main memory. The tests were performed during periods of minimal external network activity.

There are seven benchmarks that are used in the evaluation. The first test is the modified Andrew Benchmark (`mab`) [HKM88, Ous90] that is intended to model a normal mix of filing operations, and hence be representative of performance in actual use. The second and third tests are local and remote recursive `cp` and the fourth test is `grep`. Each of these tests exercise the read and write file operations. The fifth and sixth tests are `find` and `rm` programs that primarily execute recursive directory operations. The last test is the `ls` program that reads directory contents. The `mab` test is performed on a 1.3 MB tree. The `grep` and `ls` tests operate on `/usr/include/sys` that has 104 files containing 336KB of data. All other tests operate on the `/usr/include` hierarchy that contains 1311 files with 4.2 MB of data. Some of the tests were performed repeatedly to obtain measurable results.

Each of the benchmarks are done with one, two and three data replicas. The

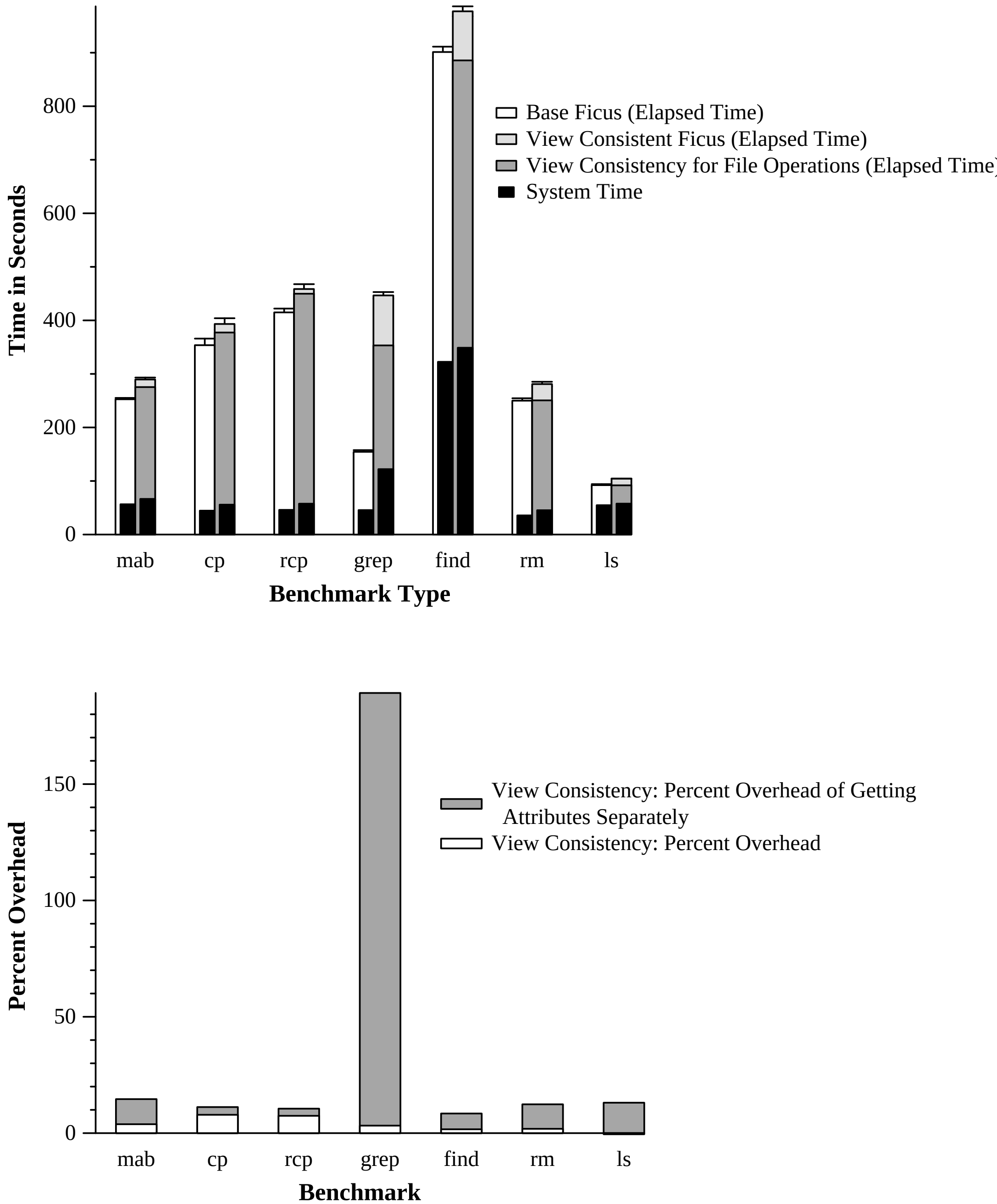

Figure 7.2: Remote access times of base Ficus and view consistent Ficus. The lower graph shows (in gray) the decrease in overhead when the view consistency attributes are obtained along with data.

results shown below are a mean of the results for different numbers of replicas. View consistency is provided for file and directory operations. It is not provided for attributes because we believe that the cost of providing attribute consistency can be prohibitive in UNIX systems. However, this is a performance issue and not an inherent weakness of the system.

Figure 7.1 shows a graph of the elapsed and system times of base Ficus and view consistent Ficus for *local* accesses. A local access is an access to a replica that is located on the local machine. We had presented the local replica optimization in Section 4.3 that eliminates all view consistency overhead for local accesses by centralized entities. We performed our experiments without the local replica optimization because we wanted to get a lower bound on the cost of providing view consistency for distributed entities, which cannot make use of the local replica optimization. The confidence intervals are only shown for the elapsed times. The view consistency overhead for most benchmarks is between 2 to 3 percent. The `ls` benchmark shows a smaller overhead because the cached view-entries are reused in successive iterations of the test. The `cp` and `grep` tests have a higher overhead of 8 percent because these programs perform many file opens. Recall that file opens may involve fetching view-entries from the disk (via the database server) unless they are cached in the kernel, and this contributes to the overhead. `Find` and `rm` do not show similar overheads because in these cases attribute and directory operations predominate, and we do not provide attribute consistency by default.

The upper graph in Figure 7.2 shows the elapsed and system times of base Ficus and view consistent Ficus for *remote* accesses. Accesses are done remotely when a local replica does not exist. The view consistency cost in the graph is divided into two parts: the cost of providing view consistency for file operations, and the cost for directory operations. Note from the upper graph that `find`, `rm` and `ls` have no view consistency overhead for file operations, since these operations predominantly operate on directories. The overhead of view consistency for remote accesses is also shown in the lower graph of Figure 7.2. This overhead includes both the file and directory operations. The overhead for all tests except `grep` is between 5 to 12 percent. The `grep` overhead is a horrible 185 percent.

To understand why grep performs this way, we performed some micro benchmarks and found that most of the overhead occurs because we obtain view consistency attributes separately from data (after each file operation). Thus for each file operation (for which view consistency is provided), we go over the wire twice. The gray area in the lower graph shows the overhead of going over the wire a second time for the view consistency attributes. We measured this overhead by using attributes that are cached during opens at the client kernel rather than going to the server for the attributes.[1] The lower graph shows that if attributes can be obtained with the data, the overhead for `grep` and for most other benchmarks decreases to between 1 to 8 percent. Again `cp` shows an overhead of 8 percent because it performs several file opens.

We have explained that most of the overhead of `grep` is due to going over the wire twice, and thus can be eliminated if the servers were to return the attributes along with the data. However, this does not explain the overall 185 percent overhead of `grep` and 11 percent overhead of `cp` although they perform similar kinds of operations. We again performed micro benchmarks with the kitrace [Kue95] kernel measurement tool. We found that the cost of getting remote attributes is 8.5 ms, cost of running `grep` on a single file is 17.4 ms and the cost of performing a `cp` from a remotely mounted file system to a remote replica in Ficus is 173.5 ms. A `grep` on each file gets remote attributes twice. This by itself doubles the

[1] This can violate view consistency for operations other than open, but is nonetheless useful for understanding the overhead.

time for executing a `grep` operation. The `cp` program gets remote attributes once for each file. Thus its cost increases from 173.5 to 182 ms, or an overhead of 5 percent.

Summarizing, the large overhead of `grep` is because the cost of getting attributes separately is a fixed cost operation and `grep` is a fast operation. Second, the overhead in gray in the lower graph of Figure 7.2 can be eliminated if the data servers are modified to return the attributes with the data. Finally, as explained in Section 4.1.2, the 2 to 8 percent overhead of view consistency can be reduced even further, by getting the view-entries from the local disk in parallel with the file operations.

7.2 Availability Measurements

Chapter 5 discusses issues related to replica switching for improving availability and for providing view consistency. Replicas are switched when the new replica improves overall access times and is view consistent. The overall availability of each replica is measured in terms of replica availability values as discussed in Section 5.1.2. The costs of replica switching are the selection of the fast-replica, checking the view consistency of the fast-replica, and performing the actual switch.

We performed an experiment that illustrates replica switching in Ficus. The aim of the experiment was to measure the costs and the benefits of replica switching. The replica availability values were simulated for the experiment as shown in the upper-most graph of Figure 7.3. This was done for two reasons: availability values do not change predictably in a real system, and do not change significantly (or frequently) in our well-connected LAN environment. The seven benchmarks that were used for measuring the overhead of view consistency for remote ac-

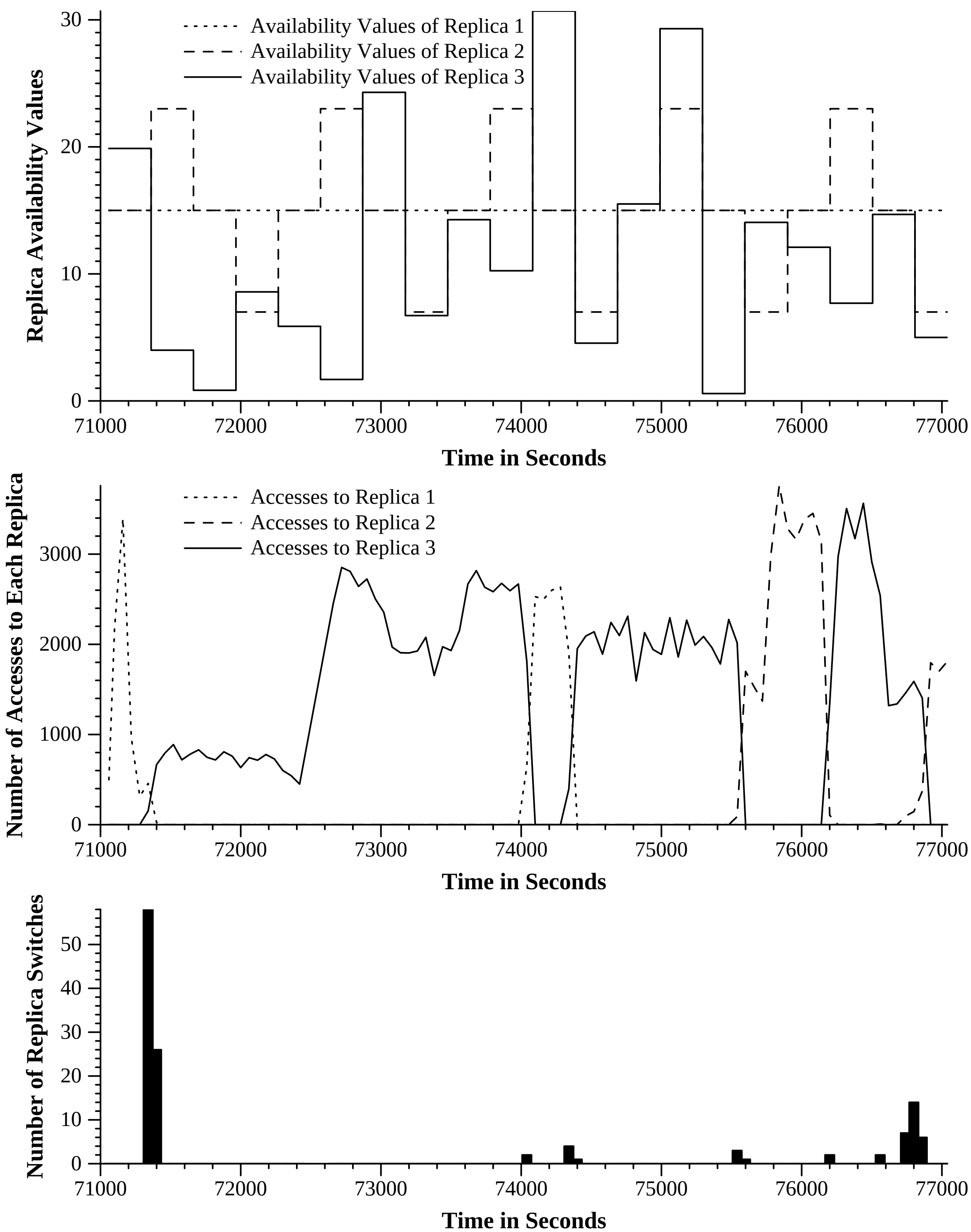

Figure 7.3: The replica availability values for three replicas, the replicas accessed, and the number of switches performed in a period of one hour and forty minutes

cesses (Section 7.1) were run with these simulated availability values. For each access, the replica that was accessed was logged (shown in the middle graph of Figure 7.3). Moreover, the number of replica switches during the period of the experiment[2] was also logged as shown in the lowest graph in Figure 7.3. The availability[3] values were changed every 300 seconds. They were fixed at 15 for replica 1, varied periodically between 7 and 23 for replica 2, and varied randomly between 0 and 31 for replica 3.

The lowest graph in Figure 7.3 shows the periods in which replica switching occurred. The middle graph shows that at the same periods more than one replica is accessed. Replica switching takes place when the currently accessed replica is at least *switching-factor* (set at 2) slower than the best replica. This is true for all the replica switches except the last one (around time 76800 seconds) when accesses switch from replica 3 to replica 2 although replica 3 is the fastest replica. This happens because replica switching for view consistency gives higher preference to the replica in the view-entry. The availability value difference between replica 2 and 3 is not much at this time. Replica 2 was being accessed in the past (around time 76000 seconds). Thus many files have replica 2 in their view-entry. Since replica 2 is not slow (not twice slower than the fastest replica), these files still try to use replica 2 (since it is known to be view consistent). This condition does not occur in any other part of the experiment.

The total number of accesses in the experiment was 192215. With no replica switching, the average access time would be 15 (in terms of availability values), since replica 1 would be accessed each time. The average time for each access

[2]The experiment was started at 19:44p.m. or approximately 71000 seconds since 12 a.m. While the average total time to perform these benchmarks is approximately 50 minutes (as can be seen by taking the sum of the elapsed times of each of the benchmarks in the upper graph of Figure 7.2), this experiment took 100 minutes because each access is logged to disk.

[3]A faster replica has a lower availability value.

with replica switching is 13.36 which represents an improvement of 11 percent. The ideal average access time (the replica with the lowest access time is accessed every time) is 9.68, a 36 percent improvement.

An interesting result of the experiment as seen in the lowest graph is that just a few replica switches induce every file to switch replicas. This happens because of the default replica switching rule. When a directory gets switched to a new replica, later accesses to the files in the directory start by accessing the same new replica. The replica in the view-entry of the file is the last accessed replica. However, this replica is very slow at each replica switching period (except the last one as explained above). Thus the new replica is given higher preference, and therefore accessed, and switching happens naturally for most files. Therefore the explicit cost of switching in Ficus is very low, and the *switching-factor* can be set to a value smaller than 2, thus further improving availability.

7.3 View-Entry Deletion Measurements

The view-entry database is a crucial element of the view consistency implementation. For example, its size can affect the performance of the system.[4] More importantly, its size can affect the system usability. A growing view-entry database can fill the available disk resources. Therefore, it is critical to be able to bound the database size.

There are several parameters that affect the average database size. The rate at which the database grows depends on the *number of accesses* to different files by an entity in a given time period. The rate at which the database can be truncated depends on the *truncation frequency*, the *data access pattern*, the *reconciliation*

[4]Since a hashing database package has been used in the implementation, we do not expect a significant reduction in performance with increasing database size.

frequency and topology between replicas, and the *number of replicas*. An effective evaluation must take all these factors into account. The data access patterns can vary widely, and several reconciliation topologies are possible. The reconciliation frequency and the number of replicas can also be varied. An effective evaluation requires a simulation that takes each of these factors into account. Such an effort is beyond the scope of this work.[5] Nevertheless, below we qualitatively examine the effect of each of these factors on the database size.

The version in the view-entry of a file is a vector quantity; its length is proportional to the number of replicas of the file. In Ficus, the view-entry size is $44 + 8 \times n$ bytes where n is the number of replicas. The 8 bytes per replica contain the version vector element and the replica identifier. Out of 44 bytes, 32 bytes are for the global Ficus file identifier. Since our experiments ran with 3 replicas, the view-entry size is 68 bytes. Assume that the database representation has approximately 50 percent space overhead. Thus each view-entry would require 100 bytes. Assume also that a user on an average accesses 200 *new* files (or files whose view-entries are not in the database) every day. Note that the user may access a much larger number of files but the view-entry of many of these files would be in the database because of file access locality [KPR94]. On this basis the view-entry database for each entity grows by 20 KB per day.

The rate at which the database can be truncated clearly depends on the frequency with which truncation is attempted. The average database size will become smaller as the frequency of truncation is increased. Other factors such as the data access pattern and the reconciliation topology affect the rate of database truncation, but do not directly depend on the truncation process itself. The view-entry

[5] A similar simulation has been done by Golding[GL93]. We hope to use his results to fine-tune our reconciliation topology.

is removable when the file modification time (and not the file access time[6]) in the view-entry is less than the acknowledgment time. Suppose the data is not modified often, or is mostly read-only. Then each replica eventually has the latest version and knows that others have seen this version. Thus the view-entry is immediately removable and can be removed the first time truncation is attempted. The view-entry truncation takes longer with increasing numbers of file writes.

Another important factor that determines the rate of view-entry truncation is the reconciliation topology. At Ficus, we use an adaptable ring topology to reconcile the replicas. The acknowledgment algorithm takes two round trip communications between replicas before every replica learns that all other replicas know about its state. Suppose each replica reconciles with the next replica in the ring in the correct order once a day. Then the acknowledgment timestamps are approximately two days old. Therefore, any file that was *modified* two or more days in the past has been acknowledged, and its view-entry can be deleted at the clients. Thus each entity needs a maximum of 40 to 60 KB of storage for the view-entry database. If files are mostly read-only and accesses do not modify the files, the storage requirements will be smaller. Note that there are two assumptions in the ring topology argument. First, a single ring of communication can complete in a day, and second, every replica can communicate with the next replica at the appropriate time. The first assumption may not hold with large numbers of replicas. A different reconciliation topology (such as a hierarchical one) can be used to speed up propagation of acknowledgment timestamps. The second assumption is more serious because replicas may not be readily available in a mobile environment. Assuming that users are willing to keep about 5 MB for the database, this would allow mobiles to remain disconnected for over 200 days.

[6]If the file access times were compared, all replicas would have to reconcile with each other before a view-entry could be removed.

The arguments in the previous paragraphs have been based on average data access rates, the reconciliation frequency and reconciliation topology in Ficus. More experience is needed with view consistency before we can confirm our analysis about the expected size of the view-entry database.

CHAPTER 8

Related Work

Database systems have traditionally used *one-copy serializability* for maintaining consistency of replicated data, because it can be enforced without knowledge of the semantics of data, or the programs modifying the data. Several approaches for supporting one-copy serializability in distributed systems have been proposed [BG81]. One-copy serializability uses a *strong consistency* replica control algorithm to map logical data into physical replicas. Strong consistency provides consistent data by restricting concurrency and availability of data [FM82, All83] to a single network partition. Moreover, it adds considerable communication overhead to support the needed synchronization. Both restricting availability and adding communication overhead contradict the goals of providing higher data availability and increased access speeds in distributed systems. *Replication* is used to overcome some of these problems. Various algorithms have been proposed for maintaining consistency among the replicated copies [Tho79, Gif79]. However, availability is still limited to a single partition when strong consistency is used for replica control.

8.1 Typed Consistency

Several researchers realized that accesses do not have to be serialized for maintaining data consistency for certain data types [FM82, All83, DS83, WB84]. The semantics of the data type can be used to define an alternative consistency criterion. Much of this research was aimed at providing consistency for the *directory* data structure that allows the *insert*, *delete* and *list* operations. Many applications such as mail and calendar use a variation of the directory data structure.

The directory consistency algorithm consists of two parts: data propagation and data integration. *Data propagation* ensures that an update (an insert or delete) operation at a replica reaches all the other replicas. Distributed logs are used to achieve data propagation. At most, an element can be inserted once. This ensures that once an object has been deleted, it cannot be inserted again. Similarly, the order of the two operations is always known, with the insert occurring before the delete. Moreover, *list* is a read operation that does not need to be serializable with respect to the other operations (it can return older data). *Data integration* (at each replica) uses these operation semantics to integrate the received update. The data propagation and integration steps help ensure eventual consistency of the replicas, even though the directory operations are not executed in a serially consistent manner.

The data propagation algorithm uses distributed logs. These logs must be garbage collected, so that the log sizes do not grow indefinitely, and their transfer between replicas does not become a bottleneck. The replicated log solution by Wuu [WB84] addresses this problem. Later, Heddaya, et al., [HHW89] propose a lower overhead solution, although garbage collection is more conservative. The acknowledgment algorithm presented in Section 6.5 borrows several ideas from Wuu.

Consistency algorithms that use data semantics provide high data availability. Garcia [GAB83], Sarin [Sar86] and Ladin [LLS90] generalize the directory solution for other data types. Garcia uses a data-patch tool that automates the data integration step. The correct final state of a database and corrective measures to reach it are predefined for various potential non-serializable operations. These are used to reach eventual consistency. Sarin uses commutativity of operations to transform the log at each site, so that it is mutually consistent with the logs at other sites. The lazy replication work by Ladin, et al., provides eventual consistency for updates and also supports causal ordering for reads.

Utilizing data semantics for eventual consistency is further explored by several systems such as Locus [PWC81], its successor Ficus [GHM90] on which we have implemented view consistency, Coda [Sat89] and Bayou [TTP95]. The data integration procedure is further developed in these systems and a distinction is made between *conflict detection* and *conflict resolution*. Locus, Coda and Ficus use version vectors to detect write-write conflicts. Bayou uses an application dependent mechanism for detecting such conflicts. It also detects read-write conflicts between replicas.

8.2 Client-Based Consistency

This section discusses work in which the clients (rather than the data replicas) provide consistency guarantees.

Unlike other eventual consistency systems that provide consistent data within a partition, Ficus and Bayou allow reads or writes to any available replica without requiring synchronization with any other replica. This provides the highest possible data availability. View consistency has been implemented on Ficus to reduce client inconsistencies. Similarly, Bayou provides session guarantees [TDP94] using the same version vector approach as ours. Session guarantees provide view consistency for processes or process groups. Bayou does not provide view consistency for other transient entities, or for persistent entities. For example, a user logging out of a session can see older data versions at the next login session. Finally, they do not discuss session guarantees for distributed entities. Our work can be generalized to distributed entities, as described in Section 9.2 on future work.

Client-based consistency has been used by Alonso, et el., [ABC90] to provide *quasi-copy* consistency. Quasi-copies are cached (or stashed) copies of data that may be somewhat out-of-date but are guaranteed to meet certain consistency predicates. For example, the predicate can state that the copy must not be more than ten minutes old, or more than two versions old. Client consistency for quasi-copies can generally be maintained only for age-dependent predicates. For example, the "not more than two versions old" predicate can only be enforced by the server. Unfortunately, it may not be possible to guarantee age-dependent predicates in a large distributed system, where network partitions are frequent, and it is essential to provide data availability.

Client-based computing has been proposed by Banerji and Cohn [BCK93]. They propose a mobile computing model where each user sees the computing world through a personalized view called the *computing persona*. Persona management includes cost-effective access to resources, and is the responsibility of the system software. Applications (with the help of application agents) migrate along with the mobile user by capturing their effective state at one site and then restarting the captured state at the next site. This solution is application specific and does not address issues related to file consistency in a replicated mobile environment.

8.3 Causal Consistency

Ladin, et al., [LLS90] support causal ordering of data reads, and causal, forced and immediate ordering for updates. This work borrows some of its ideas from previous work at ISIS [BJ87]. Since Ladin develops a more efficient solution for replicated systems, we will only discuss their work. *Causal* ordering is similar to Lamport's happened-before relationship [Lam78]. An operation A is causally related to operation B if the execution of B affects A, or if A's input can depend on B's output. The *forced* ordering ensures that a "forced" operation is performed in the same order at all the replicas with respect to *other* forced operations. Forced ordering is enforced by using a primary site mechanism, where the primary site orders the forced operations. A majority consensus method is used for committing the forced updates. *Immediate* operations are performed at all the replicas in the same order will respect to *all* other operations. A conservative locking protocol implements immediate operations. Both the forced and immediate operations require high network connectivity, which is not available in highly distributed or mobile environments. Thus we do not consider these operations to be useful in our environment.

The causal ordering of reads and updates provides guarantees that are similar to view consistency. However, unlike view consistency, which applies to all file data, support for causal ordering requires *application-specific* changes. Each application must specify the causal relationship between the operations that are performed by the application. Another important difference between causal ordering and view consistency is that causal ordering is enforced at the replicas (the data servers) while view consistency is enforced at each client. This has two implications. First, view consistency distributes the task of enforcing consistency to the clients, thus reducing the load on the servers. This makes view consistency more scalable (in terms of server load) than a causal ordering solution. Second, since causal ordering is enforced at the servers, it can provide inter-client (or inter-entity) guarantees. View consistency deals with this issue by combining the clients into a single entity, and providing consistency guarantees to this entity group.[1] However, the server-based causal ordering solution can provide more generic guarantees.

8.4 Replica Switching Schemes

Although considerable research has been done on providing consistent replicated data, not much work has focused on providing the fastest available replica to improve data availability. Systems have been built with either one or the other goal in mind. For example, wide-area file-system solutions provide consistent replicated (or cached) data, but do not aggressively search for the best available replica.

The replica switching work by Zadok and Duchamp [ZD91] addresses the problem of providing data from the fastest available replica. They improve the `amd` daemon [CS93] in UNIX systems (which demand mounts and unmounts file systems) and allow transparent switching of open files to *replacement* file systems that are dynamically discovered. The latency of the NFS `lookup` operation is monitored and used to assign availability values to different replicas. However their solution works for read-only file systems, because they do not deal with replica consistency issues. Thus issues related to tradeoffs between consistency and availability do not have to be addressed. Moreover, instead of individual files, whole file systems are switched at a time. Their switching policy aggressively

[1] The combining of the clients into a single entity can be done dynamically. View consistency must ensure that each of the clients access data that is the same as or newer than (later than) the latest data that any client has accessed

switches individual files to use the replacement file system. This disallows load balancing between the replicas since all files are accessed from either one replica or the other.

The NCSA scalable HTTP server by Katz, et al., [KBM94] also addresses some of the replica switching issues in their replicated server. The motivation for replicating the server was to serve an ever-growing body of clients. They use Transarc's Andrew File System to replicate their data. Replica switching is accomplished by dynamically changing the mapping between hostnames and IP addresses (using a modified *named* server and BIND protocol). Thus a single hostname can dynamically map to multiple data replicas. Each new query for the HTTP server returns the next data replica in a round-robin fashion.

The problem with this approach is that the consistency and availability issues are addressed separately. A system can provide better performance by considering both the issues together. For example, the round-robin approach may try to provide a replica that is currently not consistent. Consistency will ensure that the replica gets the latest copy before it returns the data. This may reduce the performance as compared to providing data from a consistent replica (that is perhaps more loaded). One of the goals of our work is to address switching issues for view consistency along with providing high availability and high system performance.

CHAPTER 9

Conclusions

We summarize the motivations for view consistency, key issues in its design and implementation, and directions for future research.

9.1 Summary

Replication for High Availability High availability of data is critical for many kinds of applications, including distributed applications. Distributed applications access data from several different machines. The key to providing highly available data for such applications is to replicate, or cache the data close to the location at which it is going to be accessed. Application clients can then access the nearby data replica most of the time. This replica is highly available and presumably provides the lowest latency and highest bandwidth access among all the data replicas. Both replication and caching introduce the data consistency problem. Intermediate states of data or inconsistent data may be visible during accesses unless special actions are taken.

Conservative Consistency Several systems have been built that use the one-copy serializability consistency protocol or its variants. These protocols require a strong consistency replica control protocol. Strong consistency ensures that each read or write access yields the latest version of the data. This makes it an attractive consistency model for users. Unfortunately, these protocols are conservative and disallow accesses in all but one network partition. Often communication links are inherently unreliable in mobile and highly distributed systems. In such situations, an application will not be able to access data, even though the data may be available in its partition.

Optimistic Consistency The approach taken by optimistically replicated systems is to allow accesses to any file replica at any time. This provides highly available data. Updates are generated at a site and integrated at other sites over time. This can lead to reads returning old data and to conflicting updates, or updates being made to old copies. These inconsistencies are detected and resolved when data is integrated, rather than when it is generated. The goal is to maintain the semantics of data without using a conservative criterion such as strong consistency. Optimistic consistency is essential for large-scale distributed systems because conservative schemes either require well connected networks or don't provide highly available data.

Problems with Optimistic Consistency Unfortunately, optimistic schemes do not provide any consistency guarantees during accesses. This does not generally cause serious problems for files that are write-shared among many users. It has been observed that such files are not frequently updated. Thus conflicting updates are uncommon for shared files. But the problem of old-reads for both shared and non-shared files and old-writes for non-shared files are still possible. The old-read problem is that users make updates and then read older copies of data in the future. The old-write problem for non-shared files is that a single user may update multiple copies of data and cause conflicts. Both the old-read and the old-write problems occur because consistency guarantees are not provided when the access is done.

View Consistency The aim of this work is to provide highly available and consistent data in very widely separated distributed and mobile systems. We propose the view consistency model that is built above an optimistic model and provides instantaneous consistency guarantees. The model allows an entity to have a consistent view of the data relative to the actions the entity has taken. This is a client or entity-based consistency model. It is our hypothesis that view consistency maintains most of the availability benefits of the optimistic model because inter-entity consistency is not enforced. The clients (and not the data servers) check for consistency during accesses. Moreover, no global information is kept or used for maintaining view consistency. The consistency criterion can be of different types, although the local, later-version criterion is most useful. Similarly, entities can be of different kinds. We have described the view consistency algorithms for centralized entities. Future work will focus on view consistency for distributed entities.

View Consistency Algorithm The view consistency implementation requires storage, retrieval and deletion of the version information of accessed files. The last version of each file seen by an entity is kept in a view-entry database. Files versions that are later than the view-entry version satisfy the consistency criterion. The retrieval and storage of the version information in the view-entry must be low overhead operations since all file operations that need view consistency invoke these operations. The number of view-entries in the database grows over time, and must be deleted so that the database size does not increase indefinitely. When all the file replica versions are known to be later than the file version in the view-entry, a view-entry is removable. An acknowledgment algorithm is needed to collect this information.

Availability and Replica Switching An important goal of a distributed system is to provide highly available data. We discuss replica switching algorithms that improve availability and, at the same time, provide view consistency. A distinction is made between accessibility (continuous access to some data replica) and optimality (accessing fastest data replica). While the former improves short-term availability, the latter improves long-term availability. There are tradeoffs in providing highly available data and consistency. Normally, view consistency does not significantly reduce availability. Moreover, non-view consistent data is very confusing to the user. Therefore, we take the position that data must always be view consistent even if it implies reducing availability. In a few cases where the user can tolerate inconsistent data, mechanisms can be provided to explicitly override view consistency.

9.2 Future Research Directions

There are several view consistency issues that are unresolved and need more research. More experience with view consistent systems is needed. Does view consistency satisfy the consistency demands of many applications? A user-level version of Ficus called *rumor* is currently being developed at UCLA. We plan to add view consistency to rumor and deploy it at various sites shortly. The benefits and costs of view consistency for large-scale disconnected and mobile use can then be measured more precisely. It will also give us an idea of the applications that benefit most from view consistency, and the applications that need higher consistency (at the cost of reduced availability).

A distinction is made between centralized and distributed entities in Section 2.2. The current system implements view consistency for centralized entities only. This is very useful when a user operates from a single machine all the time. For example, a user may carry a portable and work on it all the time. Here the machine is the centralized entity that sees consistent views.

A distributed-entity view consistent system will be useful for a much larger set of applications. For example, users work on different machines at different times. The user works on either a powerful desktop, or a server machine in the office. While commuting or at home, he uses a personal portable machine. Ideally, the user should be able to access files on any one of these machines at a given time, immaterial of whether the other machines are accessible at that time, and still see consistent files. Another example involves users working simultaneously on a small, multiple set of machines. They edit their files on one machine and compile them from the other machine. If these two machines access different replicas, the results can be very confusing. However, this problem will not arise if the two machines are considered as part of a single distributed entity.

These examples show that there are at least two modes of distributed operation: either the accesses from different machines can be non-overlapping in time, or they can be simultaneous. We discuss some thoughts related to implementing view consistency for both types of accesses.

9.2.1 Non-Overlapping Distributed Accesses

One method of implementing view consistency when accesses from a distributed set of machines are non-overlapping in time is to transfer the view-entry database itself. In effect, the view-entry database moves along with the user. Moving a user's state along with the user has been proposed earlier by Banerji and Cohn [BCK93]. Before files can be accessed from the new machine, the view-entry database from the old machine[1] is integrated with the database on the new machine. The integration involves taking a union of the view-entries of the two databases. If an entry exists in both the databases, the later entry is chosen. Once the integration is done, further file accesses will be view consistent with respect to both the machines.

The cost of the database integration mechanism lies in copying the database and its integration. Suppose a user goes home in the evening and starts working on his portable. A replica of the files that he uses in the office exists on his portable. The user has two options. The database can be copied from the office machine and integrated. If the local files are not view consistent (with respect to the integrated database), files are accessed remotely, or lazily copied (copied when required). The other option is to fully reconcile the replicas on the portable and in the office. It may often be cheaper to integrate the database and then lazily copy files, rather than do a full reconciliation. This would save copying those files that the user accesses in the office, but not at home.

If there is no connectivity between the two machines, database copying will not be possible. A solution to this problem is that the user carries a palmtop that stores the view-entry database. The palmtop (effectively being used as a portable secondary storage device) can be attached to the home machine, the office machine, or the mobile laptop. The palmtop can be used as the permanent storage area for the database, or it can be used for transferring (and then integrating) the database to the new site. If it is used as a permanent storage area, integration is not required. Even if there is no connectivity, the palmtop or the integrated database can be used to determine, and inform or deny access to older versions of files.

[1]Recall from Section 4.3 that the local replica optimization cannot be implemented for distributed entities. Thus each entity stores the view-entry database.

We assume that view-entries are much smaller than files. Otherwise, the user can carry the latest version of the files that he accesses most frequently in his palmtop. In the current web environment, files often have video and audio data attached to them. This makes the files much bigger than files in university UNIX environments. Moreover, the number of files being accessed on the web by individuals in a given time period is much larger than in traditional environments. Much work has been done on automatically identifying the most vital files that may be needed and then caching them on the mobile computer [Kue94]. When the space on the palmtop is small, no automatic method can be 100 percent correct or even close. In this context, storing version information is much more interesting than storing the data itself.

9.2.2 Simultaneous Distributed Accesses

The accesses of an entity from a distributed set of machines must be coordinated for view consistency. Each subsequent access from any of the set of machines should yield later data versions. This requires that the machines be connected to each other. If the machines in the set are network partitioned, view consistency cannot be guaranteed unless accesses are allowed in only one entity partition, just as with strong consistency. This is not considered acceptable in our environment. Then how should the machine set be defined so that the machines are always connected to each other?

A solution is that the machine set should consist of all machines to which a user is currently connected. The idea is that although a user may access files from multiple machines, there is a single user accessing these machines from a centralized point (or machine). All the machines can be coordinated through this centralized machine. The machine set varies with time but is always connected because the coordinating machine can access all these machines.[2]

The problem of synchronizing distributed accesses is well studied and several solutions exist such as primary coordinator, token passing, or voting. The primary coordinator approach has the benefits of simplicity. Moreover, it can be easily integrated with the palmtop database approach discussed earlier. We intend to study the viability of other schemes for providing view consistency to distributed entities.

9.3 The Final Word

This work aims to provide improved consistency guarantees in a highly available, eventually consistent, replicated environment. The consistency guarantees are provided at a low cost and without significantly reducing availability. This is done by providing conservative consistency to each user (or entity), while ignoring inter-entity consistency. Since it has been observed that conflicting inter-entity accesses are rare, it is not cost effective to provide inter-entity consistency. A more conservative consistency solution requires much higher interaction between replicas, which may not easily be available in a mobile, or large-scale distributed environment. Moreover, such consistency policies may significantly reduce availability, so that the solution may not be viable. We hypothesize that view consistency with replica switching for higher availability will adequately serve the consistency and availability needs of many current and future applications.

[2] If the coordinating machine cannot access one of these machines, then the user cannot access it also. This does not take into account background processes that do not have a controlling terminal.

References

[ABC90] Rafael Alonso, Daniel Barbará, and Luis L. Cova. "Using Stashing to Increase Node Autonomy in Distributed File Systems." In *Proceedings of the Ninth IEEE Symposium on Reliability in Distributed Software and Database Systems*, pp. 12–21, October 1990.

[All83] James E. Allchin. "A Suite of Robust Algorithms for Maintaining Replicated Data Using Weak Consistency Conditions." In *Proceedings of the Third IEEE Symposium on Reliability in Distributed Software and Database Systems*, October 1983.

[BCK93] Arindam Benerji, David L. Cohn, and Dinesh C. Kulkarni. "Mobile Computing Personae." In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pp. 14–20, Napa, California, October 1993. IEEE.

[BG81] P. Bernstein and N. Goodman. "Concurrency Control in Distributed Database Systems." *Computing Surveys*, **13**(2):185–221, June 1981.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. "Exploiting Virtual Synchrony in Distributed Systems." In *Proceedings of the Eleventh Symposium on Operating Systems Principles*, pp. 123–138. ACM, November 1987.

[CS93] Brent Callaghan and Satinder Singh. "The Autofs Automounter." In *USENIX Conference Proceedings*, pp. 59–68. USENIX, June 1993.

[DS83] Dean Daniels and Alfred Z. Spector. "An Algorithm for Replicated Directories." *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pp. 104–113, 17 Aug 1983.

[FE89] Richard A. Floyd and Carla Schlatter Ellis. "Directory Reference Patterns in Hierarchical File Systems." *IEEE Transactions on Knowledge and Data Engineering*, **1**(2):238–247, June 1989.

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

[GAB83] Hector Garcia-Molina, Tim Allen, Barbara Blaustein, R. Mark Chilenskas, and Daniel R. Ries. "Data-patch: Integrating Inconsistent

Copies of a Database after a Partition." In *Proceedings of the Third IEEE Symposium on Reliability in Distributed Software and Database Systems*, pp. 38–44, October 1983.

[GHM90] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier. "Implementation of the Ficus Replicated File System." In *USENIX Conference Proceedings*, pp. 63–71. USENIX, June 1990.

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

[GL93] Richard A. Golding and Darrell D. E. Long. "Modeling Replica Divergence in a Weak-Consistency Protocol for Global-Scale Distributed Data Bases." Technical Report UCSC-CRL-93-03, Computer and Information Sciences, University of California Santa Cruz, 1993.

[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.

[GW82] Hector Garcia-Molina and G. Wiederhold. "Read-Only Transactions in a Distributed Database." *ACM Transactions on Database Systems*, **7**(2), June 1982.

[HHW89] Abdelsalam Heddaya, Meichun Hsu, and William Weihl. "Two Phase Gossip: Managing Distributed Event Histories." *Information Sciences*, **49**:35–57, October 1989.

[HKM88] 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), 1994. To appear, ACM Transactions on Computer Systems, February 1994. Also available as UCLA technical report CSD-930019.

[HPG92] John S. Heidemann, Thomas W. Page, Jr., Richard G. Guy, and Gerald J. Popek. "Primarily Disconnected Operation: Experiences with Ficus." In *Proceedings of the Second Workshop on Management of Replicated Data*, pp. 2–5. IEEE, November 1992.

[KBM94] E. D. Katz, M. Butler, and R. McGrath. “A Scalable HTTP Server: The NCSA Prototype.” In *First International Conference on the World-Wide Web*, May 1994.

[KP87] Phil Karn and Craig Partridge. “Improving Round-Trip Time Estimates in Reliable Transport Protocols.” In *Proceedings, SIGCOMM ’87 Workshop*, pp. 2–7. ACM SIGCOMM, ACM Press, Aug 1987.

[KPR94] Geoffrey H. Kuenning, Gerald J. Popek, and Peter Reiher. “An Analysis of Trace Data for Predictive File Caching in Mobile Computing.” In *USENIX Conference Proceedings*, pp. 291–306. USENIX, June 1994.

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

[Kue94] Geoffrey H. Kuenning. “Design of the SEER Predictive Caching System.” In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, dec 1994.

[Kue95] Geoffrey H. Kuenning. “Kitrace: Precise Interactive Measurement of Operating Systems Kernels.” *Software—Practice and Experience*, **25**(1):1–22, January 1995.

[Lam78] Leslie Lamport. “Time, Clocks and the Ordering of Events in a Distributed System.” *Communications of the ACM*, **21**(7):558–565, July 1978.

[LLS90] Rivka Ladin, Barbara Liskov, and Liuba Shrira. “Lazy Replication: Exploiting the Semantics of Distributed Services.” In *Proceedings of the Workshop on Management of Replicated Data*, pp. 31–34. IEEE, November 1990.

[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.

[Pap79] C. H. Papadimitriou. “The Serializability of Concurrent Database Updates.” *Journal of the ACM*, **26**(4), October 1979.

[PWC81] 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.

[Rat95] David H. Ratner. "Selective Replication: Fine-Grain Control of Replicated Files." Technical Report CSD-950007, University of California, Los Angeles, March 1995. Master's Thesis.

[RHR94] 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.

[Sar86] Sunil K. Sarin. "Robust Application Design in Highly Available Distributed Databases." In *Proceedings of the Fifth IEEE Symposium on Reliability in Distributed Software and Database Systems*, pp. 87–94, January 1986.

[Sat89] Mahadev Satyanarayanan. "Coda: A Highly Available File System for a Distributed Workstation Environment." In *Proceedings of the Second Workshop on Workstation Operating Systems*. IEEE Computer Society Press, September 1989.

[Sel91] Margo Seltzer. "A New Hashing Package for UNIX." In *USENIX Conference Proceedings*. USENIX, January 1991.

[TDP94] D.B. Terry, A.J. Demers, K. Petersen, M.J. Spreitzer, M.M. Theimer, and B.B. Welch. "Session Guarantees for Weakly Consistent Replicated Data." In *Proceedings of the Third International Conference on Parallel and Dsitributed Information Systems*, pp. 140–149, sep 1994.

[Tho79] Robert H. Thomas. "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases." *ACM Transactions on Database Systems*, **4**(2):180–209, June 1979.

[TTP95] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System." In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, pp. 172–183. ACM, December 1995.

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

[ZD91] Erez Zadok and Dan Duchamp. "Discovery and Hot Replacement of Replicated Read-Only File Systems, with Application to Mobile Computing." In *USENIX Conference Proceedings*, pp. 69–85, Cincinatti, OH, June 1991. USENIX.