Roam: A Scalable Replication System for Mobile and Distributed Computing

David Howard Ratner

University of California, Los Angeles

January, 1998

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

UCLA Computer Science Department
Technical Report UCLA-CSD-970044

Thesis committee:
Gerald J. Popek, co-chair
W. W. Chu, co-chair
Eli Gafni
Mario Gerla
Donald Morisky

To my father Robert, my mother Ellen, and most importantly my wife, Darcy.

Contents

List of Figures

List of Tables

Abstract of the Dissertation

Roam: A Scalable Replication System for Mobile and Distributed Computing

by

David Howard Ratner
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1998
Professor Gerald J. Popek, Co-chair
Professor W. W. Chu, Co-chair

Mobile computing is rapidly becoming a way of life. Users carry their laptops, PDAs, and other portable devices with them almost constantly, whether their mobility takes them across town or across the world. Recent hardware innovations and improvements in chip technology have made mobile computing truly feasible.

Unfortunately, unlike the hardware industry, much of today's system software is not "mobile-ready." Such is the case with the replication service. Nomadic users require replication to store copies of critical data on their mobile machines, since disconnected or poorly connected machines must rely primarily on local resources. However, the existing replication services are designed for stationary environments, and do not provide mobile users with the capabilities they require. Replication in mobile environments requires fundamentally different solutions than those previously proposed, because nomadicity presents a fundamentally new and different computing paradigm. Mobile users require several key features from the replication system: the ability for direct synchronization between any two replicas, the capability for widespread scaling and large numbers of replicas, and detailed control over what files reside on their local (mobile) replica.

Today's replication systems do not and cannot provide these features. Therefore, mobile users must adapt their behavior to match the service provided to them, complicating their ability to work while mobile and effectively hindering mobility by costing the user additional time, money, and system-management effort simply to operate in a nomadic mode.

Here we present Roam, a replication system designed to satisfy the requirements of the mobile user. Roam is based on the *Ward Model*, a replication architecture specifically aimed at mobile environments. Together with the Ward Model and new, distributed algorithms such as improved garbage collection techniques and version vector management, Roam provides a scalable replication solution for the mobile user. We describe not only the design and motivation of such a system, but its implementation and performance as well.

Acknowledgments

This work would not have been possible without the expert tutelage, advice, and guidance from my advisor, Dr. Gerald Popek. Jerry helped focus my work and constantly encouraged me to look at the big picture, extending both the goals and an understanding of my research.

I would like to make a special acknowledgment to Dr. Peter Reiher who, while never officially bearing the status of advisor, nevertheless functioned as one. Peter always made himself available for discussions and critiques of current designs and implementation issues, and as such played an instrumental role in developing this research.

I would not have been able to dedicate myself to this research had I not been financially supported. I would like to acknowledge the support of the Defense Advanced Research Projects Agency (DARPA). During my graduate years, my work was supported by DARPA contracts N00174-91-C-0107 and DABT63-94-C-0080 under the direction of various project managers: Brian Boesch, Barry Leiner, Kevin Mills, and Rob Ruth.

I also want to thank several of my co-workers on the Ficus and FMG projects—specifically, Ashvin Goel, John Heidemann, Ted Kim, Geoff Kuenning, and Mark Yarvis. Each of them spent countless hours with me discussing design decisions and implementation problems, reviewing papers, and generally encouraging me. Additionally, I want to acknowledge Richard Guy, for much of my research built on top of Richard's previous work with FICUS, and I owe him a great deal for that.

Further, I would like to acknowledge Professor Ken Birman from Cornell University, whose many classes and projects started me on this long road, and two essential administrative assistants, Janice Martin at UCLA and Alta Stauffer at Platinum *technology, inc.* Janice spent countless hours editing the dissertation, and I am especially thankful for her effort.

I want to make a special acknowledgment to the Coffee Bean and Tea Leaf, located in downtown Westwood Village. I spent countless hours working there, and perhaps got more accomplished in their coffee house than at my own office. It is a tribute to mobile computing.

My family was instrumental throughout this whole process. Without the support of my mother, father, and sister, I do not believe I could have finished this difficult journey. Additionally, my father Robert helped with some of the mathematical development in Chapter 9.

Finally, and perhaps most importantly, I would like to acknowledge the eternal support and encouragement from my new wife, Darcy Richardes. Darcy was always there for me (as I hope she always will be), encouraging me when I was depressed, congratulating me when I was excited, and just listening to me when I needed to talk.

Chapter 1

Introduction

Computing in the 1970s and 1980s meant using almost completely stationary machines. Designed to be situated in one location and rarely if ever moved, these machines were heavy, awkward, and clumsy. Physical movement was so impractical that software designers only considered and developed algorithms with the underlying assumption that movement just did not occur. The resulting software architectures were adequate for the time, because these machines were too clumsy to move on any regular basis. Even the emerging "portable" machines at the end of the 1980s were not really portable; they could at best be described as "luggable." Simply putting a handle on a machine does not make it portable, because if it's not convenient to carry, it won't be used in a mobile fashion. Additionally, the luggable computer's compute-power to cost ratio made it rather unattractive from a user's standpoint, especially given the fact that portability was still awkward. Stationary machines were a way of life, and true mobility was still an unattainable goal.

However, by the 1990s the hardware industry had made rapid progress in chip and LCD technology, as well as in general miniaturization. Machines emerged that were truly portable and as powerful as their stationary cousins. Making use of this new type of machine with its attendant portability, users became increasingly mobile. In 1996, approximately one third of the computers sold were mobile-enabled: that is, portable form-factor with communications capability [Popek 1997].

With their new ability to compute while mobile, users unfortunately found themselves vastly under-equipped from a software architecture point of view. The hardware allowed them to be mobile, but the antiquated system software still assumed a relatively static world. This assumption not only made mobile computing more cumbersome than necessary, but in some cases actually hindered mobility and restricted functionality.

Examples of system software that assume a stationary and static world are pervasive in standard fields of computer science, and include areas such as Internet routing and the receiving of email: areas that mobile users often complain about as not working adequately or in a manner that facilitate mobility. Another important example, and the one we are interested in, is that of *data replication*. Replication services allow data to be simultaneously stored at multiple sites for improved performance and availability. Additionally, the replication service guarantees that all copies or *replicas* eventually represent the same set of updates; this is also known as *maintaining consistency* or *synchronization*. Many replication services exist today, but like the Internet routing scenario, they are not capable of adequately handling mobile computing.

For example, consider the requirements and typical usage patterns of two stereotypical mobile users, Alice and Bob. Alice is a university professor collaborating on her research with other professors at other universities. Data is updated at each location, and updates are exchanged between locations. Occasionally these professors travel, sometimes to interact directly with one another and sometimes on other business. Regardless of the reason, they want the ability to share data with the other interested parties.

Should the professors decide to jointly write a paper and present their research at a conference, Alice will update and revise sections independently of the others. She expects her updates to be transparently integrated into the paper as a whole. Prior to the presentation, the professors will meet at the conference with their laptops to discuss final revisions and perform last-minute changes. They require the ability to directly interact and exchange updates, ultimately ensuring that the presenter has the absolute latest version.

Bob has a related but different set of requirements. He is a manager in a major corporation. He works on his office desktop during the day and takes his laptop home at night. In the office Bob uses the machine with the larger screen and improved resources, but in the evening and during weekends, he works on the laptop. Therefore, data must be stored at both locations, and consistency must be maintained between the two sites. Additionally, as a member of a large corporation, the data he accesses and writes is accessed and written by others as well. The data replicas on the laptop and the office desktop will not be the only two in existence, and consistency must be maintained across all replicas throughout the company.

During meetings both within the office complex and across town, Bob brings his laptop to take notes and present data. Bob and the other users want the ability to directly connect their machines and propagate updates around the room, either by a wireless, infrared, or cabled connection. Additionally, each user arrives at the meeting with potentially different information—some users have more recent updates than others. Updates should be directly shared among the other members, instead of forcing everyone to either gather around a single screen or wait for the updates to eventually propagate through the normal paths of the consistency mechanism, which may not take effect until each user returns to his or her home office.

Furthermore, Bob occasionally travels over long distances to remote locations, bringing his laptop both to work while traveling and to interface directly with computers in the new geographic area. Working while traveling allows him to make optimal use of his time. Interfacing directly with computers in the new area provides improved synchronization performance compared to contacting a distant machine in, for example, his home office. These actions require not just replication, but the ability to dynamically change the set of participants with which Bob's machine typically synchronizes. In other words, the overall topology that maintains data consistency must be dynamically adaptable, as Bob and other users move geographically.

The Alice and Bob situations are not unrealistic; in fact, many users today would like to operate in the same mode. Additionally, these are not isolated examples. Other examples with the same general pattern of mobility, replication, and update distribution include military situations, software development scenarios, distributed database problems like airline reservation systems, and general-purpose distributed computing.

However, current replication facilities cannot support these types of operations and scenarios because they were designed assuming non-mobile machines. As a result, the existing systems actually hinder, rather than help, widespread mobile computing. There are many different replication systems in existence today, but none of them can satisfy the requirements of Alice and Bob.

Systems based on *conservative* replication strategies [Prusker *et al.* 1990, Liskov *et al.* 1991, Brereton 1986, Davčev *et al.* 1985, Herlihy 1986, Thomas 1979, Guerraoui *et al.* 1996] are not suitable for replication in mobile environments. Conservative or pessimistic strategies such as primary-site [Stonebraker 1979], majority-vote [Pâris 1989, Renesse *et al.* 1988, Bastani *et al.* 1987] or token-based techniques do not allow multiple writers when the writers cannot directly communicate to serialize the order of operations. The conservative strategies provide very high consistency but decreased availability in terms of how often users can generate updates. While useful in local area network (LAN)-style applications, they do not extend well to environments that expect both multiple writers and common network disconnections and partitions. Mobile users are often disconnected from the network—and even if they are connected, it is often via an expensive, high bandwidth, low latency link that users do not want to depend on for correctness. Additionally, an important objective of mobile computing is to permit update generation when mobile. Unless the mobile user is the only user allowed to generate updates, conservative strategies cannot apply to the mobile scenario. For example, suppose Bob replicates the company's inventory database on his laptop using a conservative protocol like primary site. Either the primary site is Bob's laptop, in which case while traveling Bob prevents all other users from updating the database, or the primary site is a machine in the office, in which case Bob cannot update the database when he travels. A similar situation exists for all conservative protocols, regardless of the specific mechanism employed.

We therefore require *optimistic* replication strategies [Davidson 1984, Davidson *et al.* 1985, Alonso *et al.* 1989, Blaustein *et al.* 1985, Ceri *et al.* 1992, Satyanarayanan *et al.* 1990, Guy *et al.* 1990a]. Optimistic strategies provide high availability by allowing independent updates to all replicas, detecting and resolving the possible *conflicts* created by concurrent updates some time later [Kumar *et al.* 1993, Reiher *et al.* 1994]. However, optimistic replication is only one facet of the solution, as there are many different optimistic replication models.

Systems based on the *client-server* model [Satyanarayanan *et al.* 1990, Honeyman *et al.* 1992] make a class-based distinction between "clients" and "servers." The client-server model by definition prohibits direct communication and synchronization among clients (i.e., the mobile machines), and servers are defined as expressly being non-mobile. Clients can only synchronize by communication through a third-party server, blocking direct inter-client communication in a meeting, a hotel room, or wherever multiple machines may encounter each other. Within the client/server framework, cooperating users in the hotel room must compete for the common phone line to contact a remote server (typically a long-distance call) and download updates over standard modem transmission speeds that were just uploaded by a computer on the other side of the room. The client-server model forces users to take extraordinary actions, such as long distance telephone calls over low-bandwidth links, to conform to the underlying static model and pretend that they haven't really moved. Users pay additional time and money to perform these extraordinary actions, while receiving a degraded level of service. Even in cases where client-to-client interaction is not strictly required, clients are typically configured to only communicate with one predefined server. Long-distance geographic motion stresses and strains the system, as each client must still communicate with its "home" server instead of synchronizing with a local machine situated in the new region.

Systems based on the traditional *peer-to-peer* model [Guy *et al.* 1990a, Terry *et al.* 1995, Reiher *et al.* 1996] allow direct communication and synchronization between all participants, but exhibit scaling problems, typically limiting the systems to one or two dozen participants. Scaling problems exist because, to date, all peer solutions have been implemented by storing all relevant information at every participant in the system. While the restriction on scale may be adequate in Alice's case, it is a serious burden, if at all acceptable, in Bob's case.

Additionally, most peer models utilize a replication granularity that users find inefficient in synchronization time and disk space usage. Both Ficus and Rumor initially provided replication at the *volume* granularity; [1,2] Bayou [Terry *et al.* 1995] replicates entire databases. Regardless of the type of container, the systems force the container to always be *fully-replicated*—either the entire container is replicated locally or else none of it can be replicated locally. Coda [Satyanarayanan *et al.* 1990, Kistler *et al.* 1991], while permitting more flexible sharing arrangements at its clients, also forces full replication between its peer servers. The full-replication policy makes the replication algorithms easier to develop and implement, but causes headaches for mobile users. Unlike their stationary cousins, mobile users are not well-connected to vast network resources and disk pools. If a stationary user doesn't have a particular data object locally replicated, he or she can obtain it relatively cheaply from the network; mobile users, however, cannot. Mobility forces users to rely much more heavily on their local resources, as obtaining data from a non-local site is typically orders of magnitude more expensive, if at all possible. Thus, mobile users want the ability to replicate exactly the data objects that they require, regardless of the "container" in which they logically reside. Unfortunately, most peer solutions force users to replicate the entire container, wasting valuable disk space on data the user doesn't require and costing the user additional connection time to maintain consistency on the unwanted data.

In summary, the central problem with mobility and replication is that mobile users replicate their data using systems that were not designed for mobility. Instead of the replication system improving the state of mobile computing, it actually can hinder mobility. Users must adjust their physical motion and computing needs to match what the system expects, which is inefficient, inappropriate, and highly unsuitable for true mobile computing.

Roam is a replication solution redesigned especially for mobile computing. Built using the Ward

[1] A volume is smaller than a file system but larger than a directory [Satyanarayanan *et al.* 1985]. For example, a user's home directory and all sub-directories might constitute a volume.

[2] Both Ficus and Rumor eventually provided a more fine-grain replication control facility, but this mechanism was implemented either as a Roam predecessor or as part of Roam itself, and was not part of the original design.

Model [Ratner *et al.* 1996c, Ratner *et al.* 1996b], it enables true mobility by providing:

- The ability to optimistically replicate data.
- Algorithms for the *selective replication* [Ratner *et al.* 1996a, Ratner 1995] of data; that is, data can be replicated independently of the logical "container" in which it resides.
- A peer-based model so that any replica can directly communicate and synchronize with any other replica, regardless of its origin or original location in the system.
- Modifications to the traditional peer model to allow greatly improved scaling in the number of replicas.
- A location-independent architecture, so that the consistency algorithms adapt to match the physical location of the replicas. Additionally, the mechanisms exhibit a cost-effective, *lazy* behavior: heavyweight actions do not occur on behalf of a transient and short-lived physical move.
- System controls that allow any replica to become mobile without *a priori* specification while not significantly increasing the costs or system overhead when replicas don't move.
- New and improved distributed algorithms that accomplish necessary replication tasks more quickly and minimize data structure size, when compared to the standard algorithms used in traditional peer models.

Roam addresses scalability with a three-pronged attack. The Ward Model addresses issues of replica management, consistency topologies, and update distribution. Dynamic algorithms and mechanisms handle the scalability of the versioning information, required for consistency maintenance. Finally, we address consistency itself with new algorithms and mobile-friendly semantics. In summary, Roam is a comprehensive replication system for mobile and non-mobile users alike. With it, users can truly compute while mobile, paving the way for both improved user productivity and new and unseen research along mobile computing avenues.

Chapter 2 discusses the solution space, the variables that must be specified before designing a replication system, our position in the solution space and the motivation behind our decisions. It also provides definitions of mobility and an enumeration of mobile users' requirements. Chapter 3 provides a background into replication, Ficus, Rumor and other replication definitions and topics. The design and implementation of the Ward Model, the basis of Roam, are described in Chapter 4. Chapter 5 discusses the maintenance of consistency, while Chapter 6 describes the details and implementation of selective replication. Chapters 7 and 8 discuss the role and details behind two key distributed algorithms—garbage collection and dynamic version vector management, respectively. A performance evaluation is provided in Chapter 9. Related work is described in Chapter 10, some future work ideas in Chapter 11, and we conclude our discussion in Chapter 12.

Chapter 2

Solution Requirements

There are many different variables that must be considered before adopting a particular data replication solution. These variables construct an n-dimensional solution space, where any specific solution occupies a specific point in that space. Without careful consideration of the various facets involved, the adopted solution may fail to operate correctly or adequately in the target environment. It is for this very reason that we are redesigning a replication system for mobile environments. Previous endeavors did not consider support for mobility an important factor, and therefore did not choose a position in the solution space that was mobile-compatible.

This chapter first outlines the different variables that constitute the solution space. We then discuss mobility, providing both definitions and user requirements. We conclude with our position in the solution space and the motivations behind each decision.

2.1 Solution Space

A replication system has many independent variables, that together constitute an n-level solution space. These variables are as follows:

Network partition frequency: This variable expresses how often a partition occurs somewhere in the network, and ranges from almost never to infinitely often. Network in this sense refers not to a LAN-style entity, but more generally to the set of connections that connect all participants.

Length of partition time: Irrespective of how often partitions occur, one must also decide the length of time a given partition is assumed to remain in existence. Even if partitions occur infinitely often, if any single partition was instantaneously healed, then no node would be partitioned for any significant length of time. The length of time can range from zero to infinity.

Partition size: Finally, we must also consider the expected size of partitions when they do occur. If the partition size was never greater than one—that is, a partitioned machine never communicated with anyone until the partition healed—we could utilize very simple algorithms. We would only have to allow partitions to occur; we wouldn't necessarily have to enable communication between various participants during partitions. However, if partitions could be arbitrarily large, we would require much more robust algorithms and controls.

Clock assumptions: Computers have internal clocks to assist them with processing and decision-making. The set of clocks at all participants can be either synchronized or unsynchronized.

Replication model: There are a variety of different replication models. The basic three are master-slave, client-server, and peer-to-peer, discussed in depth in Chapter 3.

Consistency method: Independent of the replication model is the method the system uses to maintain consistency between all replicas. Consistency can be maintained optimistically, conservatively, or with a mixture of both approaches.

Replicated data: Designing a replication system involves selecting what type of data to replicate. One could replicate databases, the file system, World Wide Web pages or other alternatives. The choice impacts not only the other design parameters but also the specific target environments for deployment. For instance, the vast majority of web page accesses are read-only, while a significant percentage of file system accesses are writes. A system aimed at replicating web pages would therefore have very different constraints from one replicating file system objects.

Update rate: Closely tied to the type of data is the notion of how often new updates are generated. Clearly, systems may be designed very differently depending upon whether updates are exceedingly rare or they occur rapidly and constantly. The update rate can range from never to infinitely often, and typically depends on the choice of replicated data.

Update distribution: Independent of the update rate is the distribution of updates among the objects in the system. One million updates to a single file is very different than one update each to a million files. The update distribution captures the notion of how updates are applied to the various replicated objects, and also often depends on the choice of replicated data.

Replication granularity: A system that provides replication does so with a fixed unit of replication, and users cannot replicate on a finer granularity than that unit. The unit is essentially arbitrary. When replicating file systems, typical choices are a complete file system, a volume, a directory or a file. In the database context, the unit could range from complete databases to individual records or fields in a record. Other choices are clearly possible.

Allowable replication factors: A replication system fundamentally exists to provide multiple replicas of file objects. The system can provide both read/write replicas and read-only replicas; read-only replicas are generally much easier to maintain, given their limited effect on the system as a whole. In both read/write and read-only cases, the system has a maximum allowable number of replicas above which the system either performs poorly or fails completely. The number ranges from two to infinity.

Number of system participants: Independent of the number of replicas of any given object is the number of participants using the system. Each participant could be replicating different data, so theoretically the number of participants could be infinite even if the allowable replication factor was two. Conversely, a single user could require any number of replicas of a single object. The number of system participants ranges from one to infinity.

Level of trust: Security is a always a concern, and security becomes an even bigger problem in the mobile domain. This variable indicates the degree of trust bestowed upon the participant machines: that is, whether or not we believe them to be operating correctly and in an uncompromised state. The variable ranges from high degrees of trust (trust everyone) to low degrees of trust (trust no one).

Degree of mobility: Not all participants need be mobile-enabled; a system could, for instance, deem that only certain replicas could act in a mobile fashion. The remainder would be traditional stationary players in the system. This variable identifies what percentage of the system participants can act in a mobile fashion, and ranges from zero to one hundred percent.

Length of time one remains mobile: Since mobile users may operate away from their "home" for varying lengths of time, the issue must be addressed regarding how long a mobile user may remain mobile. This variable ranges from zero (no mobility) to infinity (a mobile user need never return to his or her original "home").

Reliability Leslie Lamport has said "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable" [Lamport 1987]. His comment was not entirely in jest. A distributed system has many inter-related machines, not unlike a large machine with many inter-connecting parts. The reliability of the system as a whole reflects the stability and correct operation of any given site in the face of the failure of one (or many) other sites.

There are also many facets that are independent of the actual replication system, and these are not discussed here. Examples include the choice of transport mechanism and security protocols or features used during transport.

2.2 Mobility

Mobility has rapidly become a buzz-word that many people use without a solid, underlying definition. Without first understanding what environment we are envisioning and what we mean by mobility and mobile computing, it is difficult, if at all possible, to postulate a solution. This section describes our definition of mobility, mobile machines, and mobile users. We then discuss the requirements that these definitions place on the replication system.

2.2.1 Definition of mobility

Kleinrock [Kleinrock 1997a] defines four types of mobility, and we adopt his definitions here:

Social: moving from one context or mind-set to another

Physical: moving from one physical location to another

Appliance: moving from device to device or system to system

Application: moving from one task to another

This dissertation is primarily concerned with support for physical and appliance mobility, although the other forms should not be neglected. Within the realm of physical mobility, we define a mobile machine to be any computer or computing device that is small, lightweight, and capable of easily being moved. A large desktop machine is in some sense portable in that it can be physically moved from one destination to another, but we do not include it in our definition of a mobile machine because the physical motion is not *easy* to perform; it requires disconnecting cables, properly packing it in shipping containers, and a host of other procedures. Our definition of a mobile machine uses the underlying criteria that the machine must be trivial to move, exhibiting a "get-up and go" type of behavior. Additionally, the machine should not require extensive setup or tear-down time to make it mobile. Over one-third of form-factor machines sold today fit this description, and the market percentage is constantly growing [Popek 1997].

A mobile user is, therefore, someone who uses a mobile machine and occasionally travels with the machine. Regardless of how often the traveling occurs or how far the distance traveled, we include all such examples in our definition of a mobile user. Users who travel infrequently nevertheless require the *ability* to be mobile users when the need arises. While some forms of mobility can be reasonably predicted, such as a business-person leaving the office every day at five o'clock, others appear more random and even chaotic—the sudden emergency, the unscheduled meeting, or the occasional schedule change. For example, a business-person may never travel for business and may never work at home, but occasionally may need to work from locations other than his or her normal office due to the irregularities and unpredictable twists of normal life.

Similarly, users who only travel short distances when mobile nevertheless require much the same functionality as those that travel long distances [Kleinrock 1997b]. Regardless of the distance, any change in location should still be considered an act of mobility because of the changes in the environment, including but not limited to location, communication medium and accessible devices. For example, moving just from one's desk to the sofa across the office may result in changing the communication protocol from Ethernet to a wireless LAN. Different resources (like printers) might become available while existing resources become less accessible. Communication to a central server could become more difficult, more expensive, or slower, making communication with a nearby peer more attractive. Thus, even when the central server is theoretically accessible or the degree of motion is very small, users may prefer the ability to directly communicate with a nearby neighbor.

2.2.2 A mobile user's requirements

Mobile users have special requirements above and beyond those required by all users wishing to share and replicate data. In this section we discuss the requirements that are particular to mobile use: any-to-any communication, larger replication factors, detailed controls over replication behavior, and the lack of pre-motion actions. We omit discussion of well-understood ideas, such as the case for optimistic replication, discussed in [Guy *et al.* 1990a, Heidemann *et al.* 1992, Kistler *et al.* 1992, Satyanarayanan *et al.* 1993].

Any-to-any communication

By definition, mobile users change their geographic location. Accordingly, it cannot be predicted or established *a priori* which machines will be geographically co-located at any given time. Given that it is typically cheaper, faster, and more efficient to communicate with a local partner rather than a remote one, mobile users want the ability to directly communicate and synchronize with whomever is "nearby." Consistency can of course be correctly maintained even if two machines cannot directly synchronize with each other, as demonstrated by systems based on the client-server model [Honeyman *et al.* 1992, Satyanarayanan *et al.* 1990], but local synchronization increases usability and the level of functionality while decreasing the inherent synchronization cost. The issue is usability, expected functionality, and inherent cost. Users who are geographically co-located don't want updates to *eventually* propagate through a long-distance, sub-optimal path; the two machines are next to each other, and the synchronization should be nearly instantaneous.

Users understand and are used to dealing with delays based on geographic distance. For instance, common knowledge dictates that it takes longer to send a postal letter or digital data between distant locations than nearby ones. Users expect the distance-invariant to remain true even when mobile. For example, sending a postal letter from Los Angeles to New York may take four to five days, but if the sender physically travels to New York and mails it from there, it should only take one or two days. The same should therefore be true of data synchronization. Machines that are closely connected should synchronize more quickly than those farther apart.

Since users expect that nearby machines can synchronize with each other quickly and efficiently, and it cannot be predicted which machines will be geographically co-located at any point in the future, we require a replication model capable of supporting *any-to-any communication*. That is, the model must allow any machine to communicate with any other machine; there can be no second-class clients in the system. For instance, compare the class-based environment depicted in Figure 2.1 with the peer environment of Figure 2.2. In the first figure, the mobile users experience very poor synchronization performance; they really want the environment supported by the second figure.

Any-to-any communication is also required in other mobile contexts, such as in appliance mobility. A given user may utilize a desktop machine, a laptop, and a palmtop. Enforcing a fixed order on the synchronization between the three machines means that two of the machines cannot directly synchronize without using the third as an intermediary. While such a policy potentially makes sense if all three are stationary and typically connected, it seems ludicrous in a mobile environment, where it is quite possible that the two machines that cannot directly communicate may need to. For example, if the established

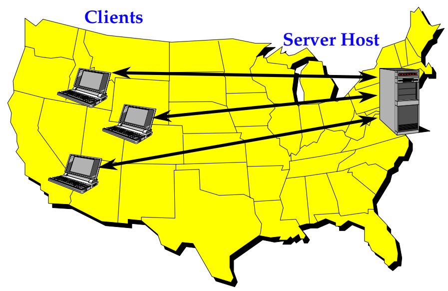

Figure 2.1: Mobile, co-located clients cannot synchronize directly among themselves.

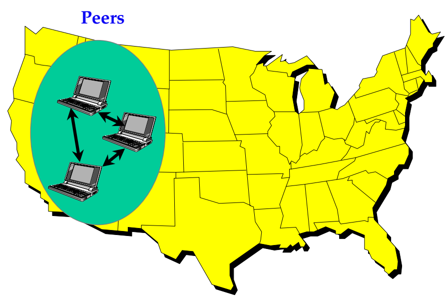

Figure 2.2: Peers can directly synchronize among themselves.

policy is that the palmtop synchronizes with the laptop which synchronizes with the desktop, users are barred from exchanging updates directly between the palmtop and the desktop. Forcing users to always synchronize the palmtop with the laptop prior to synchronizing the laptop with the desktop may hinder mobility more than enabling it. Similar examples exist in the social mobility context as well.

Providing any-to-any communication is equivalent to utilizing a peer-to-peer replication model; if anyone can directly synchronize and exchange updates with anyone else, then everyone must by definition be equals, at least with respect to update-generation and reconciliation abilities. Some, however, have argued against peer models in mobile environments because of the relative insecurity regarding the physical devices themselves—for example, laptops are often stolen. The argument is that since mobile computers are physically less secure, they should be "second-class" citizens with respect to the highly secure servers located behind locked doors [Satyanarayanan 1992]. The class-based distinction supposedly provides improved security by limiting the potential security breach to only a second-class object.

The argument is based on the assumption that security features must be encapsulated within the peer model, and thereby the unauthorized access of any peer thwarts all security barriers and mechanisms. However, projects such as TRUFFLES [Reiher *et al.* 1993b] have demonstrated that security policies can be modularized and logically situated around a peer replication framework while still remaining independent of the replication system. Tight controls can be placed on individual replicas, providing good security mechanisms even within an underlying peer framework. With such an architecture, the problems caused by the unauthorized access of a peer replica are no different from the unauthorized access of a client in a client-server model. Thus, the question of update-exchange topologies (any-to-any as compared to a more stylized, rigid structure as in client-server models) can be dealt with independently of the security issue and the question of how to enforce proper security controls.

Finally, Kleinrock (among others) has recently observed that the term "client-server" is technically a misnomer, as the more appropriate description is client-*network*-server [Kleinrock 1997b]. The network is an integral part of the client-server model. Since nomadicity exacerbates network connectivity, the network effectively shrinks and often becomes non-existent. CODA has demonstrated that the client-server model can be optimized in several ways to handle the shrinking network problem [Mummert *et al.* 1994, Mummert *et al.* 1995]; however, an overall cleaner approach may result from utilizing a peer model, and therefore not relying on the network's presence or the need for optimizations.

Larger replication factors

Most replication systems only provide for a handful of replicas of any given object. Additionally, peer algorithms have never traditionally scaled well in the number of replicas. Scaling has not typically been an issue with replication services, because most scenarios have not required large numbers (more than a dozen) of writable replicas. Some have, in fact, argued that peer solutions by their nature simply cannot scale to large numbers [Satyanarayanan 1992, Gray *et al.* 1996].

However, while mobile environments seem to require a peer-based solution (as described above), they also seem to negate the assumption that a handful of replicas is enough. While we do not claim a need for thousands of writable copies, it does seem likely that the environments common today and envisioned for the near future will require larger replication factors than current systems allow—high tens and low hundreds of replicas.

First and foremost, mobile users require local replicas on their laptops. Once each user creates an additional replica by replicating onto his or her laptop, replication factors at least double and perhaps grow larger, depending on the size of the user base. For example, given two replicas of a spreadsheet in an office environment and twenty mobile users who require access to it, the replication factor of the object increases tenfold.

Replication factors can often be minimized in office environments due to LAN-style sharing and remote-access capabilities like NFS$^{\mathrm{TM}}$ [Sandberg *et al.* 1985]: one machine stores a replica that every machine in the office can access quickly, cheaply and easily. However, network-based file sharing cannot be utilized in mobile environments due to the frequency of network partitions and the wide range of available bandwidth and transfer latency. The network simply cannot be relied upon as a key component of the solution, further increasing the need for larger replication factors.

Second, we already have anecdotal evidence to support our claim that mobility increases replication factors. An informal survey of mobile users at Nomadic 96 indicates that over 71% of the users have files spread across more than one machine.[1]

Third, consider the case of appliance mobility. Our first argument above assumes that each user has one stationary machine and one mobile machine; however, the near-term future will see the use of many more "smart" devices capable of storing replicated data. Palmtop computers are becoming more common, and there is even a wristwatch that can download calendar data from another computer. Researchers [Terry *et al.* 1995, Want *et al.* 1995] have built systems that allow laptop and palmtop machines to share data dynamically and opportunistically. It is not difficult to imagine other devices in the near future having the capability to store and conceivably update replicated data; such devices increase replication factors

[1] The survey, conducted by Kleinrock, was admittedly an informal survey done without conducting random samples or utilizing typical statistical protocols. Nevertheless, the results still serve as anecdotal evidence concerning the trends in mobile computing. The survey was taken of approximately 100 people in attendance at Nomadic 96.

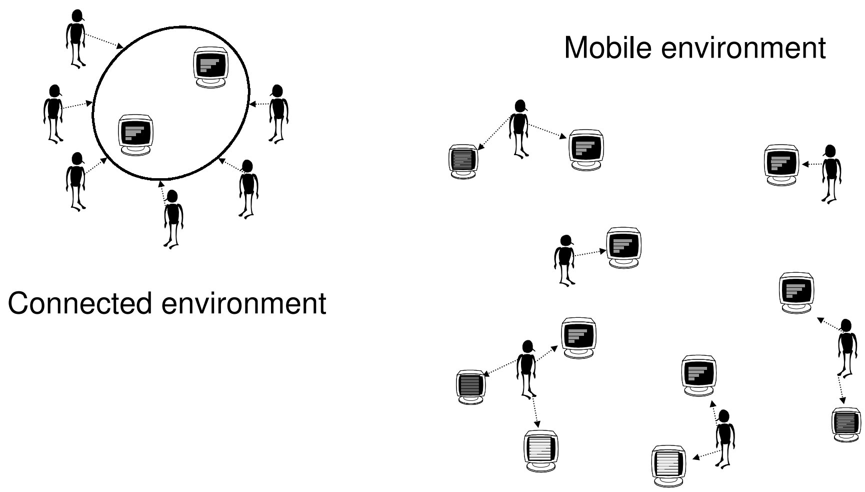

Figure 2.3: Mobility will increase replication factors. In the connected environment, two replicas can service all six users via network-based sharing mechanisms. In the mobile environment, not only can't the users rely on the network, but also the number of devices per user increases. In this example, the replication factors increase sixfold.

dramatically. The loss of the network and the addition of potentially multiple devices for each mobile user imply that replication factors could dramatically increase, potentially by an order of magnitude, as illustrated in Figure 2.3.

Finally, some have argued the need for larger replication factors independent of the mobile scenario, such as in the case of air traffic control [Quéinnec *et al.* 1993]. Other examples of environments requiring larger replication factors include stock exchanges, network routing tables and mechanisms, large database systems like airline reservation systems, and military command and control.

Read-only strategies and other class-based techniques cannot adequately solve the scaling problem, at least in the mobile scenario. Class-based solutions are not applicable to mobility, because, as described above in Section 2.2.2, they limit the abilities of the second-class citizens. The second-class citizens are generally the very same mobile machines that require the *greatest* level of functionality, not the least—for instance, clients cannot directly synchronize with other clients in the client-server model.

Read-only solutions are just another form of class distinction: they force users to pre-select the writable replicas beforehand. In general one cannot predict which replicas require write-access and which ones do not. Users will inevitably find themselves needing to generate updates but only having access to a read-only replica. For maximum flexibility and functionality, we must provide the *ability* for all replicas to generate updates, even though some may never do so.

Similarly, strategies that provide *upgradable read-mostly* replicas (upgradable on demand from read-only to read-write) are yet another form of class-based replication, and therefore equally not viable in the mobile arena.

Gray et al. [Gray *et al.* 1996] have devised analytical models of replication which they claim demonstrate that optimistic, peer replication cannot scale. However, their models are based on transactional databases, and therefore are substantially different from the target of this dissertation (the file system). Gray assumes that transactions cannot be coalesced into a single transaction or update; additionally, he assumes that the entire transaction must be transmitted and replayed at every replica. Both assumptions are incorrect for typical file systems. Multiple updates can be viewed as a single large update; in fact, RUMOR [Reiher *et al.* 1996] on UNIX™ coalesces all updates that occur between synchronization intervals into a single update and only increments the associated version vector once. Additionally, only the new value or resulting state need be transmitted to the other replicas—the entire "update" does not need to be replayed, as in transactional databases. Gray's result simply does not apply to the file system scenario.

Detailed controls over replication decisions

By definition a replication service provides users with some degree of replication *control*—a method of indicating what objects users want replicated at which sites. Many systems provide replication on a large-granularity basis, meaning that users requiring one piece of the replication container must locally replicate the entire container. Such systems are perhaps adequate in stationary environments when users have access to large disk pools and can effectively and efficiently utilize the network resources when important data cannot be stored locally. However, replication control becomes vastly more important to mobile users. Nomadic users do not in general have access to off-machine resources, and therefore objects that are not locally stored are effectively inaccessible. Everything the user requires must be replicated locally, which becomes problematic when the replication container is large.

Replicating a large granularity container means that some of the replicated objects will be deemed unimportant to the particular user, in that if the user doesn't access it, it is unnecessary to have it locally stored. Unimportant data occupies otherwise usable disk space, which therefore cannot be used for more critical objects. In the mobile context, important data that cannot be locally stored causes problems ranging from minor inconveniences to complete stoppages of work and productivity, as described by Kuenning [Kuenning 1997]. Kuenning's studies of user behavior indicate that the set of required data can in fact be completely stored locally, but only if the underlying replication service provides the appropriate flexibility to individually select objects for replication. Since it is *possible* to locally store all of the user's required data, and because the mobile user often *cannot* work if the data is not local, the replication service must provide replication control at a fine-granularity. Users and automated tools require fairly detailed controls over the set of replicated objects.

Pre-motion actions

One possible design point would have users "register" themselves as nomads for a specific time duration before becoming mobile. In doing so, the control structures and algorithms of the replication system could be greatly simplified; users would be generally stationary, and register their motion as the unusual case. For instance, suppose a user was taking a three-day trip from Los Angeles to New York. Before traveling, machines in Los Angeles and New York could exchange state to "re-configure" the user's portable to correctly interact with the machines in New York. Since replication requires distributed algorithms, part of the reconfiguration process would require changing and saving the distributed state stored on the portable to ensure correct algorithm execution. The reconfiguration process would represent the "unusual" case; the normal case would assume no motion, and could therefore afford to make great optimizations and assumptions in the underlying algorithms.

However, such a design policy drastically restricts the way in which mobility can occur, and does not match with the reality of mobile use. Mobility cannot always be predicted or scheduled. Often the chaos of real life causes unpredicted mobility: the car fails *en route* to work, freeway traffic causes unforseeable delays, a child has to be picked up early from school, a family emergency occurs, or weather delays travel plans. Users are often forced to become mobile earlier or remain mobile longer than they had initially intended. In general, we cannot require that users know *a priori* either when they will become mobile or for how long.

Variable	Assumption
Network partition frequency	Often
Length of partition time	Nearly infinite; no hard limit
Partition size	Arbitrarily large
Clock assumptions	Unsynchronized clocks
Replication model	Peer-based, but must scale well
Consistency method	Optimistic
Replicated data	File system objects
Update rate	Moderate
Update distribution	Moderate
Replication granularity	File granularity
Allowable replication factors	Hundreds
Number of system participants	Thousands/millions
Level of trust	High
Degree of mobility	Anyone can be mobile at any time
Length of mobile time	Infinite
Reliability	High

Table 2.1: The solution space and our position.

Additionally, the previous hypothetical design policy makes underlying assumptions about the connectivity and accessibility of machines in the two affected geographic areas: Los Angeles and New York, in the above example. It assumes that before mobility occurs, the necessary machines are all accessible so the state-transformation operation can occur. Inaccessibility of any participant in this process blocks the user's mobility. Such a policy seems overly restrictive, and does not match the reality of mobile use. Perhaps a user wants to change geographic locations precisely *because* a local machine is unavailable, or perhaps a user needs to become mobile at an instant when the multiple participants are not all jointly connected. Since neither mobility nor connectivity can be predicted, one cannot make assumptions on the combination of the two.

For these reasons, we believe that solutions that require "pre-motion" actions are not viable in the mobile scenario. Pre-motion actions force users to adapt to the system rather than having the system support the desired user behavior. Any real solution must provide the type of "get-up and go" functionality required by people for everyday use.

2.3 Solution Characteristics

Given the n-level solution space, the definitions of mobility and the requirements of the mobile user, we can now describe and defend our particular position in the solution space. We do so by explaining our position for each of the previous n independent variables (Section 2.1). The results are collected in tabular form in Table 2.1.

Network partition frequency: We assume that the networks are often partitioned. Mobile users are, by definition, often disconnected from the network. Wireless communication, while available in some areas, is neither globally available nor inexpensive, making it impractical to rely upon it for continual or efficient connectivity.

Length of partition time: We believe it is impractical and infeasible to set a hard limit on the length of time a given node can remain partitioned. Any hard limit may be exceeded by the atypical user, unless the hard limit is so large that it becomes ineffective as a limit. Since mobility and connectivity are unpredictable and often chaotic, we require algorithms that gracefully degrade as opposed to relying on hard time-outs. We therefore allow given nodes to remain partitioned almost indefinitely. Clearly, if they were partitioned forever, we would have no hope of ever maintaining global consistency, so we must assume that the partition eventual heals, although we cannot predict when this will occur.

Partition size: We assume and allow arbitrary partition sizes. A given laptop, when it disconnects, forms a partition size of one. However, if it encounters other laptops and forms a mobile workgroup, the size of the partition increases. More common, perhaps, is the failed router or gateway that isolates a group of machines from the rest of the global community. We would like the group to still function normally, and therefore must support arbitrarily large partition sizes.

Clock assumptions: We assume that the clocks are unsynchronized. Given the commonality of network partitions, the length of time that partitions may exist, and the number of system participants, it is simply infeasible to ensure clock synchronization. Systems like the Global Positioning System (GPS) [Herring 1996] may change this assumption in the future, but designing a solution for unsynchronized clocks forces the solution to be more robust in the face of clock-synchronization failures. Additionally, many independent sources, especially the military, are developing GPS jammers, which means that GPS may not always be available at critical moments. Solutions that rely upon it, perhaps for clock synchronization, are potentially doomed to incorrect behavior during critical periods.

Replication model: Mobile computing requires a peer replication model. Client-server and other class-based approaches simply will not work because clients cannot intercommunicate. Mobile users require the ability to directly synchronize with each other; otherwise the system model becomes inflexible and too expensive for effective mobile collaboration. Detailed motivation is discussed in Section 2.2.2.

Consistency method: Optimistic replication is required. Conservative schemes are not viable in environments with common network partitions and disconnections, because the conservative protocols (like majority-vote or primary-site) can rarely if ever be completed.

With intermittent network availability, conservative protocols degrade to only granting write access to at most one replica; sometimes no one can write. For instance, with a primary-site technique, only the primary site can generate updates when it is mobile and disconnected from the network. With a majority-vote technique, no one can generate an update if half or more of the replicas are inaccessible.

Therefore, for realistic sharing and use while mobile, we require an optimistic strategy. Optimistic strategies permit updates at every replica, and must later detect and resolve the potential problems caused by concurrent updates. Chapter 3 discusses optimistic replication further.

Replicated data: We decided to replicate file system objects, and therefore provide a replicated file system. Other types of data can be replicated using the file system, such as World Wide Web pages which, after all, are simply HTML documents in the file system.[2] Additionally, replicated databases can be achieved by storing the raw database data in the file system and dynamically converting it into and out of the database format, as demonstrated by Platinum *technology, inc.*; as a proof-of-concept, they combined Rumor, a database, and special-purpose database applications to form a replicated database. Finally, the mechanics and operational semantics of the file system are relatively simple and well understood, making it a desirable choice.

[2] Some Web documents such as those generated by CGI scripts are not real files, never reside in the file system, and are dynamically generated on demand. However, it is unclear what "replicating" such an object really implies, given that each generation of the object is potentially different.

Update rate: We assume that the load presented to the file service is that of "engineering/office applications" [Ousterhout *et al.* 1989], which consists of many applications using many small files, with a large degree of sequential read-write sharing but little concurrent sharing. This is considered the "standard" workload by most research file system designs.

Update distribution: Again, we assume that the file system load is typical of "engineering/office applications" [Ousterhout *et al.* 1989]. The update distribution therefore contains considerable read-write sharing, but little simultaneous sharing. We expect the updates to be spread across a number of different replicas, rather than having all updates applied to a single replica.

Replication granularity: Replication flexibility is extremely important to the mobile user. Since mobile users have less available resources than network-connected users, they must be more careful to ensure that the set of necessary data resides on their local machine.

We believe the file is the appropriate replication granularity in this environment. There are clearly other choices in the file system environment, such as directories, but the rationale behind namespace and semantic groupings often differs from that controlling replication requirements [Ratner 1995]. There is little necessity to replicate at any granularity smaller than a file, as the file object is typically treated as a unit, and a piece of a file is almost worthless to most applications. However, there are many scenarios that require replication on a granularity smaller than a directory—for instance, when a directory holds many types of semantically related objects (such as C++ source and derived object files) that should be replicated independently (user may want to replicate only the smaller, C++ source files). The file granularity therefore seems the appropriate choice.

Allowable replication factors: Widespread mobile computing will require larger read-write replication factors than previously provided by most systems, as explained in Section 2.2.2. However, massive scalability in terms of thousands of writable replicas seems like a solution in search of a problem. We believe there will be great need for a mobile replication service that can provide hundreds of read-write replicas: examples include large businesses, the military, and the other examples from Section 2.2.2. Additionally, such a service still performs well at a more reduced scale, such as that for a group of collaborating professors or private individuals. We therefore aim our solution at being able to support hundreds of read-write replicas, but do not see the need to support thousands.

Our solution does not provide read-only replicas. Read-only replication is fundamentally different from read-write replication in terms of control structure and system impact. Read-only replication could achieved with much simpler mechanisms than those proposed by this solution, and could be easily added on as an incremental improvement. Security policies [Reiher *et al.* 1993b] can be erected if the sole goal of the read-only replication is data protection and privacy.

Number of system participants: A major goal of this dissertation is to enable widespread mobile computing. As such, the system would not be a success if it limited the number of overall users. Roam is implemented at the user-level in a portable fashion, making it widely available and implementable on a number of operating system platforms. While the number of actual replicas of any given object is limited by the allowable replication factors discussed above, the system should allow thousands if not millions of participants to independently replicate different data sets.

Level of trust: Mobility makes computing inherently insecure. First, there are security issues with regard to data transport over insecure media, such as wireless connections. Second, because mobile machines change geographic locations, firewalls can easily be subverted and identities can more easily be masked. Finally, there is the security of the machine itself; a light-weight portable machine can be stolen or accessed much more easily than a large server locked in a secure room.

However, we believe that such security features can be adequately handled outside the domain of the actual replication system. Security policies may change over time or may differ between organizations; the replication service should not be tied to any specific security implementation. The replication service clearly needs to be designed with appropriate hooks for security policies and protocols, although it is not clear that intermixing the security functionality with the replication control structure yields the best design, especially when considering modularity arguments.

The dissertation therefore assumes a high level of trust between the actual replicas. It is assumed that the replicas are neither malevolent nor Byzantine; solutions such as TRUFFLES [Reiher *et al.* 1993b] can be used in conjunction to provide better security. This dissertation clearly acknowledges that data transport must always be protected, and provides for the encryption and authentication of all data traveling into and out of each replica, but leaves the higher-level security concerns to other modules.

Degree of mobility: The system must provide that anyone can become mobile at any time, without beforehand registering themselves as a "mobile" candidate or performing pre-motion actions. Mobility is often unpredictable: the network may fail and users may decide to simply work elsewhere. Users may be re-routed to alternate locations during the day by natural (e.g., weather), man-made (e.g., traffic, appointments, human-caused delays), or scheduled (e.g., meetings, business travel) occurrences. Having to decide each day, or each hour, whether or not one needs to become mobile in the next time quantum is tantamount to asking users to predict the future. The only reasonable default is to allow anyone to become mobile at any time.

Length of time one remains mobile: Similar to the argument in the "length of partition time" section above, we believe it is impractical to establish a hard limit on the length of time one may remain mobile. Mobility cannot be predicted, and often appears chaotic at best. Trips which may be expected to last for two days may be extended at the last minute; travel delays and connectivity outages may force users to remain mobile longer than previously anticipated. While users typically return to their "home" after some length of time, it is impossible, in general, to predict how long it will take and in fact if it will ever occur—some users may simply decide to alter their definition of "home." Instead of erecting hard limits, we desire a solution that provides correctness, good performance, and graceful degradation in all situations. We therefore establish no limit on the length of time one may remain mobile; it could, in fact, be infinite if users remain nomadic or constantly change their definition of "home."

Reliability Simply put, the system must be reliable. The failure of a given machine or set of machines should not affect the correct operation of the remaining machines.

2.4 Solution Summary

We have outlined the desired solution characteristics for a reliable, scalable, file replication system aimed at supporting mobile use. The particular point in the solution space is one which we believe addresses a real problem and has wide applicability, not just for replicating files in a mobile context but more generally for a wide range of replication-related problems, including but not limited to military cases, software development scenarios, distributed database problems like airline reservation systems, and general-purpose distributed computing. The remaining chapters describe ROAM's design and implementation, as specified by the above criteria.

Chapter 3

Background

This chapter provides the necessary background material and definitions required to fully understand the later chapters and overall system design. We provide details regarding optimistic replication and discuss FICUS and RUMOR, two earlier systems that are logical and intellectual predecessors of ROAM. The chapter defines a number of replication terms that will be used throughout the dissertation, and concludes with some comments about design assumptions, the development environment, and the development effort.

3.1 Optimistic Replication

Replication protocols can be separated into two categories based on the controls used to achieve global consistency. Conservative schemes utilize some form of locking, voting, or primary-site techniques to prevent conflicting updates. Conflicting updates include not only those that occur at the same physical time, but all updates that are generated without knowing the globally latest data version—the latter are said to occur at the same *virtual* time for concurrency purposes. Conservative schemes guard against all concurrent updates, but in doing so reduce data availability. If an update cannot be written because a lock cannot be obtained or a majority of other sites cannot be queried, the data object is effectively unavailable. Since disconnections and network partitions are normal occurrences in mobile computing, conservative strategies are not a viable solution.

The alternative is an optimistic approach. Optimism assumes that concurrent updates, or conflicts, are rare. It allows updates to be performed independently at any replica, thereby greatly improving availability. However, when conflicts do occur, special action must be taken to resolve the conflict and merge the concurrent updates into a single data object. The merging is referred to as *conflict resolution*. Studies and analyses of typical usage patterns have demonstrated that concurrent write-sharing is not very common, thus justifying that an optimistic approach is the right one [Kure 1988, Smith 1981, Kuenning *et al.* 1994, Wang *et al.* 1997]. When conflicts do occur, many can be resolved transparently and automatically without user involvement [Kumar *et al.* 1993, Reiher *et al.* 1994].

ROAM is based on an optimistic approach, and is the logical outgrowth of two earlier optimistic replication systems, FICUS and RUMOR.

3.1.1 Ficus

FICUS [Guy *et al.* 1990a, Guy *et al.* 1993] is an optimistic file system that provides data replication as well as transparent replica selection and remote access to non-local data. Based on the traditional peer-to-peer model, it allows any two replicas to directly synchronize with each other. It was originally designed using the volume as its unit of replication, although selective replication was later added [Ratner 1995].

FICUS maintains data consistency with two separate mechanisms. At update time, *update notification* messages are sent to all accessible replicas in a "one-time, best-effort" manner. A second process called *rec-*

onciliation runs periodically to guarantee consistency in the face of lost or undeliverable update notification messages.

FICUS is implemented in SunOS 4.1.1 and is designed using *stackable layers* [Heidemann *et al.* 1994]. As such, it is tightly integrated in the kernel, making it difficult to distribute and port to other operating systems.

Additionally, FICUS shares the scaling problems common to most peer models, and never experienced real use with more than a dozen replicas of any given data object. Nevertheless, FICUS is a real system with over 256 man-months of practical use and experience. Many of ROAM's distributed algorithms are either descendents or direct adaptations of the FICUS algorithms.

3.1.2 Rumor

RUMOR [Reiher *et al.* 1996, Salomone 1998] is the logical outgrowth of FICUS. It is the embodiment of the same optimistic algorithms and control structures in a user-level, file-system independent package. RUMOR's functionality is more limited than FICUS's, because it does not currently support either update propagation (real-time distribution of updates) or transparent remote access. Nevertheless, it does support the optimistic replication of data in a manner that is largely file-system independent [Salomone 1998]. The code base is separated into file-system independent and dependent portions. Porting RUMOR to a new operating system involves only rewriting the file-system dependent portion.

RUMOR maintains consistency entirely with a periodic synchronization process known as *reconciliation* [Reiher *et al.* 1996]. Rather than performing real-time update propagation like FICUS, RUMOR only maintains consistency at specific reconciliation intervals, or more generally whenever the user or the system initiates a reconciliation process. The tradeoff is one of system performance versus consistency. The FICUS approach attempts to maintain real-time consistency via update propagation, but in doing so degrades the performance of normal file system operations. The RUMOR approach imposes no performance impact on the native operating system, meaning that the user sees no performance penalty when reconciliation is not active. However, RUMOR loses the benefits of real-time consistency maintenance, which is often useful in closely connected LAN-style environments. ROAM follows in RUMOR's footsteps and utilizes the reconciliation-only approach, thereby guaranteeing good performance. Real time update propagation is much easier to enable in ROAM than RUMOR, and will most likely be enabled at some future point, as discussed in Chapter 11.

RUMOR, like FICUS, originally provided replication at the volume granularity. Selective replication was added later [Ratner 1997], as one stage in the ROAM redesign.

RUMOR is a real system in actual use, currently running on two different operating systems—Linux and WINDOWS®95. Where possible, ROAM was constructed using existing RUMOR code, to ease and speed development. As a side effect, ROAM therefore gains the benefits of platform independence. RUMOR is publicly available on the World Wide Web. [1]

3.2 Replication Models

There are three basic replication models that have been espoused and utilized in practically all replication facilities. These are the *master-slave*, *client-server* and *peer-to-peer* models. We discuss each with an eye towards their communication, synchronization, and update generation characteristics.

3.2.1 Master-slave

The master-slave model labels one replica the "master"; all other replicas are slaves. The replication paradigm is that slaves should always be identical to the master. The model is very simple, yielding a simple implementation, and has been used in many replication packages such as LAPLINK [Nance 1995] and

[1] RUMOR can be downloaded from `http://fmg-www.cs.ucla.edu/rumor`.

RDIST [Cooper 1992]. However, with the simple model comes limited functionality: the slave is essentially read-only. Most master-slave services ignore all updates or modifications performed at the slave, and "undo" the update during synchronization, making the slave identical to the master. While some provide crude abilities for some notion of update generation at the slave, all limit the set of operations that can be performed. For instance, object removal cannot be performed at the slave in LAPLINK, as the object will be reinstated by the replication service as part of making the slave identical to the master. In general, modifications can only be reliably performed at the master, and slaves must synchronize directly with the master.

3.2.2 Client-server

The client-server model is similar to the master-slave in that it designates one server which serves multiple clients. However, the functionality of the clients is greatly improved, and multiple inter-communicating servers are permitted. All types of data modifications and updates can be generated at the client. The client-server model has been successfully implemented in replication systems such as CODA [Satyanarayanan *et al.* 1990, Kistler *et al.* 1991] and LITTLE WORK [Honeyman *et al.* 1992].

The major drawback from the mobility vantage point is the communication restrictions placed on the client. Clients cannot intercommunicate or synchronize with each other, limiting their functionality when mobile. Two clients occupying the same hotel room or visiting the same remote location cannot directly exchange updates; rather, they must both communicate with the remote server in series, as one uploads its changes and the other downloads them. As discussed in Chapter 2, such a restriction is simply not viable in the mobile context.

Scaling is not typically a problem in the model since multiple servers can recursively be combined in a client-server arrangement. However, doing so increases the system's overall dependence on the servers at the upper levels of the hierarchy. Many client-server systems only allow clients to communicate with their "home" server. In such a scenario, the failure of a single server isolates all clients served by it, and the overall reliability of the system is impacted by the failure of a single server.

3.2.3 Peer-to-peer

The peer-to-peer model takes a very different approach from both the master-slave and the client-server models. The peer model is not class based: all replicas are equals, or peers. Any replica can synchronize with any other replica, and any file system modification or update can be applied at any accessible replica. The peer model has been implemented in systems such as LOCUS [Walker *et al.* 1983, Popek *et al.* 1981], BAYOU [Terry *et al.* 1995], FICUS [Guy *et al.* 1990a, Page *et al.* 1991], and RUMOR [Reiher *et al.* 1996, Salomone 1998] and has more generally been espoused in other distributed environments such as xFS in the NOW project [Anderson *et al.* 1995].

The peer model provides a very rich and robust communication framework. However, it has typically suffered from scaling problems. Peer models have traditionally been implemented by storing all necessary replication knowledge at every site; each replica thereby has full knowledge about everyone else, and synchronization and communication is permissible between any two hosts. Such an approach, however, results in exceedingly large replicated data structures (e.g., lists that contain the set of all known replicas) and clearly does not scale well. Additionally, distributed algorithms that determine global state must, by definition, communicate with or hear about (via gossiping) each replica at least once and often twice. Since all replicas are peers, any single machine could potentially affect the outcome of such distributed algorithms; therefore each must participate before the algorithm can complete, again leading to potential scaling problems.

3.3 Replication Definitions and Terms

For clarity of discussion, we define a number of replication terms that will be used throughout this dissertation. These are: replicas and machines, volumes, selective replication, version vectors, garbage collection, and synchronization.

3.3.1 Replicas and machines

A *machine* is any computer, portable or not. A *replica* is any copy of a data object—that is, a piece of data controlled by the replication service. A given data object may only have one replica, in which case it is the only copy in existence, or it may have `N` replicas, with `N` ranging into the low hundreds. Multiple replicas may exist on the same machine.

3.3.2 Volumes

The *volume* [Satyanarayanan *et al.* 1985] is a file system construct originally developed for the Andrew File System [Howard *et al.* 1988] but later used by replication systems such as CODA and FICUS. It is defined to be smaller than a disk partition but larger than a single directory; however, since it is a loose definition, there is typically no enforced maximum or minimum size. For example, a user's home directory and all sub-directories might constitute a volume. This volume clearly grows and shrinks over time, as the user adds and removes files. Additionally, the same sub-tree can be split into multiple volumes at any time.

The volume provides a large, coarse-grained container in which a collection of files can be stored and acted upon as a single unit. It is typically used because it provides several system-related benefits:

Locality and Performance: Logically-connected files are grouped together in one physical place. Performance-intensive tasks such as UNIX mounting can be executed once for the volume, rather than once per object in the volume.

Integrity: Volumes provide natural firewalls for preventing the propagation of errors and for establishing fundamental security barriers. Since a volume can be treated as a separate and complete entity, with only a single point of entry (the volume *root*), security barriers can be easily established at the single entry point, rather than individually protecting each volume member.

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

3.3.3 Selective replication

For a variety of reasons discussed more fully in Chapter 2, replication control needs to be independent of the layout of the namespace [Ratner *et al.* 1996a]. The motivations and rationale behind replication are often different from those that govern namespace issues. For instance, a given directory (or volume) may contain several different types of semantically related files, such as:

- source code and derived object and executable files
- sound, video, and related multimedia
- fonts, clip art, and other presentation accessories

While it makes sense to group these together in the same volume, users may want the flexibility to replicate only some of them, independent of the others; replicating more than one requires occupying extra disk space and costs time and money to maintain consistency. *Selective replication* [Ratner *et al.* 1996a, Ratner 1995] allows objects from the volume or other large container to be individually selected for replication purposes. Selective replication provides a much-improved user model, both in terms of functionality and performance. The details regarding selective replication are discussed in Chapter 6.

Vector 1	Vector 2	Result
{{1,1}, {2,5}}	{{1,1}, {2,4}}	Replica 1 has more recent data
{{1,1}, {2,5}}	{{1,1}, {2,6}}	Replica 2 has more recent data
{{1,1}, {2,5}}	{{1,1}, {2,5}}	Both replicas have identical data
{{1,2}, {2,5}}	{{1,1}, {2,6}}	Independent or conflicting updates
{{1,2}, {2,5}, {3,1}}	{{1,1}, {2,5}}	Replica 1 has more recent data
{{1,2}, {2,5}, {3,0}}	{{1,1}, {2,5}}	Both replicas have identical data

Table 3.1: Version vectors are indicated as tuples of {replica identifier, value}. In each case, the result of comparing the two vectors is indicated. Note that the result is achieved without actually analyzing physical file data.

3.3.4 Version vectors

Version vectors [Parker *et al.* 1983] are a method for tracking updates in a distributed system. They are utilized in FICUS, RUMOR, BAYOU, and ROAM; a variant is implemented in CODA.

The version vector is an array of length equal to the number of replicas. (Chapter 8 will discuss new methods of reducing the vector's length.) Each replica R_j maintains its own vector independently, and tracks in position `i` the number of updates generated by replica R_i that are known at R_j. In other words, each element is a monotonically increasing counter, and each counter `i` tracks the total number of known updates generated by replica R_i. Since each replica independently maintains its own version vector, each could temporarily have different values for `i`'s position, reflecting the fact that some replicas have more recent data than others. Eventually, when all replicas are consistent, all replicas will have the same value in every position in the vector.

Consistency is verified and maintained by comparing version vectors. By comparing any two vectors, the update histories of the corresponding file replicas can be compared, removing the need to compare actual data contents or rely on synchronized clocks. Table 3.1 illustrates an example.

As seen in Table 3.1, version vectors are compared in a pairwise fashion, matching elements of one vector with the corresponding element from another and comparing two matched elements. Matching is done by replica identifier so that both replicas compare elements for the same replica R_i. The comparison function (comparing version vectors `V1` and `V2`) has three possible output states:

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

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

`V1` and `V2` conflict : `V1` does not dominate `V2` and `V2` does not dominate `V1`.

Note that the above definition allows for two version vectors to dominate each other if they are equivalent element by element. This feature is often useful in peer-based systems, although such systems still differentiate between "equivalence" and true "dominance" before requesting and copying remote data.

When one version vector dominates the other, we maintain consistency by replacing file data and the version vector at the dominated replica with that from the dominating replica. That is, data and the associated version vector are copied from the dominating replica to the dominated replica and installed verbatim. When version vectors are found to conflict, their respective file replicas are said to be "in conflict." Special mechanisms must be invoked to resolve the conflict [Kumar *et al.* 1993, Reiher *et al.* 1994]. As replicas communicate and the pairwise version-vector comparison continues throughout all replicas, the most recent data propagates, and all replicas eventually converge to a common global state.

3.3.5 Garbage collection

Objects in the file system use system resources, such as disk space. When the user deletes an object, its resources must be deallocated for re-use by other objects. *Garbage collection* is the deallocation of the disk space held by deleted file system objects. While a relatively simple process in a centralized system, dynamic naming and potentially long-term communication barriers make garbage collection more difficult in replicated, distributed systems. CODA uses a simple log-based strategy that works well in specific scenarios but lacks generality and burdens the mobile user with conflicts as a side effect (described more fully in Chapter 10). FICUS and RUMOR use a fully distributed, two-phase, coordinator-free algorithm to ensure that all replicas are knowledgeable about the garbage collection process and eventually complete it, although any given set of participants may never be present simultaneously [Guy *et al.* 1993]. We build on this algorithm and improve its efficiency correctness, and mobile-friendliness, as discussed in Chapter 7.

3.3.6 Synchronization

We call the synchronization process *reconciliation.* It is a pairwise process between two replicas. The pairwise architecture is important and necessary, because in a mobile environment there are no guarantees that more than two replicas will ever be simultaneously available and accessible. (Synchronization, by definition, requires at least two replicas.)

All replicas need not be mutually accessible, and no given replica need be accessible at any given time. The only assumption imposed on the replicas is that they are *gossip connected.* Between any replicas $\mathtt{R}_i$ and $\mathtt{R}_j$, there must be a path of replicas $\mathtt{R}_1$, $\mathtt{R}_2$, ..., $\mathtt{R}_n$ such that $\mathtt{R}_i$ can directly communicate with $\mathtt{R}_1$, $\mathtt{R}_k$ can directly communicate with $\mathtt{R}_{k+1}$ for $k < n$, and $\mathtt{R}_n$ can directly communicate with $\mathtt{R}_j$. Furthermore, we allow the set $\mathtt{R}_1$, $\mathtt{R}_2$, ..., $\mathtt{R}_n$ to dynamically change and to temporarily be non-existent as network failures and mobile disconnections cause inaccessibility. However, we require that $\mathtt{R}_i$ and $\mathtt{R}_j$ are connected infinitely often by some set of third-party replicas. We call the communication of data from $\mathtt{R}_i$ to $\mathtt{R}_j$ via intermediary sources *gossiping.*

The particular communication patterns among the replicas (such as a ring or a star) form the *reconciliation topology.* While the algorithms discussed in Chapters 4 and 6 are topology independent, in that they are correct regardless of the actual communication patterns between replicas, the physical topology can affect both the number of messages exchanged between all replicas and the time required to reach global consistency.

3.4 Environment Assumptions

There are two key underlying assumptions in this dissertation that account for either design decisions or omitted functionality. These assumptions deal with aspects of computer communication. We believe the assumptions to be fairly general and standard; they are simply outlined here for completeness.

We assume an underlying transport mechanism capable of routing messages from computer `A` to computer `B` based only on the name of the machine. That is, we assume that messages sent from `A` to `B` will actually reach `B` if there exists a direct or indirect connection of intervening machines and networks. This dissertation does not discuss issues of how to route the messages or how to determine if such a connection exists; it assumes that if one does exist, the messages will reach their destination. Error detection and retries should be handled by the underlying network layer, and are not in the scope of this dissertation.

Additionally, we assume that computer communication while mobile is not free of monetary cost. Whether communicating by modem or wireless technology, we assume the mobile user pays a much higher cost per byte than when connected by Ethernet or a similar-quality service and typically experiences lower bandwidth and higher latency. If communication was always free to the user, and the bandwidth and latency was always acceptable, then there would be no point in performing replication; the data could always be accessed from a remote server on demand. We do not believe that communication is or will ever be free, and therefore see a continuing need for data replication.

3.5 Development Environment

ROAM was developed as part of the Travler project [Bagrodia *et al.* 1995] at the University of California, Los Angeles. It was developed in C++ with occasional Perl scripts. ROAM is a system in real use; in fact, ROAM's source code and executables are replicated using ROAM.

Obtaining a total figure of the number of lines of code is difficult, because ROAM's development took place over many months, during which the developer was simultaneously making modifications and fixing bugs in RUMOR. As a result, a sizeable amount of ROAM-specific code became part of standard RUMOR early in the development cycle; for instance, selective replication was implemented in RUMOR largely because of ROAM's requirement for it. In the same way, RUMOR maintenance and modifications were often performed in a manner that would facilitate ROAM's development down the road. For example, when the version vector class was redesigned, it was done with the forethought that new algorithms would later be implemented. Therefore, obtaining an actual lines-of-code figure is difficult, but it is estimated to be approximately the following:

- 3500 lines of C++ code for the selective replication implementation, plus 1000 lines of Perl code for selective replication utilities. The majority of the 3500 lines of code are platform-independent. Less than 10% of the C++ code is located in the file-system dependent version of the RUMOR architecture.
- 16,000 lines of C++ code to implement the Ward Model, the majority of which is file-system independent.
- 8000 lines of C++ code to implement the new server architecture, although this number under-represents the actual size of the server, because we made use of a large quantity of original RUMOR code.
- 200 lines of C++ code to implement the new garbage collection algorithms.
- 400 lines of C++ code to implement the new version vector management algorithms.
- All together, the entire source tree is approximately 79,000 lines of code. This figure includes the above figures and original RUMOR code as well, some of which was modified or enhanced during development.[2]

[2]The size of the project has caused some to comment that ROAM wasn't built in a day.

Chapter 4

Ward Model Design

Chapter 2 describes the need for a replication model that scales well and supports any-to-any communication. Traditional client-server architectures are not capable of supporting the desired infrastructure and providing the required communication capabilities among replicas. While such functionality could have potentially been added to the client-server model, we expected that modifying a fundamentally client-server design to incorporate a rich communications structure would require such basic structural changes as to imply effectively starting from scratch. Additionally, allowing clients to inter-communicate seems to violate basic invariants of the client-server model. Building upon the peer model seems an esthetically cleaner approach, and ought to better preserve the pureness of the underlying framework.

The goal is therefore to design a peer-based model that scales well. Replication services based on the traditional peer model, such as FICUS [Guy *et al.* 1990b], RUMOR [Reiher *et al.* 1996], and BAYOU [Demers *et al.* 1994], all suffer from scaling problems. In response, we have designed the *Ward Model*. It provides a new and different form of the peer model, one based on a hybrid between client-server and peer solutions that clusters replicas into groups without affecting the underlying any-to-any communication capability between all system participants. The result is a replication model that provides both the desired functionality and scaling ability.

This chapter describes the overall architectural design of the Ward Model. We first describe the model as a static architecture, and then describe the support for mobility and mobile operation. We then discuss the model's characteristics and its scalability. We conclude with some implementation details.

4.1 General Approach

We would like to combine the advantages of the client-server and peer models while minimizing their disadvantages. We do so by formulating a hybrid model. Additionally, hierarchical models are a tried and true method of improving scalability: clustering objects into groups, and having one group member represent its entire group in a meta-group. We employ a similar approach here. Our groups are called *wards*, for Wide Area Replication Domains. The *ward master* is the one ward member responsible for representing the group in the higher-level meta-group. Although we distinguish between two types of replicas within the ward (member and ward master), we maintain a peer model between all replicas. Other than the additional responsibilities and the data structures required to carry out those responsibilities, the ward master is equivalent to any other ward member; in fact, the ward master's duties can be trivially moved from one ward member to another as needs require. The distinguishing differences between the ward members and the ward master are only in name and additional responsibilities, rather than actual control mechanisms. Overall, the code executed by the ward master compared to that of the other ward members differs by less than fifty lines of C++ code.

The description of the Ward Model is layed out as follows. First, we discuss a very simple approach, the basic Ward Model, and define wards, ward masters, and the synchronization architecture in a simplified form.

We then enhance and elaborate upon the definitions and describe the advanced model. Our implementation in Roam supports the advanced model.

4.2 Basic Ward Model

We describe the basic Ward Model by defining wards, ward masters, and ward sets. We then explain both how the model maintains consistency and how the model supports mobility.

4.2.1 Wards

The key idea behind wards is to group volume replicas into containers that capture the notion of common or typical communication partners. For example, given four replicas in Los Angeles and four replicas in New York, the system would perform poorly if each replica in Los Angeles typically synchronized with a replica in New York. While the topology produces correct results, each machine pays the additional cost of long-distance communication, in terms of latency, efficiency, and price per byte. Additionally, at synchronization time, there is a greater chance of a failed network connection between Los Angeles and New York than between two local partners in either location, given that generally the long distance communication depends on having the local communication operational. The obvious improved approach is to have one candidate in Los Angeles communicate with one candidate in New York; these two candidates afterwards disseminate the information among their local colleagues.

Given that synchronization should typically occur with a local partner, both for economic and efficiency reasons, we would like to group replicas together to capture the notion of synchronization locality. We therefore build wards as a collection of "nearby" volume replicas, the details of which are discussed below. [1] The ward members are only required to be loosely connected—continual, high-quality connectivity is not necessary.

For maximum flexibility, each shared volume of data has its own ward. Thus, if a given machine stores four replicated volumes, it will be a member of four different wards, which may or may not have intersecting membership depending on the replication factor of the volume itself. An alternative approach would combine all volumes on all nearby machines and define a ward as the set of machines and the volumes which they store. While the alternative approach captures the desired idea of synchronization locality, it lacks the flexibility of our original definition. For instance, given three replicas of volume V_1 in Los Angeles, Santa Barbara, and New York, we may wish to group the replicas into two wards: one for Southern California (Los Angeles and Santa Barbara) and one for New York. However, given a second volume V_2 with twenty replicas in each Southern California city, we would probably want separate wards for Los Angeles and Santa Barbara, as demonstrated by the example in Figure 4.1. The existence of V_2 should not impact our ward decisions for V_1; otherwise, the resulting model always caters to the lowest common denominator, incurring performance problems for the other volumes. We therefore allow the wards designated for volume V_1 to be entirely distinct from those designated for V_2.

As with any grouping mechanism, there must be a method for assigning members to groups, as well as an enumeration of the basic abilities and characteristics bestowed upon group members. Both are described below.

Determining ward membership

A primary criteria governing ward membership is the expected connection *quality* between members, where quality encompasses the available bandwidth, connection latency, expected network connectivity, and the cost of the network. Ideally, a decision regarding grouping should be based on several criteria:

[1] A true definition of "nearby" is difficult, and depends on many complex factors, such as sharing patterns. We incorporate some characteristics of what it means for one replica to be "near" another; others are discussed as future work in Chapter 11.

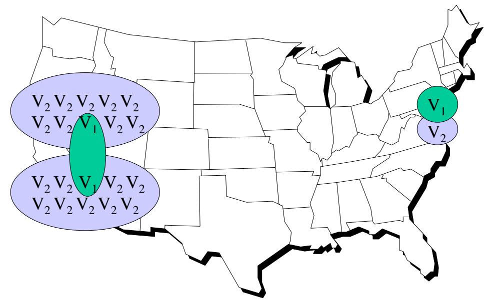

Figure 4.1: Wards are structured on a per-volume basis for maximum flexibility in selecting wards and ward members. Here we make two different ward decisions for the replicas of two different volumes $\mathtt{V}_1$ and $\mathtt{V}_2$. Wards are indicated by circles around $\mathtt{V}_1$ and $\mathtt{V}_2$; each $\mathtt{V}_i$ indicates another replica of volume i.

Geographic location: Machines that are physically near each other should typically be grouped in the same ward. To a first approximation, geographic co-location captures the notion of locality that underlies the ward definition. Since communication with a nearby partner is easier and more efficient than communication with a distant one, nearby partners are the "typical" good synchronization choices.

Of course, geographic co-location has no direct correlation on the experienced connection quality; it is only a simple heuristic. Therefore, the following parameters should also be taken into account.

Expected bandwidth: Machines that are connected by high bandwidth links should be grouped together in the same ward; machines connected by low bandwidth networks should be in separate wards.

Connection latency: Machines connected by low latency connections should be grouped together in the same ward; those connected by high latency connections should be in separate wards.

Expected network connectivity: Machines that are "typically" connected should be in the same ward; machines typically disconnected should be in different wards.

Network cost: Machines connected by an inexpensive or free network should be grouped together in the same ward; those connected by expensive communication media should be in separate wards.

Additionally, the ideal grouping mechanism would dynamically track and change ward membership when the above parameters change. However, the existing implementation does not automatically determine and dynamically reconfigure ward membership. Currently, the task of deciding what ward to join when creating a new replica belongs to the system administrator or knowledgeable user creating the replica. When creating a new replica, the user decides if he or she wants to join an existing ward or form a new one. The decision can of course be altered later by using the utilities to change wards (Section 4.2.4). This dissertation does not address the issue of optimal ward placement, although the topic is discussed as future work in Chapter 11.

Ward characteristics

All ward members are peers in the traditional peer-model sense. As a result, any ward member can directly synchronize and communicate with any other ward member. The communication flexibility is in stark contrast to client-server solutions that cannot support client-to-client communication and are therefore severely impacted by an inaccessible server. Since all ward members are peers, the Ward Model has no bottleneck with respect to intra-ward communication and synchronization.

Each ward has a *ward master*. The ward master serves as the doorway between wards; it is responsible for maintaining consistency with the other wards. The ward master is similar to a server in client-server terminology in that its duties are similar, but there are several important distinctions:

- Any ward member can serve as the ward master. Since all ward members are peers, anyone can perform the duties of ward master. The role of ward master can easily be shifted to any other ward member. Additionally, ward master *re-election* algorithms, in the case of ward master failure, are straightforward to perform and execute. In client-server models, the clients cannot assume the role of the server, and the model is fundamentally broken if the server fails.

- The ward master need not physically store all data objects that are replicated within the ward. Rather, the ward master need only be able to *name* the objects. In the basic ward model, naming the objects is equivalent to naming the volume itself, but the issue of identifying the set becomes more complex in the advanced Ward Model. In a client-server model, the server must store a complete super-set of all objects.

  When the ward master does not physically store a particular object, it is said to store a *virtual* replica. For synchronization purposes it "pretends" to store a replica, but whenever it requires actual data the ward master obtains it from a physical replica within the ward.

- The ward master is not the central bottleneck with respect to intra-ward synchronization. Since any ward member can directly synchronize with any other ward member, the intra-ward synchronization architecture has no central bottleneck. In a client-server scenario, the server is the central synchronization bottleneck.

As the only member of the ward with knowledge of replicas outside the ward, the ward master is responsible for synchronizing all intra-ward data with the other wards. The ward master effectively belongs to two wards: the local ward and a "higher-level" ward that consists of all the ward masters. In this sense, the ward master can be considered a "super-replica." Semantically, the model collapses the entire ward into a single replica—the ward master—for inter-ward synchronization purposes. Section 4.2.3 describes the synchronization details in depth.

4.2.2 Ward set

The *ward set* refers to the set of replicated data stored within the ward. In the basic Ward Model, the ward set is, by definition, equivalent to the entire volume, although the definition changes in the advanced model. Like the volume itself, the ward set is dynamic in character; it changes as the volume itself changes in response to new object creations and existing object deletions.

It is the ward master's responsibility to synchronize its ward set with the ward sets from other wards. In the basic Ward Model, ward-set synchronization can be accomplished simply by contacting one other ward master: since all wards store the entire volume, all ward sets are identical. In the advanced model a more complex synchronization architecture will be utilized.

4.2.3 Maintenance of consistency

Consistency is maintained simultaneously both within each ward and among wards. In both intra- and inter-ward scenarios, the consistency *topology* refers to the communication pattern used between replicas. All of our consistency algorithms are topology-independent. Their correctness does not depend on the pattern of communication, but only requires that information can flow from any ward member to any other ward member in a finite number of steps and through a finite number of other replicas.[2] However, different topology patterns yield different results in terms of the performance of distributed algorithms and message volume complexity. For example, a quadratic message complexity results from an all-pairs reconciliation topology, but a ring (using a gossip-based transfer of information) reduces the message complexity to a linear cost. A superior messaging plan avoids the quadratic cost when inter-site communication is available, but gracefully handles degraded communication.

Using two separate topologies within the ward and between wards reduces the generality of the model and increases the requirement for special-case code. Although one could potentially identify hypothetical reasons why two topologies would be required, a topology that applies equally well to both scenarios is clearly a better choice for generality and simplicity.

We have chosen an *adaptive ring* topology, both within and among wards. We will first define the topology and then describe its implementation. The actual details of synchronization (the consistency mechanism that uses the topology) is discussed in Chapter 5.

Adaptive-ring topology

We identify the set of replicas in the ward as the set Φ. In a ring topology, participants reconcile with the "next" member in Φ. We order the set Φ by replica identifier; our definition of "next" is the "next smallest." Each replica therefore synchronizes with the site corresponding to the next smallest replica identifier.

We make the ring topology *adaptive* in two ways:

1. The ring is dynamically reconfigured when the set Φ increases or decreases in size.

2. During periods of network failures, a given replica $\mathtt{R} \in \Phi$ reconciles not with the next replica $\mathtt{R}-1 \bmod |\Phi|$ but with the next *accessible* replica $\mathtt{R}-e \bmod |\Phi|$ where $e \in \{1, ..., |\Phi - 1|\}$.

The adaptive ring requires only a linear message complexity of $O(|\Phi|)$ messages to propagate information to all replicas. As illustrated by the second adaptation above, the ring adapts itself to changing network topologies and is therefore quite robust. Replicas gossip about the status of other replicas: that is, a given replica transmits all information it has received about the state of other replicas to those which follow in the sequence. Thus, the topology does not require point-to-point links interconnecting all replicas. The adaptive ring allows rarely or never-communicating sites to share data by relying on third-party replicas to gossip on their behalf. For these reasons, it is an attractive topology to consider, and one that applies equally well both within and between wards. Figure 4.2 illustrates the adaptive ring topology both within and between wards.

Adaptive ring implementation

We use the adaptive ring topology both within and between wards. Each volume replica, regardless of its ward membership, is assigned a globally unique replica identifier that is guaranteed never to be reused. Using these identifiers, we generate adaptive rings within each ward and between the ward masters. When a new replica joins the ward or an existing member leaves, we use their unique replica identifier to dynamically adjust the adaptive ring.

The adaptive ring topology is not statically stored at any given replica. Instead, when a replica decides to synchronize, it automatically and dynamically builds its local portion of the adaptive ring using local data regarding the other ward members. Using local data may occasionally result in a sub-optimal ring, because information regarding ward membership propagates lazily throughout the ward. For instance, assume the ward contains replicas 2, 4, and 6. If replica 5 joins but only replica 4 is knowledgeable of 5's existence, then replica 6 will synchronize with replica 4, which 6 *believes* to be the next replica in the ring. However, as part of 6's synchronization it learns of 5's existence, so the ring automatically "heals" itself as the new information propagates. Since the underlying algorithms are topology independent, correctness is not affected by temporary spokes on the adaptive ring. The example is illustrated graphically in Figure 4.3.

[2] Ward is used generally here, and also refers to the meta-ward that contains all ward masters.

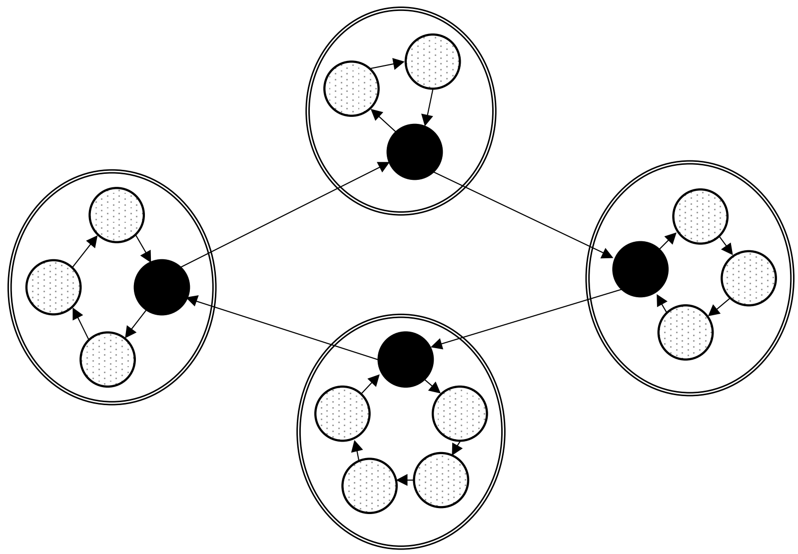

Figure 4.2: The basic adaptive ring topology, both within and between wards. Ward masters are illustrated as solid black replicas; wards are double circles surrounding the replicas. Arrows indicate synchronization paths, although the synchronization topology adapts itself to network topology, as explained in Section 4.2.3.

4.2.4 Support for mobility

The model provides support for *intra-ward mobility* essentially for free, since everyone in the ward is a peer in the traditional sense. Intra-ward mobility occurs anytime the user is mobile within a restricted geographic area and is therefore only likely to encounter other machines from the same ward. For instance, moving within one's office from the desk to the couch demonstrates an intra-ward mobile action, as does moving around the office or within town to the local coffee shop. In all examples, the degree of motion is large enough to potentially change the set of "best" or most efficient communication partners, but not large enough to bring the user outside of his or her ward. For example, two colleagues in the office may never directly synchronize, typically relying on a set of third-party replicas to relay updates. However, if they meet at someone's home to discuss plans for the following day, they will usually want their machines to directly communicate.

Intra-ward mobility is an important case, because it happens frequently and certainly more frequently than long-distance geographic motion. As such, support for it should be tremendously fast and inexpensive. The Ward Model fundamentally incorporates the notion of intra-ward mobility as a basic ability and common occurrence in its construction of wards and assignment of ward membership.

Of course, we must also allow the user to travel outside the ward and directly synchronize with machines from other wards. While *inter-ward mobility* does not occur as often as intra-ward mobility, when it does occur the costs associated with synchronization can be much greater. Without the ability to dynamically change one's synchronization partners and adapt to the resources (i.e. replicas) found in the new geographic area, the users are forced to communicate over long distance links back to their "home." Long distance telephone calls cost more than local ones and long-distance connections typically exhibit poorer quality in terms of less bandwidth and higher latency. Inter-ward mobility can therefore be considered the fundamental problem with regard to mobility, as solutions that do not adequately and efficiently solve it do not address the hard problem in the solution space. Inter-ward mobility is of extreme importance to the mobile user,

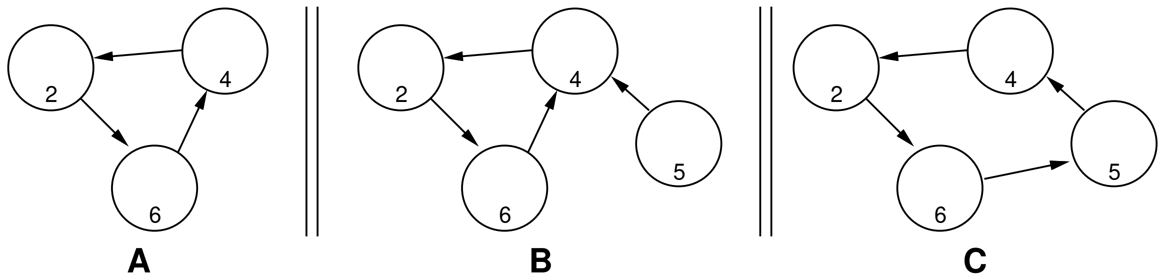

Figure 4.3: When new replicas join the system, the adaptive ring has temporary "spokes" until the information regarding the new replica is propagated to everyone. In situation `B`, replica 5 joins the system, known only to replica 4. Once replica 6 learns of 5's existence, in situation `C`, the adaptive ring correctly re-forms.

because without it the costs associated with mobility can outweigh the benefits associated with replication, and at the very least the high costs create headaches and heartaches for users.

In the basic Ward Model, there is only one form of inter-ward mobility or *ward motion*. (In the advanced model we will introduce a second form.) Inter-ward mobility is enabled through a mechanism called *ward changing*. Ward changing allows a given machine to physically change its ward membership on a volume-by-volume basis. When a machine `M` from ward W_1 wants to directly communicate with a machine from ward W_2, perhaps as a result of physically moving to a different area, `M` withdraws its membership in W_1 and becomes a member of W_2. As a member of W_2, `M` has all the standard rights and privileges, including the ability to directly synchronize with the other members of W_2. Should `M` later move either to a different ward altogether or back to W_1, it again performs the ward change operation. Withdrawing oneself as a ward member requires contacting only one other member of the original ward; the information propagates lazily and asynchronously throughout the original ward. If no members of the original ward are accessible, the withdraw operation can be delayed until a member of the original ward becomes available. Correctness is not impacted by either the timing of the withdraw or the speed at which it propagates throughout the original ward. The only cost associated with delaying the operation is that caused by maintaining the additional data structures at the moving replica and stalling the intra-ward distributed algorithms in the old ward.

In the case where `M` really just wants a one-time direct communication with someone from another ward, we provide a special synchronization option as an optimization. The optimization allows the user to perform the synchronization without the cost of dropping, adding, and then again dropping ward membership.

4.3 Advanced Ward Model

In the advanced Ward Model we introduce selective replication, the ability for a ward member to physically store only select portions of the complete volume. The details concerning how the reconciliation algorithms and controls handle selective replication are discussed in Chapter 6. Here we describe the changes selective replication makes to the ward definitions, controls, and constructs.

4.3.1 Wards

The basic definition of a ward remains unchanged. The ward is still a collection of "nearby" volume replicas. However, all replicas need not store the same portions of the volume, which impacts both the ward set and synchronization topologies, as described in the following sections.

The characteristics and abilities of ward members similarly remain unchanged. Specifically, any-to-any communication is still enabled between any two ward members. However, due to selective replication, any-

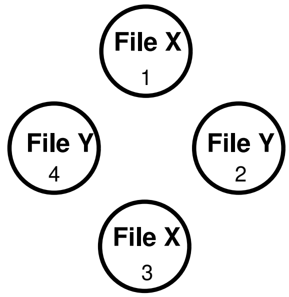

Figure 4.4: Selective replication's impact on any-to-any communication. Replicas are indicated by circles with their replica identifier.

to-any communication between all replicas may not always make physical sense for any particular layout of the file objects onto the set of volume replicas. For example, consider the scenario in Figure 4.4. If the only two objects in existence were files `X` and `Y`, direct communication between replicas 2 and 3 is meaningless, because 2 and 3 do not have anything in common to talk about. However, we must support the underlying *ability* for any-to-any communication, because if replica 3 later adds `Y` or replica 2 adds `X`, then clearly the two replicas would want the ability to directly communicate.

It is clearly possible to force the intermediary replicas to store file data, either temporarily or permanently, for the sake of gossiping. For instance, we could force replica 2 to store a copy of `X`'s data solely for relaying updates between replicas 1 and 3. Doing so, however, nullifies two key advantages of selective replication: namely, saving disk space and reducing synchronization time. Therefore, while we still preserve the ability for any-to-any communication, it is only useful in the context of two replicas storing common data objects. More formally, let the set of data locally stored by a given replica be called the *replica set*. Any-to-any communication is only meaningful for replicas within the ward whose replica sets intersect.

4.3.2 Ward set

The ward set is defined to be set of replicated data stored within the ward. In the basic Ward Model the ward set is equivalent to the volume. However, selective replication allows each ward member to store select portions of the volume, meaning that the set of data stored at each replica may be smaller than the entire volume. The ward set itself, therefore, may be smaller than volume, though of course it can never be larger. It is equivalent to the union of all replica sets for all replicas in the ward.

The ward set changes dynamically as the set of data stored within the ward changes. For instance, if a given ward member uses the selective replication controls to locally add a new file system object, the ward set expands to include this new object. The ward set similarly decreases in size when replicas locally drop file replicas. Additionally, ward motion can change the ward set, as replicas move into and out of a given ward, as described below in Section 4.3.5.

4.3.3 Ward masters

The ward master is responsible for the inter-ward synchronization of the entire ward set, and therefore must be able to identify the complete ward set. Use of both selective replication and ward changing, however, can change the ward set. Since any ward member can dynamically and optimistically change its replica set using the selective replication controls, and because new machines can at any time move into the ward (carrying with them their accompanying replica set), the ward set changes dynamically, lazily, and without global coordination. Without selective replication the ward set could be identified simply by naming the volume. With selective replication, however, the best mechanism for the ward master to identify its complete ward set is to individually name all objects in it.

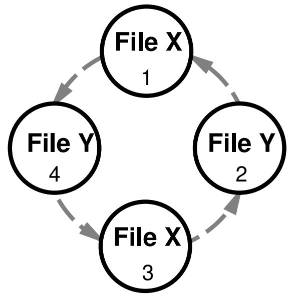

Figure 4.5: Selective replication's impact on the simple adaptive ring topology. Replicas are indicated by circles with their replica identifier; the adaptive ring is indicated by the series of dotted arrows.

Having to individually name all objects is not a ward-specific phenomenon. Any volume-granularity replication facility can name a set of data by naming the volume; a system providing selective replication, such as Ficus and Rumor before Roam, requires each replica to individually list the objects it locally stores [Ratner 1995, Ratner *et al.* 1996a]. Individually naming the elements of the ward set is simply the higher-level equivalent of individually naming the elements of a replica set. We discuss the issue of naming the ward set, possible alternatives, and the impact naming has on the scalability of the entire system in Section 4.6.

Finally, we would like selective replication to help alleviate disk storage at the ward master. A ward member that wants to store a 100MB file should not necessarily force the ward master to also store that 100MB file solely for inter-ward synchronization purposes. We allow the ward master to store only a *virtual* replica—the equivalent to naming the object without storing file data. When data is required for synchronization purposes, the ward master relays it via the physical data site.

4.3.4 Maintenance of consistency

Since each replica in the ward has a potentially different replica set, it follows that each ward master has a potentially different ward set. Therefore, in both intra- and inter-ward synchronization, we must use a more robust topology than a simple adaptive ring. Consider again the example of Figure 4.4 along with the simple adaptive ring topology from the basic Ward Model, illustrated in Figure 4.5. Using this topology, updates to file `Y` can only propagate between replicas 2 and 4 when replicas 1 and 3 are inaccessible. Therefore, if replicas 1 and 3 are almost always accessible, synchronization on `Y` would almost always be broken. A similar situation exists for file `X`. We therefore augment the basic adaptive ring to account for the differences between replica sets and ward sets. When the ward master is viewed as a "super-replica," the ward set appears exactly the same as a replica set for a super-replica, meaning that one approach can again be used in both intra- and inter-ward synchronization.

The solution uses an adaptive ring for each file object, rather than one for the whole volume, and then coalesces multiple per-file rings into a single ring based on the intersection between replica sets [Ratner *et al.* 1996a]. The approach has previously been implemented and used for selective replication synchronization in both Ficus and Rumor. The full details are described in Chapter 6; we provide only a brief overview here.

When a given replica decides to synchronize its local data, it dynamically calculates its local portion of the adaptive ring for each object it locally stores. As in the volume adaptive ring, each file's adaptive ring is calculated using local data regarding which other sites store replicas. While the location data may be outdated, since it is maintained optimistically, it is updated as a result of the synchronization process itself, so the per-file ring automatically "heals" itself. As before, the ring is dynamically computed using each replica's unique replica identifier as a global ordering. (Pictures and more details are provided in

Chapter 6.)

Multiple per-file adaptive rings are coalesced into one for improved performance and batch communication. When a replica R decides it shares at least one object with a second replica S, the two communicate about all objects in the intersection of their replica sets. All objects that they both physically store are synchronized in one "synchronization action." The volume adaptive ring is therefore a special case of the per-file adaptive ring that occurs when all sites store all objects (i.e., when all replica sets are equivalent, or selective replication is not used).

A given replica may have to synchronize with multiple other sites before it can synchronize all of its local data. After synchronizing with one host, it looks at its replica set, decides what objects haven't yet been synchronized, inspects the per-file adaptive rings for these objects, coalesces the rings for batch communication as appropriate and synchronizes with the next available candidate. The process continues until all local data has been synchronized or the set of available candidates is exhausted. That is, replica R synchronizes with a set of replicas S_1, S_2, ..., S_n along the ring topology such that R's replica set is a subset of the union of the replica sets for S_1 through S_n.

4.3.5 Support for mobility

Selective replication introduces new difficulties for the ward motion algorithms. In the basic Ward Model, a machine moving into a new ward is guaranteed to have the same replica set as the new ward's ward set, because all participants store full volumes. Since the ward set is equivalent to the moving replica's replica set, it is straightforward for the new machine to integrate with the new ward. With selective replication, the mobile machine's replica set may differ from the new ward's existing ward set. As a result, the advanced Ward Model requires more rich and robust ward motion algorithms.

The ward master is responsible for the inter-ward synchronization of all intra-ward data. When a new machine enters the ward and brings with it data files not in the current ward set, there are two options: either the ward set expands to incorporate the new data objects, or it doesn't. In the former case the ward set changes, possibly causing changes at other ward masters since they must keep track of what is stored at the various ward masters to properly form their adaptive per-file rings. In the latter case the ward set remains unchanged, and there are no ripple effects affecting the other ward masters, but the mobile machine cannot synchronize all of its data completely within the new ward. Some of the data must be synchronized with another ward, most likely the original ward that the mobile machine came from. The two possibilities are summarized graphically in Figure 4.6.

To properly decide which option is best, we must look at how physical motion actually occurs. Real mobility seems to occur in one of two modes:

visit: a temporary trip to a remote location, measured in hours or a small number of days

long stay: a longer stay at a remote location, perhaps for "a while," perhaps more permanently

Each mode has accompanying expectations of cost and performance. Users expect a temporary mobile move to be lightweight and inexpensive: since they're not planning on staying very long, they don't want to pay a large up-front cost. Additionally, users will generally accept sub-optimal performance, given that they know the motion is temporary and the up-front cost is minimal. On the other hand, users moving for longer periods of time are generally willing to pay a more expensive up-front cost to gain better performance. Since they know they will remain at the remote location for a long time, they want good performance while there. The up-front cost is amortized over the length of the stay; users staying a short time do not gain significant benefit from the large cost, and are therefore generally unwilling to pay it.

Real motion, of course, occurs over a continuous time spectrum, and doesn't always fall exactly into one of the two classifications. However, as a general paradigm the classifications seem to work fairly well, especially since positions in the middle of the spectrum can essentially be placed in either category.

Given the two types of physical motion, and their accompanying performance implications, we need to design ward motion to match the reality of physical motion and to behave the way users expect. The advanced Ward Model therefore provides two forms of ward motion, unlike the basic Ward Model that

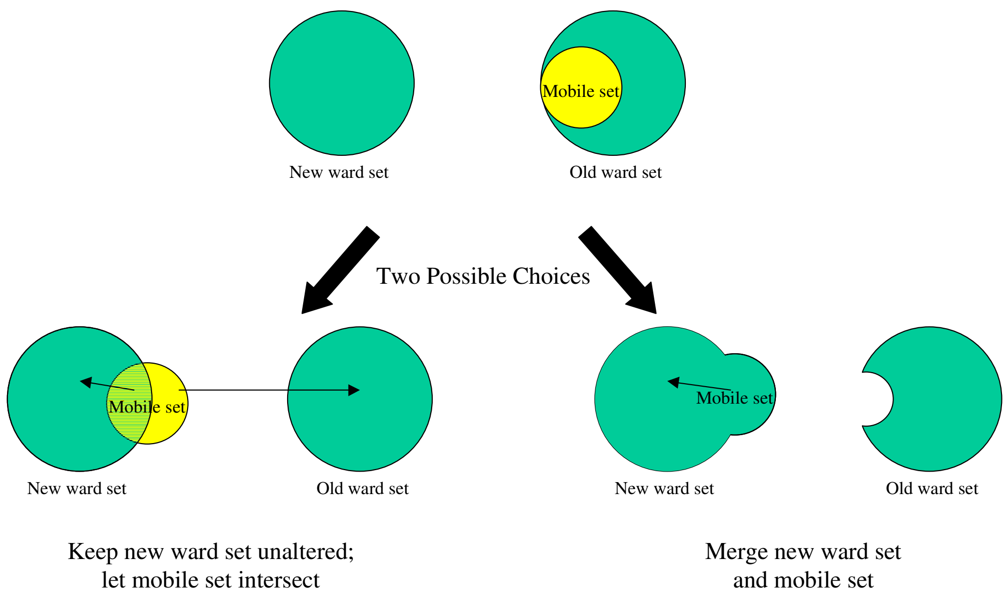

Figure 4.6: The effect of mobility on the ward set. In both cases the mobile machine with its associated replica set (labeled mobile set) moves from the old ward to the new ward. If we assume the mobile set and the new ward set have a non-empty set difference, then there are two possible solutions. In one case, the new ward set expands and the mobile replica can synchronize all of its data within the new ward. In the other case, only part of the mobile set intersects with the new ward set. The intersection can be synchronized locally in the new ward, the set difference cannot. Arrows indicate the required synchronization paths.

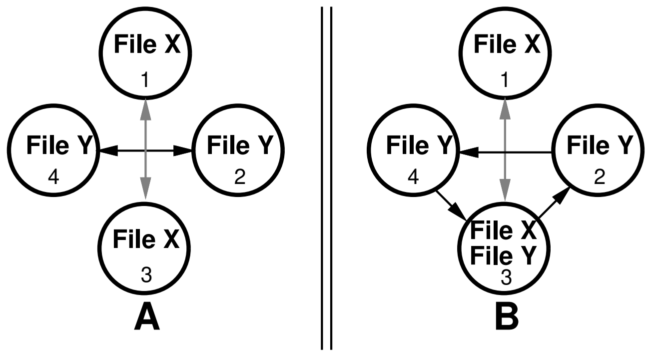

Figure 4.7: New topology information propagates on a "need to know" basis. Replicas 1,2,3,4 could be ward members indicating the files they store; alternatively, they could be ward masters indicating their ward sets.

provides only one. Ward *changing* is used during long stays, while ward *overlapping* is used during visits. Both are now described in more detail.

Ward changing

Ward changing involves a long-term, perhaps permanent, change in ward membership. The moving replica physically changes its notion of its "home" ward, forgetting all information from the previous ward; similarly, the other participants in the old and new wards alter their notion of current membership. Since ward membership information is maintained optimistically, the problem of tracking membership in often-disconnected environments is straightforward.

Recall from Figure 4.6 that the addition of the new ward member may change the ward set. Similarly, the ward set at the old ward may shrink in size, as the ward set is dynamically and optimistically recalculated when ward membership changes. The changes in the ward set propagate to the other ward masters in an optimistic, "need-to-know" fashion so that only the ward masters that are interested in the changes learn of them. A ward master is interested in the changes to a remote ward set when they affect how that ward master constructs its portion of the adaptive per-file ring for any of its data objects. Replicas (whether they are ward members or masters) must compute their own local portion of the per-file ring for the objects they locally store. When that ring changes, because someone in the ring no longer stores the object or a new member is added to the ring, this information must propagate to ensure correct ring formation. For example, consider the case of Figure 4.7. The initial picture is depicted in scenario `A`. When file `Y` is added to replica 3 in scenario `B`, replica 4 must learn the information to properly adjust the adaptive ring topology. Replica 2 must also learn the information, both so it can adapt its ring (when replica 4 is inaccessible) and also so it can gossip the information to 4. However, replica 1 is completely uninterested in `Y`, and therefore does not need to learn any new information. In the example, the replicas could be normal ward members or ward masters; the algorithms treat both cases equivalently.

Since both ward sets of the old and new wards can potentially change, and these changes are eventually propagated to other ward masters, ward changing can be a heavyweight operation. However, users benefit from the up-front cost. All local data can be synchronized completely within the local ward, giving users the best possible quality of service and reconciliation performance. By definition, there is no one cheaper to synchronize with than a local ward member. Ward changing is depicted graphically in Figure 4.6.

Ward overlapping

In contrast, ward overlapping is intended as a very lightweight mechanism with minimal up-front cost. Instead of enlarging the ward set of the new ward, we keep it unchanged, thereby not causing any global changes within the system. Only the new ward is affected by the operation; the information concerning the addition of a new member must be propagated to the other members. The old ward, in fact, never becomes directly knowledgeable of the operation.[3] The localization of changes makes it a lightweight operation, both to perform and to undo.

Ward overlapping allows simultaneous multi-ward membership, enabling direct communication with the members of each ward. We avoid changing the ward set of the new ward by making the new replica an "overlapped" member. No one can distinguish between "real" and overlapped members; the only difference is in the management of the ward set and the actions that occur when performing the initial motion. Instead of merging the existing ward set with the mobile machine's replica set, the ward set remains unaltered. Data in the intersection of the replica set and the ward set can be reconciled locally with the members of the new ward. However, data in the set difference cannot be reconciled locally, and must either temporarily remain unsynchronized or else be reconciled with the original home ward. Ward overlapping is illustrated graphically in Figure 4.6.

Ward motion summary

When a replica enters another ward, there are only two possibilities: the ward set can change or remain the same. The former creates a performance-improving but heavyweight solution; the latter causes a moderate performance degradation when synchronizing data not stored in the new ward, but provides a very lightweight solution for transient mobile situations. Since both are functionally equivalent, the system can transparently upgrade from overlapping to changing if the motion appears more permanent than first expected.

Additionally, since ward formation is itself dynamic, users can easily form *mobile workgroups* by identifying a set of mobile replicas as a new (possibly temporary) ward. With ward overlapping, mobile workgroups can be formed in a lightweight fashion without leaving the old wards. Ward motion and dynamic ward formation and destruction provide for direct any-to-any communication between any set of replicas in the entire system.

At the implementation level, the difference between ward changing and ward overlapping is minor, although some code for changing wouldn't exist if overlapping were the only form of ward motion. In both cases the mobile replica communicates with a replica in the new ward. At the outset of the communication, the mobile replica specifies the requested type of motion. If none is specified, the system assumes it is a visit (overlap).

When performing a ward overlap, the mobile replica simply learns about the contents of the ward set. For each data object in the ward set that is also in the mobile replica's replica set, the data object is marked as being in both old and new wards simultaneously—the object is "overlapped" between the two wards. Objects in the set difference (i.e., not in the intersection of the two sets) remain unchanged at the mobile replica. At the end of the operation, a number of data objects have been marked as overlapped at the mobile replica, and the mobile replica has been added as member of the new ward. Any overlapped object can be synchronized locally within the new ward; other objects at the mobile replica must be synchronized with the original ward. Of course, since the overlapped objects belong to both wards, they too can be synchronized with the original ward, but synchronization performance would presumably be worse than local synchronization within the new ward.

In contrast, when performing a ward change, the mobile replica communicates its entire replica set to a member of the new ward. Each object that is not already part of the new ward set is added to the ward set, possibly only in a *virtual* manner at the ward master. At the end of the operation, the ward set has expanded by the difference between the original ward set and the mobile replica's replica set. Each object is marked as belonging to the new ward. The mobile replica completes the operation by communicating with a member of the old ward and dropping its membership altogether.

[3] While replicas in the old ward may discover that the moving replica is no longer available for normal synchronization, the replicas in the old ward receive no direct notification when a replica performs ward overlapping.

4.4 Ward Master Re-election

When a ward master physically dies, or becomes inaccessible for long periods of time, a new ward master must be elected. Overall correctness is not impacted by an inaccessible ward master; the negative impact on the system is that new information can neither flow into nor out of that ward. Additionally, the meta-ward of ward masters cannot make progress on their distributed algorithms, since they are waiting for the inaccessible ward master to participate. For global consistency, a new ward master should be elected.

There are many systems and strategies for performing election in distributed environments with common partitions [Marchetti-Spaccamela 1987, Awerbuch 1987, Itai *et al.* 1990, Singh 1994, Garcia-Molina 1982, Gafni 1985]. Fortunately, our problem of ward master re-election is not as complicated as in other scenarios. The ward master need not be re-elected immediately, adding considerable simplification to the re-election protocol. Since optimistic consistency algorithms can tolerate partitions and communication delays, there is no reason to attempt immediate healing of the partition. We therefore avoid using network heartbeats and other mechanisms for fast partition detection and ward master failure, and instead detect the problem lazily. At synchronization time, when a ward member attempts to synchronize with the ward master but fails (because the ward master is inaccessible), such an attempt is logged. After a sufficient threshold of failed attempts has been reached, the ward members know they must re-elect a new ward master. Using a threshold means that we tolerate temporary failures and disconnections by the ward master.

Additionally, many election strategies must specifically guard against multiple master scenarios that can arise during partition healing. Once each partition has re-elected a new master, the two different masters both attempt to assert authority during partition healing, which often causes problems. Fortunately, the Ward Model tolerates multiple masters, and can therefore address the situation lazily. Correctness is not affected by having multiple masters in the same ward. The situation is not cost effective or efficient, since multiple sites are storing additional data structures and are performing inter-ward synchronization, but the model still functions correctly. Eventually, the two ward masters discover each other as a product of normal intra-ward synchronization, and the two of them agree upon a single master for the ward.

Ward master re-election can therefore afford to be a very simple process. We order all replicas by replica identifier, and further allow replicas to specify themselves as willing or not-willing candidates for the role of master. Re-election identifies the master as the accessible replica with the smallest replica identifier that has marked itself as willing or, if there are no willing candidates, simply the site with smallest replica identifier. Each member in the partition reaches the same decision, and we therefore avoid the expense of complicated, distributed consensus algorithms.

Once we elect a new ward master, the new master must learn the entire ward set for inter-ward synchronization, discover the members of the higher-level ward to form the inter-ward synchronization topology, and recover state information such as what is physically stored at the other ward masters, again for inter-ward synchronization. Each is discussed in turn.

4.4.1 Re-learning the ward set

Re-learning the ward set occurs lazily and asynchronously. Initially, the new ward master knows only its replica set. As synchronization propagates information within the ward, the new ward master slowly learns the complete ward set. Objects unknown to the new ward master are not "advertised" to other ward masters as being stored in the ward until the new ward master learns of them; new updates to a given intra-ward object are only propagated into the ward once the ward master knows that the object is in the ward set. However, overall correctness is not impacted by the lazy generation of the ward set. The ward set is always only optimistically known at the ward master, since individual replicas can change their replica sets without informing the ward master. Therefore, the ward master is never guaranteed to know the *exact* ward set, only an optimistic *view* of the ward set. We employ the same mechanism in reconstructing the ward set after ward master re-election. The solution works well, because we leverage the existing mechanisms rather than constructing special-case code for re-election purposes. The tradeoff is one of consistency: the intra-ward objects that are not known at the new ward master will not be synchronized between wards until the new ward master properly learns of them. The knowledge of the ward set propagates relatively quickly throughout the ward as part of standard synchronization, so the tradeoff seems to be favorable.

4.4.2 Discovering the higher-level ward

The new ward master needs to become a member of the higher-level ward consisting of the other ward masters. Doing so involves two actions. First, the new ward master must learn the identities of the other ward masters, to properly form the synchronization topologies. Second, the existing ward masters must learn the identity of the new ward master, so they properly include it when they form their synchronization rings.

Both actions can be accomplished simply by having the new ward master communicate with an existing ward master. The existing ward master adds the new ward master into its list, and the identity of the new ward master slowly propagates during normal synchronization to the other ward masters. Additionally, the new ward master learns the set of other ward masters from the existing ward master. The main issue, therefore, is to identify an existing ward master.

Flooding would be one approach. We could flood the network with packets asking for other ward masters to identify themselves. Flooding is not an efficient use of bandwidth, however, and does not produce correct behavior during network partitions. We therefore replicate the identities of other ward masters within each ward. After ward master re-election, the identities of the other ward masters are therefore locally known to the new ward master. We make the assumption that the identities of the other ward masters do not change faster than the synchronization architecture updates the cached intra-ward information. If the identities stored within the ward are all out-dated, human intervention will be required to connect the ward masters. We believe our assumption to be a valid one, and therefore do not expect human intervention to be required in general.

4.4.3 Recovering state information

The new ward master must learn more than just the identities of the other ward masters. It must recover state information, including:

- Finding out what data objects are stored at the other ward masters
- Discovering the status and participating in the garbage collection of deleted objects

Fortunately, state recovery occurs automatically as a result of normal synchronization. Since the state information is replicated at the other ward masters, the new ward master slowly learns the information as it synchronizes with the various other ward masters.

4.5 Detection of Failed Machines

The previous section discussed the issue of ward master re-election; however, we must also handle the failures of normal ward members. Since we distribute global state among all ward members, such as that used for performing garbage collection (Chapter 7) and version vector compression (Chapter 8), the undetected failure of a given machine could permanently stall all distributed algorithms that expect its participation. Since the Ward Model must already tolerate network partitions and temporarily inaccessible replicas, due to the realities of mobile computing, we ignore temporary failures (which automatically correct themselves) and only address permanent machine failures.

Roam has a general framework for notifying a given replica `R` that some other replica `S` has permanently failed and distributing that information to all other replicas within the ward. Basically, each replica maintains a list of the replicas that have permanently failed. Replicas gossip about the contents of the list as part of normal synchronization, and we employ a two-phase algorithm (essentially the same as garbage collection) for removing `S`'s entry from the list once all replicas learn that `S` has failed.

However, our current decision criteria for determining that a machine has permanently failed involves manual control. A robust solution must automate the decision process, and once the system makes the decision, a number of complicated race conditions and other issues arise, because a permanently failed machine cannot be distinguished from one which has simply been absent for a long time.

The underlying issue concerns the distributed algorithms that stall, waiting for participation from the failed machine. If we allow the algorithms to make progress and complete without participation by the failed machine, and then the failed machine returns (having not actually failed, but merely been absent), incorrect behavior can result. In the case of the garbage collection algorithm, removed files can reappear in the namespace. In the case of the version vector compression algorithm, false update/update conflicts can be detected.

However, by not instituting a time-out period for the detection of failed machines, and by relying on human intervention to decide which machines have failed, the algorithms can stall for arbitrarily long periods of time. A real solution must avoid stalling the algorithms indefinitely, while still correctly handling the case of the returning machine.

Our design uses time-outs to avoid stalling the algorithms and proxies to ensure correct operation if a machine returns after the time-out period has expired. To remove race conditions, we only allow one machine to decide that replica `R` has timed-out, called `R`'s *time keeper*. When `R`'s time keeper decides that `R` has failed, it propagates that fact to all other replicas via gossiping during reconciliation. Additionally, with `R`'s time-out notice is sent the identity of a proxy which will take responsibility for `R`'s re-entry should it re-appear. For simplicity, the proxy can be the same replica that times-out replica `R`, although this need not be the case. The same machine may serve as the proxy for multiple replicas; additionally, the same machine may serve as the time keeper for multiple replicas. An easy choice for the time keeper is the preceding replica in the adaptive ring, although any design choice can be made.

Once a replica learns that `R` has timed-out, it can ignore `R`'s participation and make progress on, if not complete, the distributed algorithms. The identity of `R`'s proxy propagates with the knowledge that `R` has timed-out; it is therefore impossible for a given replica to complete a distributed algorithm without knowing `R`'s proxy. The race conditions that occur when one replica times-out `R` while `R` re-appears by contacting some other replica can therefore be avoided. If replica `R` re-appears and contacts a replica that has not yet learned out the time-out, then `R` can still learn the state of the algorithm; on the other hand, any replica that believes that `R` has timed-out must know `R`'s proxy, and can therefore refuse contact with `R` until `R` first contacts the proxy. Incorrect results can only occur when `R` re-appears and contacts a replica that unknown to `R` has completed a distributed algorithm; the time-out/proxy solution guarantees that this case cannot occur.

For `R` to properly re-integrate itself, it must first contact the proxy. The proxy logs enough information for `R` to properly replay the distributed algorithms and bring itself into a consistent state; afterwards, the proxy propagates the knowledge that `R` is no longer timed-out. The proxy should therefore have enough free disk space to maintain the log information. In essence, the proxy participates in the algorithms on `R`'s behalf, and then later informs `R` what happened. If replica `R` actually had the latest version of data or a new name for a file, these actions will be dealt with correctly by normal reconciliation. On the other hand, for its participation in the distributed algorithms, `R` must replay the log information. After a sufficiently long second time-out period, a great deal longer than the first, `R` can be declared officially failed, and the log information can be destroyed. The proxy maintains the knowledge that `R`'s log has been destroyed; if by some chance `R` re-appears, the proxy blocks its participation, and `R`'s on-disk data structures must be reconstructed with manual intervention.

4.6 Scalability of the Ward Model

The Ward Model as defined so far is a two-level hierarchy containing replicas and ward masters, which can be considered "super-replicas." As such, it scales much better than a traditional peer architecture, which is a one-level hierarchy, but clearly is not as scalable as a general hierarchical architecture. While the implementation in Roam only supports two levels, the Ward Model is a fully general hierarchical model and easily scales into a three- or four-level hierarchy to provide larger replication factors if required. In this section we discuss the scalability of the Ward Model and the enhancements that would be required to support a general, multi-level hierarchy.

4.6.1 Scalability

The underlying issue impacting the scalability of the system is the storage space required to identify the ward set. Since the ward master synchronizes on behalf of its entire ward, it must know the entire ward set. However, selective replication provides each replica with a large degree of replication freedom. As such, the ward set changes dynamically, optimistically, and without global coordination. Additionally, replicas themselves are free to dynamically change their ward membership via the preceding ward-changing algorithms. Ward changing affects the ward set, as the ward set must expand to incorporate the new data objects brought into the ward or shrink when the last remaining replica of a data object leaves the ward.

The ward set is therefore potentially dynamic and volatile. The ward master, however, must be able to identify the complete set. To do so, the ward master must list each entry individually. Selective replication forces the same constraint on individual replicas when they construct their replica set; it follows that the constraint carries into the ward master and ward set, since the ward master is just a "super-replica."

Individually naming each entry in the ward set becomes expensive, and costs $O(N)$, where N is the size of the ward set. As the number of hierarchical levels increases, the amount of storage space committed to naming the entire ward set increases as well (since the size of the ward set grows), until the ward set equals the size of the volume. When the ward set is the complete volume, the ward set can be identified just by naming the volume, requiring only $O(1)$ space. Naming, after all, was one of the chief advantages of volumes, as described in Chapter 3.

Therefore, allowing the Ward Model to properly scale in a multi-level hierarchical fashion requires using the knowledge that identifying the complete volume requires only $O(1)$ space. When the size of the ward set is less than half the size of the volume, it is better identified by individually listing elements; when it is more than half the volume, it is better identified by listing the elements it *doesn't* contain. In this way, the Ward Model scales nicely in the fully-general hierarchical model, because as the number of hierarchies grows, the space required to identify the ward set actually grows smaller, starting at the halfway point. When the ward set becomes the full volume, the space required is only $O(1)$ for that ward master and all hierarchical masters above it.

Critics may argue that since identifying a volume requires $O(1)$ space, an improved scaling solution involves naming a ward set with a single name, regardless of the size of the ward set. However, doing so does not offer any real improvement. Simply naming a replica set may reduce the size required to store it, but reconciliation must be able to determine the relationships *among* replica sets. Recall that for a given site to synchronize all of its local data, it dynamically constructs its portion of the adaptive ring for each locally stored object and then synchronizes the objects along their respective topologies. The algorithm requires the ability to determine the intersection and set difference between two replica sets, which is not possible in general simply by naming the sets. Lists of members must be identified, which nullifies most the advantage of naming the replica sets in the first place.

4.6.2 General hierarchies

Although the Ward Model scales nicely and could be a fully general, multi-level hierarchy, deploying it as such involves more than simply modifying the data structures to support multiple levels instead of two. In a general, multi-level hierarchy the participants at the upper hierarchical levels become increasingly important for overall system reliability, given that if they fail they affect a large percentage of other machines. The failure or temporary unavailability of a ward master in a two-level hierarchy only isolates that one ward from the rest of the community; however, in a multi-level hierarchy the failure of a given master can have a much more global impact, demonstrated by Figure 4.8. Many have previously addressed the single point

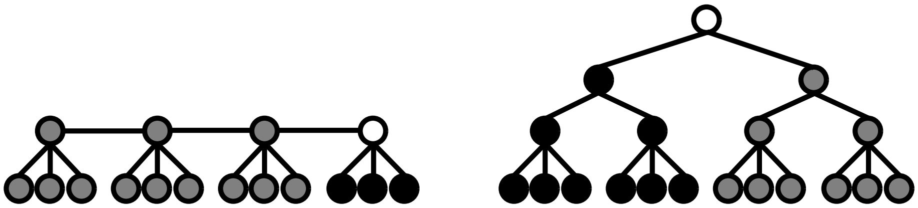

Figure 4.8: Two hierarchical models: a two-level and a multi-level. Replicas are indicated by circles in a hierarchical fashion, pointing to their ward master if one exists. In each case, assuming the white master fails, the failure isolates all black replicas from the rest of the community. That is, no black replica can communicate with or learn new information from any gray replica.

of failure issue by replicating the important nodes at the higher levels of the hierarchy. In our case, we would require multiple ward masters at higher hierarchical levels, to ensure that consistency was never adversely impacted. Additionally, we would require more resilient algorithms, such as those governing ward master re-election. Our current re-election algorithms are relatively simple and operate lazily, because our existing, two-level environment does not require more robust algorithms. However, given the importance of recovering from failures at higher levels in a multi-level hierarchy, we would need faster and more resilient re-election protocols to properly maintain the hierarchy in the fully general model.

4.7 Ward Model Implementation

The main data structure behind the implementation is the *ward vector*. The ward vector stores the ward-specific data structures for a given ward and contains all information required for intra-ward operation, including synchronization, replica selection, and the execution of distributed algorithms. The actual ward vector is a list of ward members and a series of information about them, including whether or not they physically store the object in question (selective replication), their participation in the garbage collection of the object if it is deleted, and other intra-ward pieces of information.

For maximum flexibility, each file object maintains its own ward vector independently. Having each file maintain a separate ward vector allows objects to belong to different sets of wards (i.e., ward overlapping), allows distributed algorithms such as garbage collection (Chapter 7) to proceed independently between objects, and is required for the implementation of selective replication.

Each ward vector is tagged with a specific *ward identifier*, uniquely naming the ward. Wards are named by the tuple {first ward master, counter at that ward master}, guaranteeing that no two wards will ever have the same name. While maintaining a ward vector per file object may initially seem expensive, performance data in Chapter 9 demonstrates that Roam is not significantly more expensive than Rumor.

Ward motion simply requires adjusting the ward vector. Ward changing involves removing the entire ward vector and substituting a new one in its place, corresponding to the new ward. In contrast, ward overlapping maintains the old ward vector and adds an additional ward vector for the new ward. Replicas must therefore be capable of storing and indexing multiple ward vectors, one for each ward it simultaneously belongs to. Ward masters, in fact, store two ward vectors; one corresponding to the ward itself, and one corresponding to the higher-level ward of all ward masters. When a ward master communicates with another site within its ward, the master uses the intra-ward vector; when communicating with another ward master, the inter-ward vector is used. In general, two sites communicate using the ward vector corresponding to the ward that contains both as members.

Ward motion additionally requires dropping one's membership in a ward. When changing wards, the moving replica must revoke its membership in the old ward; when "undoing" a ward overlap, the moving replica must revoke its temporary membership in the new ward. Dropping one's membership can be done by contacting any existing member of the ward in question. Each ward maintains a list of the previous members. The list is maintained optimistically and is stored at each current ward member. Therefore, dropping one's membership simply requires updating the list at an existing ward member.

Of course, without pruning the list, it grows linearly over time. The list is pruned by executing a distributed consensus algorithm similar to garbage collection (Chapter 7) among all ward members. Once everyone knows that a given replica `R` has dropped its membership, and everyone has `R` in their list, everyone can prune the list and completely remove all mention of `R`.

Chapter 5

Consistency Maintenance

At the very heart of any replication system is the synchronization method: the mechanism that propagates updates and ensures consistency between all replicas. In ROAM, this process is called *reconciliation.*[1] Without correct reconciliation, replicas diverge, and the whole benefit of replication is lost.

The issue of reconciliation topologies and communication patterns between replicas and ward masters has already been discussed in Chapter 4. This chapter discusses the actual details regarding synchronization. In its most basic form, reconciliation is a pairwise process and consists of just two replicas exchanging updates. Although selective replication often requires communication with multiple other replicas (the topology is discussed in Chapter 4), the remote sites are queried in series, not in parallel, so reconciliation never directly involves more than two replicas. In a mobile context, or any environment where disconnections and network partitions are commonplace, multiple replicas may never be simultaneously and mutually accessible. Relying on multiple simultaneous communication partners for correctness is not a viable solution in general. The minimal requirement for reconciliation is two replicas, since one replica cannot learn new information by itself; similarly, the maximal state that can be reliably guaranteed is two replicas. Reconciliation is therefore a pairwise process requiring only two replicas.

Furthermore, reconciliation is a pull-only process, meaning that new information is only propagated in one direction. A target replica is said to reconcile *with* or *from* a source replica; at the end of the process, the target replica knows all information known by the source, but the source replica does not learn anything from the exchange. The source replica only learns new information when it initiates its own reconciliation.

The pull-only strategy has several benefits. First, it is simpler to model and describe, both from an algorithmic and implementation standpoint. Second, it is a more general approach, in that it allows support for inherently one-way communication media, such as reconciliation by floppy-disk transfer. While it is clearly possible to design a floppy-disk transfer mechanism with a two-way flow of information, doing so is inherently impractical, given the physical constraints. Disks are easily misplaced or lost, and the large latency for a complete round-trip means that upper bounds on protocols and time-outs on procedures cannot easily be determined. Often, models with significant latency on round-trip communication are simply better modeled as two one-way communication paths.

Of course, a two-way synchronization protocol has its merits as well. The primary advantage is performance. Even though a two-way model can be built out of two one-way exchanges, the combination does not generally perform as well as the integrated solution. Utilizing a two-way model would also maintain a higher degree of consistency, since at each synchronization interval two replicas would receive newer information, rather than just one. However, the main reason ROAM uses a one-way pull of information is because RUMOR, the underlying replication framework, is built with a one-way model. The framework is currently being modified to use a real two-way synchronization protocol; however, at this time the effort has not been completed.

This chapter describes the details of the pairwise, pull-only process, including the detection of updates

[1] The term was first used by Guy in FICUS [Guy 1987].

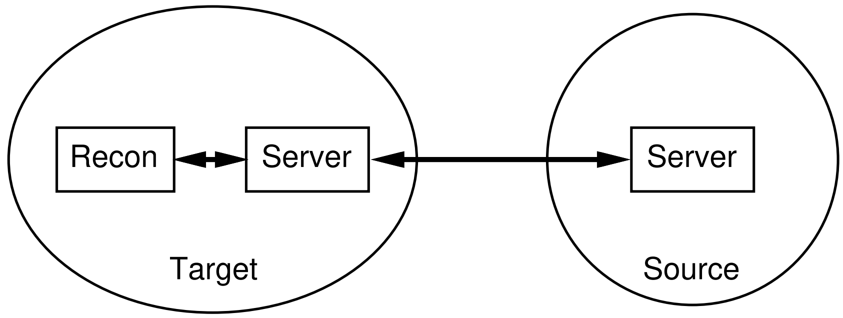

Figure 5.1: A high-level illustration of the reconciliation architecture. Machines are represented as circles; processes as rectangles within them. Communication is indicated as arrows between processes.

and the fetching and installation of data. We will first provide an overview of the reconciliation architecture and then describe its major pieces. We conclude with a state diagram illustrating the flow of control. Selective-replication specific issues, such as what happens when the two replicas maintain different replica sets, are described separately in Chapter 6.

5.1 Reconciliation Architecture

Reconciliation is separated into two major processes. The *recon* process obtains meta-information from a remote replica and decides what local objects require updates. The *server* process handles data requests from the recon process, fetches file system data from remote replicas, and installs the actual file system modifications. The recon process is transitory; it is initiated on demand when either users or automated daemons decide to initiate reconciliation. In contrast, the server process is always resident on each participant host, although in a manner that consumes almost no resources when it is non-active. Figure 5.1 provides a graphical representation of the high-level architecture.

The division of labor is desirable for a number of reasons. First, it separates functionality into two major categories: comparing replicas and sending and receiving data. Since these are orthogonal issues, it makes sense to have them logically separated in different processes. Additionally, the complexity of each is reduced by limiting its responsibilities; a process responsible for both would constantly be switching between data installation and object reconciliation duties. Rumor operates in such a mode [Reiher *et al.* 1996].

Second, the division provides a common point—the server—that controls the transport of all information into and out of the replica. (The request for remote meta-information is initiated though the server process.) Having all communication paths onto and off of the machine travel through the server process provides both a single point for security authentication and better encapsulation of various security policies. In this sense, the server can act as a security firewall for the reconciliation process. Additionally, having all communication paths flow through a single point allows a *transport-independent* design, so the process can execute independently of the physical transport mechanism being used between machines. The same machine can use different types of connections to different hosts, such as email, Unix `rshell`, TCP sockets, or floppy-disk transfer.

5.2 The Recon Process

The recon process selects an appropriate remote replica to synchronize with, according to the adaptive ring topology described in Chapter 4. For each locally replicated file, the recon process determines whether or not the remote replica's version of the object is more recent. If the remote replica is deemed to have newer data, or new files to be stored locally, the recon process asks the server to fetch file data and make the appropriate file system modifications.

To make its decisions, the recon process is separated into three sub-processes:

- A file-system scanner that runs locally
- A file-system scanner that runs remotely
- A decision module

We will discuss each in turn.

5.2.1 File-system scanner

The file-system scanner is essentially identical to the one used in RUMOR; as such, we only briefly describe it here. A more detailed discussion can be found elsewhere [Reiher *et al.* 1996].

The file-system scanner infers the user's file system modifications that have been performed since the last scan of the file system and updates ROAM's data structures accordingly. Since ROAM (and RUMOR) do not intercept file system operations, they must periodically re-determine the state of the file system and update their own data structures. New objects and names are detected, as are removals of existing objects and names. Files that have been updated have their version vectors incremented. At the end of the scan phase, the internal data structures match what exists on disk.

Of course, nothing prevents the user from immediately updating a file which was just scanned; all that is required for correctness is that we generate an *atomic* scan of each object, so that our data structures correspond to an actual previous state (perhaps the current state) of the file. Before any permanent action is taken on the file, it will be rescanned to ensure correctness and atomicity. Additionally, updates "missed" during one scan (because they occurred on a file after we scanned it) will be reliably detected during the next scan. Correctness is therefore preserved even when the data structures lag behind the current state of the file system.

One scan phase executes locally, and one executes at the remote replica. The remote replica updates its own data structures and additionally sends these data structures, the replication meta-information including the version vectors and other replicated attributes, to the recon process. The transportation of the remote meta-information, as well as the later transport of data, occurs in a *transport-independent* manner. The recon process itself has no knowledge of the specific transfer mechanism being used. Currently implemented transport mechanisms include email, UNIX `rshell`, TCP sockets, and floppy-disk transfer.

The scanning process generates lists called *filelists* containing each local object and its current attributes. Attributes in the filelist include not only the native file system attributes (such as `mtime`, `ctime`, and the mode bits in UNIX) but also all replicated attributes maintained by the replication facility, such as version vectors and ward vectors.[2] Additionally, directory meta-information contains a list of all directory entries. A complete list of the replicated attributes and their specific purpose is discussed in [Salomone 1998]. Since the filelist contains only attributes and no data, it is small and relatively inexpensive to transfer.

5.2.2 Decision module

Each scanning process generates a set of replication meta-information about the state of the replicated data on that machine. By comparing the two sets of replication meta-information, the decision module decides what file system modifications need to applied locally. It only makes the decisions; file system modifications and the fetching and installation of file data from the remote machine are performed by the server process.

Decisions regarding existing files are determined by comparing version vectors. If the remote version vector dominates the local one, then remote data is requested. The server will asynchronously receive the

[2] For portability between operating systems, the replicated attributes also contain attributes native to other file systems when replicas exist on multiple operating system platforms [Salomone 1998].

new data from the remote replica and install it, along with its accompanying version vector. If the remote version conflicts with the local one, an update/update conflict has been detected and is handled by the conflict handling mechanism [Reiher *et al.* 1994]. Conflicts are handled in the same manner as in Ficus and Rumor: remote data is requested so that it can be accessed locally for either automatic or user-involved conflict resolution. If the semantics of the file are well-known, and a special automated tool known as a *resolver* for that file type exists, the conflict can typically be automatically resolved. Otherwise, the user is notified of the conflict by email. By accessing both copies of the data, he or she can determine the most appropriate resulting state.

Decisions regarding other file system modifications, such as the creation of new files and the deletion of others, are determined by reconciling directories and acting upon the file names. Objects with new names must be installed with accompanying data, so the recon process asks the server process to fetch their data. Objects with no local names are considered locally deleted, and they begin the garbage collection process. New names can be distinguished from old ones that have been deleted at one of the two replicas by a process known as create/delete disambiguation (discussed in depth in Chapter 7). Full details on the decision-making process can be found elsewhere [Guy *et al.* 1990c, Guy *et al.* 1993, Ratner 1995].

When all decisions have been made, and all requests given to the server, the recon process waits until the server has fulfilled all data requests and performed all system modifications. Upon notification from the server, the recon process terminates. The waiting is required for correctness in scenarios where the user might take immediate action upon termination of the recon process. For instance, a mobile user reconciling via modem may wish to disconnect as soon as possible, perhaps to save money or to allow phone use by others. Alternatively, a user may wish to immediately execute some process on the newly-installed data. However, the only user-visible process is the recon process. The user has no direct way of knowing when the server has fulfilled all data requests, given that the server operates independently and data flows to the server asynchronously. In the mobile-user scenario, if the network connection is terminated too soon, some requests will go unfulfilled. While not incorrect from a system point of view (the next time reconciliation runs it will again request data for those objects which did not receive new data last time), such behavior seems incorrect from the user's point of view, given that the user expected all the updates, not just some of them.

In the other scenario, where the user wants to execute a process immediately upon the completion of reconciliation, the user's job may actually generate incorrect results if it executes before all updates have been installed locally. Therefore, to protect users in all scenarios, we make the recon process wait for confirmation from the server before terminating; at this point the user is guaranteed that all updates have been received.

5.2.3 Optimizations

Several performance-improving optimizations can be made to the above process structure. First and foremost, the two scanning phases (local and remote) can be executed in parallel. The decision module simply needs the output of both processes; it doesn't care which completes first.

More importantly, however, the scanning phases need not be performed at the actual reconciliation time. Since objects are rescanned before committing any permanent action on them, such as installing new data, we can still guarantee correct behavior even if the scan is performed some time prior to reconciliation time. Updates not detected by the current reconciliation process (because they occurred after the last scan) will be detected by the next scan. The scan phases can be executed during idle periods or periods of low file system activity, resulting in both faster reconciliations (since the scan does not need to be performed at reconciliation time) and better performance from the user's point of view (because the user does not notice the expense of the scan).

Finally, the recon process need not wait for the server to acknowledge fulfillment of all data requests, only the "important" ones. Users, or agents acting on the users' behalf, may indicate that they only require updates to a specific set of objects. Independently of whether the server has serviced all requests, the recon process may choose to exit when this smaller set of "critical" objects has had new data installed.

5.2.4 Non-privileged operation

The recon process does not require supervisory (root) access. It runs entirely with normal user privileges.

5.3 The Server Process

The second half of reconciliation is the server process. The server is responsible for handling all data requests, fetching and installing the data, and making all file system modifications.

The server process is actually composed of two pieces, the `wardd` and the `in-out server`. The `wardd` is always running at each site, and dynamically starts `in-out server`s when they are required; in this way the `wardd` functions in a similar fashion to other UNIX resource daemons like `inetd`. It controls resource allocation by only starting `in-out server`s when they are required. The `in-out server` services the volume's data requests and installs data on behalf of the recon process. Both will now be described in more detail.

5.3.1 `wardd`

The `wardd`, or ward daemon, functions as the well-known contact at every machine. It controls the allocation of `in-out server`s and initiates new ones when they are required. Both local and remote processes requesting communication with an `in-out server` first communicate with the `wardd`, requesting a communication handle for a particular volume replica. The `wardd` starts a new `in-out server` if necessary and passes a communication handle back to the requesting process, which can now contact the `in-out server` directly. The communication handle is currently a TCP socket number, though it could in theory be any opaque data object used for communication, such as an email address.

Communication with the `wardd` occurs in one of two forms. *Registration* messages allow a new volume replica to be added to the `wardd`'s internal list. *Query* messages ask the `wardd` for a communication handle (TCP socket number) used for communicating with an `in-out server` for a particular volume replica. The `wardd` finds the entry in its internal table, starts the server if one is not already running, and returns the communication handle to the requesting process.

The `wardd` is designed to run either in supervisory (root) or normal user mode. When running in root mode, it maintains on non-volatile storage a list of the local volumes and the information required to start the `in-out server`s. When running in normal user mode, the list is maintained only in memory, and therefore must be re-initialized after machine reboots.

The `in-out server` can also execute in either root or normal user mode. When running in root mode, one `in-out server` can service all local volume replicas. However, when running in user mode, the `in-out server` is more restricted in the file system operations it can perform (only root can perform a UNIX `chown`, which changes object ownership). We therefore potentially require one `in-out server` per local volume replica, each one running with different permissions. To support the architecture, the `wardd` stores the pathname of the correct `in-out server` for each volume in its list, and guarantees that the proper one is running for the given reconciliation process.

5.3.2 `in-out server`

The `in-out server` processes all data requests from the recon process, fetches the data from a remote `in-out server`, and installs the data and performs the file system modifications.[3] The `in-out server` is designed to service multiple volume replicas simultaneously; alternatively, one can have multiple `in-out servers` running on the same machine. In this way the design supports installations with both root and non-root privileges, respectively.

[3] We call it "in-out" to distinguish it from previous data servers (such as in RUMOR) that only responded to requests and did not themselves receive data or perform file system changes.

Running the `in-out server` with root privileges simplifies the model and makes more efficient use of system resources, since only one server is running. However, some users simply do not have root privileges on their machines, or for security concerns may wish to limit the number of programs that run as root. We therefore allow the flexibility of having multiple `in-out server`s, each one with a different set of permissions for a different volume replica.

When the `in-out server` executes as the superuser (root), it must be capable of servicing multiple volume replicas. To do so, the `in-out server` spawns a child process to handle each individual reconciliation process. Each recon process therefore has its own dedicated `in-out server` handling its requests. Additionally, the parent server remains active, ready to create other child servers for other simultaneous reconciliation processes. The design allows reconciliations on different volume replicas to execute concurrently. We explicitly disallow concurrent reconciliations on the same volume replica due to data structure and attribute database consistency issues. Each child terminates when the associated recon process terminates. The parent `in-out server` terminates whenever it has no active children and has not experienced any activity for "some time."[4] Since the `wardd` will execute a new `in-out server` if the current one terminates, the "some time" tradeoff is one of system resource utilization versus `in-out server` start-up time.

Recall from Chapter 4 that the ward master need not store actual file data for all intra-ward objects. Instead, the ward master may only store a *virtual* replica. However, during synchronization, the ward master may have to service data requests (fetching or installing) for an object that it only virtually stores. Requests for virtual replicas are forwarded to an `in-out server` at a site within the ward that stores data, using local information regarding which intra-ward participants store the object in question.

Unfortunately, forwarding the request may be impossible if no data-storing replica can be located. Reconciliation logs the occasions when request-forwarding could not occur. If request-forwarding fails often enough, reconciliation knows that the ward master must store a real replica, rather than a virtual one, to properly maintain consistency.

Communication with the `in-out server` takes one of two forms. *Control* messages are concerned with overall functionality, while the *data* messages carry with them either requests for data or the data itself. We describe both below.

Control messages

There are several different control messages, which can be grouped into four main categories:

- Those used to maintain data structure consistency between processes
- Those used to indicate requested file system modifications
- Those used for communication between the recon process and the local `in-out server`
- Those used to establish communication between the local `in-out server` and the remote `in-out server`

We discuss only a few select messages. The complete list of control messages and their descriptions can be found in Table 5.1.

Data-structure consistency messages are used to maintain consistency on data structures (like the filelist) between processes. For instance, during reconciliation the recon process updates the attributes for a given object; the new attributes need to be communicated to the `in-out server` since it stores a copy of them in memory for performance. Communication of new attributes is done with a *NewFilelist* message. Additionally, there are a pair of messages for identifying the set of volume replicas on the local machine—these are *NewVolumeReplica* and *RemoveVolumeReplica*, which add and delete volume replicas from the server's internal tables.

Recall that the recon process waits for the `in-out server` to receive and install data for all requests. The communication of what has been requested and which objects the recon process wants verification of

[4] Currently the time-out period is set at 12 minutes.

Message Name	Description
NewFilelist	A filelist entry has been updated
NewVolumeReplica	A new volume replica has been created locally
RemoveVolumeReplica	An existing local volume replica has been removed
NewDirectoryName	Create the directory or symbolic link with a given name in the file system
StartRecording	Start recording what data objects are received
RecordingQuery	The recon process wants to wait for a specific set of data requests to be received
EndRecording	The recon process doesn't want to wait anymore; tell it what has been received so far
ChildDone	A child server has finished
RequestCommChild	Request that the remote `in-out server` create a child server to handle our remote requests for data
EndCommChild	We are done communicating with the remote server
Ping	Is the server alive and does it service a particular volume replica?

Table 5.1: A list of the `in-out server` control messages and their descriptions.

Message Name	Description
GetData	A request for the server to fetch data
SendData	A request between servers, asking for data
ReceiveData	Requested data and the required information to install it

Table 5.2: A list of the `in-out server` data messages and their descriptions.

delivery is done with the *StartRecording*, *RecordingQuery*, and *EndRecording* messages. The first instructs the server to start recording the data objects it installs. The second identifies to the server the complete set of objects for which the recon process wants confirmation of receipt. The third tells the server that the recon process is unwilling to wait any longer, and the server should immediately communicate the set of objects that has been successfully received.

Finally, to service the data messages (described below) the local server must establish communication with a remote server. Communication is established with a *RequestCommChild* message and terminated with a *EndCommChild* message.

For security during transmission, all control messages sent to the server should be authenticated and encrypted.

Data messages

There are three types of data messages: *GetData*, *SendData*, and *ReceiveData*. Their descriptions are summarized in Table 5.2. *GetData* messages are sent from the recon process to the local `in-out server`, asking it to fetch data for a specific object. The `in-out server` processes the request by sending a *SendData* message to the remote `in-out server`, which packages the data and sends it back in a *ReceiveData* message. Upon receipt of the *ReceiveData* message, the data is locally installed. Figure 5.2 presents a visual picture of the flow of data and messages.

Recall that ward masters may not store physical data for a given object: they may only store a virtual replica. When responding to either a data request or data receipt for a virtual replica, the message must be routed to a real data-storing replica within the ward, as described in Chapter 4. The flow of data and

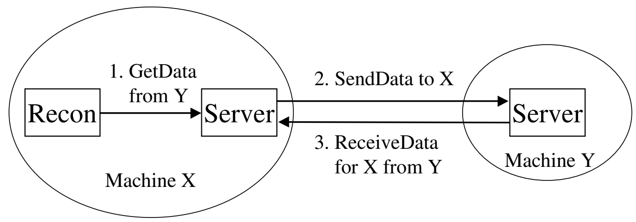

Figure 5.2: The flow of message and data corresponding to a request for data by the recon process of machine `X`.

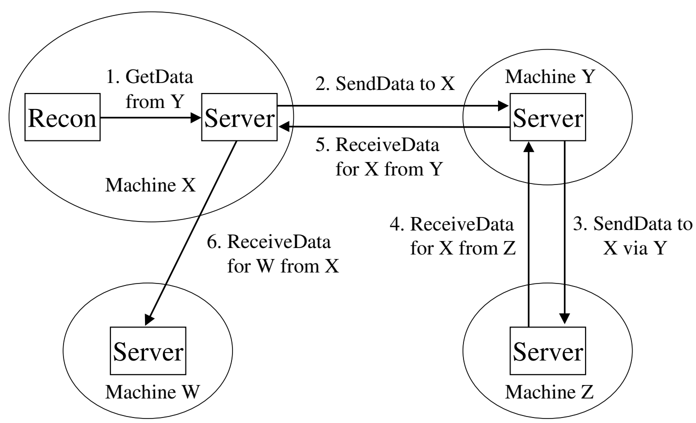

Figure 5.3: The flow of message and data corresponding to a request for data and the receipt of data where both store only virtual replicas. Both machines `X` and `Y` store virtual replicas; `W` and `Z` store the physical file data.

messages therefore becomes more intricate, as demonstrated by Figure 5.3.

In all cases, data packaging must occur atomically at the remote site. Errors such as false update/update conflicts could arise if the attributes from one point in time are attached to the file data from another. For instance, the remote site's replica could have been updated between the time that the initial attributes were sent to the recon process and the time that the data request is serviced. Additionally, the object could be updated simultaneously with the processing of the *SendData* message. To maintain correctness, we guarantee that the object is re-scanned to detect new updates, and the data and attributes are packaged atomically (which may itself include additional re-scans). The atomic package is sent back to the local server within a *ReceiveData* message.

Similarly, the local server must install the data in an atomic fashion, making sure not to overwrite updates that have been locally performed since the recon process scanned the local file system. Before applying an update, the local object is re-scanned. If it has been updated, an update/update conflict has been identified; otherwise, there is no conflict, and the remote data and attributes are installed locally.

For security during transmission, all data messages sent between the servers should be encrypted and authenticated.

5.4 Flow of Control

The flow of control of the reconciliation process, as seen by the target replica, is illustrated in Figure 5.4. All information into and out of each replica first passes through the server on that replica. The request and communication of the remote filelist are shown as a direct connection between the two processes, as that is the eventual state. First, however, the request for the remote filelist is given to the local server, which communicates with the remote server. The remote server initiates the scan on the remote machine, and each server "hands-off" its half of the connection to the appropriate process, thereby removing itself from the communication.

The `wardd` is not indicated in the flow of control, although contact with the `wardd` would be required to obtain a socket number for both the local and remote `in-out servers`.

5.5 Consistency Summary

We have just described the architectural design and implementation of reconciliation, separated into the recon and server processes. The design separates reconciliation's functionality into two categories: comparing replicas and sending and receiving data. The separation allows for a cleaner, more esthetic architectural design as well as more modular one, reducing the complexity of each individual portion. Furthermore, the division of labor means that the server process controls the transport of all information into and out of the replica. The server process becomes the single point for security authentication and encapsulation of various security policies, enabling an easy incorporation of the TRUFFLES [Reiher *et al.* 1993b] security manager. Furthermore, the single point allows a *transport-independent* design, so the reconciliation processes can be designed and execute independently of the physical transport mechanism being used between machines. Performance data on the architectural design is discussed later in Chapter 9.

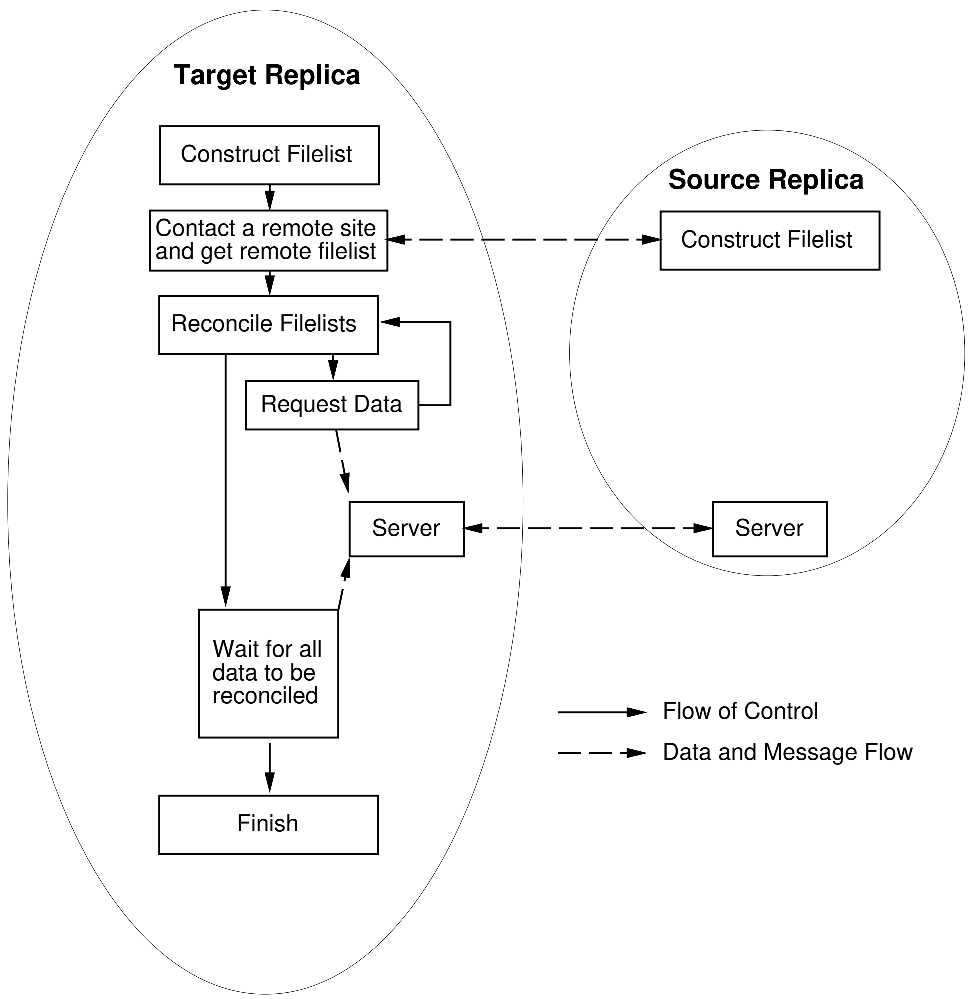

Figure 5.4: Flow of control diagram for a given reconciliation process, from the point of view of one replica. Both control and data flow are indicated.

Chapter 6

Selective Replication

Replication at the volume granularity is only a partial solution. While it provides a mechanism for optimistic replication, the mechanism is expensive and inefficient. Files and replicated objects are grouped into volumes for locality, performance, security, and naming reasons, as discussed in Chapter 3. However, these functions are orthogonal to the actual replication of the data objects within the volume. The replication of an object should be independent of its position in the namespace. Due to limitations on disk space and transmission cost and time, (users always want the least-cost, best-performance solution), the desired grouping for replication control often differs from the grouping defined by semantic relationships, such as those which govern directory usage.

The replication facility should provide a *namespace-independent* solution to minimize the costs of replication. Storing additional data does not affect correctness, but adds to the cost of replication in terms of disk space and data transmission. First, additional data occupies what would otherwise be usable disk space. If the user does not require specific objects, they might as well not reside on the local disk, allowing that disk space to be used for more important data. Kuenning's studies with file system trace data [Kuenning 1997] indicate that the user's set of currently "required" data objects, called the *working set*, is small enough to fit completely on the portable computer. However, if disk space is occupied by unimportant data not in the user's current working-set, then the entire working set may no longer fit. When the complete working set cannot be locally stored, the mobile user encounters problems ranging from minor inconveniences to complete stoppages of work and productivity [Kuenning 1997]. Selective replication therefore allows for better disk space utilization and, in cases where the entire working set could not otherwise be locally stored, is required for correctness.

Second, locally replicating unimportant data requires that additional time be spent to download updates and maintain consistency on the unimportant data. Mobile users are often poorly connected, and are therefore very concerned with the amount of data that must be transferred onto and off of their machines, as well as the total amount of time required to synchronize with another replica. When transferring data over high latency, low bandwidth, expensive links, time really is money. Selective replication therefore reduces the actual costs of replication, in that users only pay for replication on data they are locally interested in.

Selective replication is therefore required for a complete replication solution for the mobile user. The following sections describe the user model and semantics provided by selective replication, the details of the algorithms and implementations, the replication controls available to the user and the actual synchronization and maintenance of consistency. We delay discussion of the performance impact and benefits until Chapter 9.

6.1 User Model

Any implementation of selective replication requires an understandable and usable representation of files that are not locally stored. Selective replication should not break name transparency; therefore, the optimal solution utilizes transparent remote access when connected to other replicas and provides full access to the files wherever they are stored. When users are not connected to other replicas, a special error code should be returned to the requesting process, indicating that the file exists but that no replica is accessible. FICUS [Ratner 1995] provides this level of implementation.

However, ROAM (and the underlying RUMOR) is at the application level, not embedded in the operating system like FICUS. It cannot therefore easily represent non-local files differently to the user or return special access codes to requesting processes. While we could modify basic user libraries such as `libc` on UNIX or change standard programs like `ls`, such options make the software more difficult to install because changing such basic items requires supervisory access. Additionally, these solutions typically work poorly unless the actual vendor who produces a new version can be encouraged to adopt the particular modifications.

The set of options are therefore rather limited. Files that are not locally stored could be represented in the following ways:

- A zero-length file
- A file with special permissions or modes
- A file owned by a special user (e.g., the not-locally-stored user)
- A symbolic link to a non-existent file, where the contents of the link indicate that the file is not locally stored
- It could not appear at all in the local namespace

Since we cannot provide the optimal implementation, none of the available solutions are ideal. We will enumerate the advantages and disadvantages of each of the above solutions, discuss the solution that we selected, and describe its impact on users.

6.1.1 Possible solutions

Solutions that could be misconstrued as normal file system occurrences, such as zero-length files or files with special permissions, are not viable options because of the possibility of having them occur naturally as a result of normal activity. Having the object owned by a special user forces that special user to have a reserved position in the password file, meaning the system requires supervisory (root) access during installation. Additionally, the solution cannot be implemented on systems without notions of file ownership, such as WINDOWS 95. Finally, having the object be represented as a special symbolic link fools processes trying to access the object because its file system type does not remain constant. If, for example, the object is actually a directory, applications receive a confusing error code (ENOTDIR in UNIX, or "error: not a directory") when attempting to treat it as its true file system type.

6.1.2 Chosen solution

We decided to remove the name of the object entirely from the namespace. While name transparency is not preserved, none of the above solutions do an adequate job of preserving it. In all solutions, we are unable to return special error codes to the applications when a non-local object is requested; arguably, returning the error "file not found" (ENOENT in UNIX) is an appropriate, though less than ideal, alternative. Additionally, transparent remote access is not currently supported in RUMOR, meaning that even if the user is connected to another replica, he or she doesn't have an easy mechanism for accessing a remote replica of a file. While we provide the hooks in the replication databases for locating remote replicas, the presentation of the namespace is the responsibility of the remote access mechanism. We therefore selected a namespace solution that would not interfere with one implemented by an eventual remote access solution. Of course, a special tool that queries the replication facility and supplies information about objects that are not locally stored could be easily provided as a secondary user aid.

Having the name not appear in the local namespace suffers in that potentially more *name conflicts* can occur: that is, creations of multiple objects in the same directory with the same name. While typically rare in normal operation [Reiher *et al.* 1994], these conflicts could become more common when the names do not appear in the namespace. Automated tools that create new objects in response to access failures (i.e., the object isn't there so create a new one) could increase the number of name conflicts when access failure results from not physically storing a local replica. However, the potential problem is only theoretical, has not been experienced in practice, and the system already has mechanisms to resolve name conflicts. Should these mechanisms prove incapable of handling any significant rise in the number of name conflicts, the issue of namespace presentation should be reevaluated.

6.1.3 Solution impact

Our chosen policy—removing the file name entirely from the namespace—causes two effects. First, since name transparency cannot be preserved without transparent remote access, applications that work correctly at one replica may fail at another due to differences in replication patterns. Without name transparency or a method of returning special error codes to the calling processes, the situation cannot be improved. As mentioned above, in these cases perhaps the best error code is a "file not found" error.

Second, and perhaps more important, concerns the issue of using selective replication to add objects to the local namespace. Since the pathname is not available to the user through the file system, alternative methods must be employed to determine the correct name of the object before it can be added. However, this is not a major problem for three reasons. First, automated tools such as SEER [Kuenning 1997] will often be the primary clients of selective replication. SEER performs predictive hoarding of data for mobile computers; it watches the user's file system activity and ensures that the "important" objects get replicated on the user's portable prior to disconnection. Doing so relies on internally storing the pathnames of objects; thus, SEER knows the pathname information and can provide it to the selective replication tools. One could argue that most, if not all, automated clients of selective replication must internally store knowledge of the filenames in question, and can therefore supply them when needed to the selective replication facility.

Second, when users are invoking selective replication mechanisms themselves, they are doing so because they have knowledge external to the system about non-local data objects. Via out-of-band communication or external inference, users discover the existence of data objects they want locally stored. Given external knowledge about the file, the user either already knows or can easily determine the object's name.

Finally, if required, a special tool could be used to inspect the replication facility itself and gather information, specifically pathnames, regarding files that are not locally stored. Clearly it would be more desirable to incorporate this tool with the standard method of viewing the file system. However, since our goal is a replication facility that resides outside the kernel, we cannot, in general, alter the standard method of inspecting the file system. We must therefore rely on the special tool when the above two scenarios do not apply.

6.2 Design and Implementation

This section describes the overall selective replication design. Several important design characteristics are worth noting at the outset:

- We manage both files and file groups (volumes) for several reasons described below
- Replication control information is itself managed optimistically, resulting in considerable simplifications
- Replication decisions can be performed at any replica; replica `R` can make decisions on behalf of replica `S`, even without being connected to `S`
- Locally stored data is always available, even during complete network disconnection
- We provide local knowledge about the portions of the namespace that are not stored locally

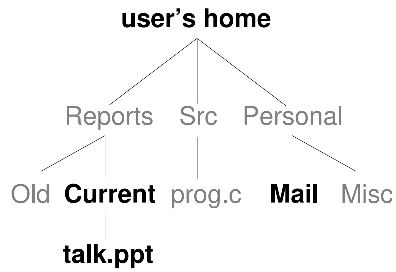

Figure 6.1: A partial volume replica. Files listed in bold are stored locally; the others are not.

Each of these items will be addressed below, including motivation, design impact, and implementation effects. The section concludes with a description of the replication policies and tools.

6.2.1 A two-level granularity structure

Instead of a file-granularity design, ROAM maintains the volume as a coarse-granularity abstraction and permits fine-grain file replication within each volume. Volumes provide several benefits, described elsewhere (Chapter 3), but users, especially mobile ones, require the flexibility of fine-grain replication for performance and correctness issues. We therefore provide the ability to selectively replicate individual files within the volume. *Partial volume replicas* maintain only the data structures and replica information for the selected portion of the volume which they store. [1] Figure 6.1 depicts a partial volume replica example.

The design yields a two-level granularity architecture that we believe provides efficient results. System functionality is logically separated by granularity: functions that should operate on a large-granularity object for performance reasons, and those that need to be finely controllable for flexibility requirements. For example, consider the security problem. Rather than individually protecting each file and restricting access to it, we can more easily and cleanly erect a single security perimeter around the volume as a whole. Since the volume has a single entry point, we can guarantee security for the entire volume by enforcing protection once at the volume root. However, the actual replication controls must be fine-grain to provide the flexibility, correctness, and cost-minimization characteristics that users require, and therefore must operate on individual objects within the volume.

6.2.2 Maintaining replication information

Sites need to determine which files are stored at which replicas to maintain data consistency and perform replica selection. [2] Many different solutions are possible. For example, central "managers" could record the storage locations for all files within each volume. Alternatively, primary site controls could be employed such that each object or set of objects has a specific authoritative site that knows which other sites store replicas. However, centralized schemes and replicated schemes based on conservative protocols (like primary site) place restrictions on when replication factors can change. For the same reasons that optimism is required for data updates, optimism is also required for metadata updates, including the physical locations of objects. Therefore we utilize an optimistic strategy for maintaining replication information at each replica. Like all optimistic strategies, this approach operates smoothly in the common cases and resorts to special actions in the face of failures—i.e., when the replication information is incorrect.

Each replica maintains a *status vector* for each locally stored file. The status vector consists of N elements, where N is the number of volume replicas in the particular ward. Each element is a storage *status* value indicating what information the corresponding volume replica stores about the file. Thus, the basic

[1] The volume root must always be stored locally.

[2] Even though ROAM does not provide transparent remote access and replica selection, we provide hooks for such mechanisms to be built.

Replica	Status Vector
1	{(1, *data*), (2, *nothing*), (3, *data*)}
2	none stored
3	{(1, *data*), (2, *nothing*), (3, *data*)}

Table 6.1: A status vector example. Of the three volume replicas, the file in question is replicated at only two of them (replicas 1 and 3). The status vector is shown as an array of tuples {volume replica identifier, status value}. Note that replica 2 stores no status vector for the object because the object is not stored at replica 2.

Replica	Status Vector
1	{(1, *data*), (2, *data*), (3, *data*)}
2	?
3	{(1, *data*), (2, *nothing*), (3, *data*)}

Table 6.2: An example of conflicting status vectors. The status vector is shown as an array of tuples {volume replica identifier, status value}. Replicas 1 and 3 have conflicting notions concerning the status of the object at replica 2. Additional mechanism is required to resolve the conflict. See Section 6.2.2.

status vector structure is an array of length N, with each array element being a tuple of the form {`volume replica identifier, status value`}.

A particular status element can have one of several values. The two most user-visible values indicate whether or not file contents are stored at the particular volume replica, referred to as the *data* and *nothing* status values, respectively.[3] For example, the status vectors for a given file at each of three volume replicas are illustrated in Table 6.1. Each storage site records its view of where copies of the file reside, and sites that don't store the file (volume replica 2) have no storage overhead. Status vectors are only locally maintained for locally stored files.

Since status vectors are independently and optimistically maintained structures, partitioned updates (i.e., changes in who stores what) can generate inconsistencies, resulting in *conflicting* status vectors. An example of conflicting status vectors can be seen in Table 6.2. Additional mechanism is needed to identify the true status value for replica 2. Not even replica 2 can resolve the conflict, since to permit users dynamic and free decisions regarding where to store files, we allow any replica `R` to modify any other replica `S`'s status vector. Selective replication is intended as replication control and performance enhancement, not as a security mechanism. Thus, in Table 6.2, replica 2 could have out-dated information with respect to its own status value.

It is important to allow the flexibility for replica `R` to modify replica `S`'s status vector independent of communication with `S`. First, the flexibility allows a more general-purpose design. More importantly, however, the flexibility is often required for correct behavior. Often a user at one replica *needs* to make changes at another replica. For instance, two users may be temporarily working from the same replica when one realizes that he'll need the same data stored at his (currently inaccessible) replica. Alternatively, a group of users may be working on a common project. When one includes new data files, she must ensure that the new data files are correctly stored at all participating sites. Finally, system administration duties can easily be made from one site, instead of having to independently and serially access each site, some of which will be unavailable.

The conflict is resolved by applying a version vector [Parker *et al.* 1983] approach to the status vector. Each element is implemented as a monotonically increasing counter, like the version vector. The low-order bits are used to indicate the actual status; the high-order bits implement the counter. With this approach,

[3] Additional status values will be discussed in the following sections.

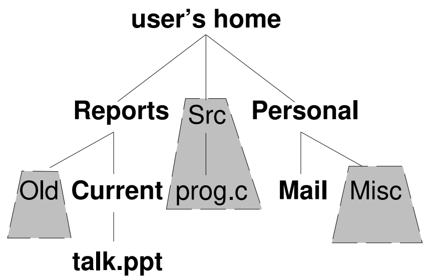

Figure 6.2: Full backstoring as applied to the partial volume replica from Figure 6.1. To store the `talk.ppt` and `Mail` files, the intermediate directories must be stored as well. Shaded subtrees are not stored locally.

two status values can be compared using the standard integer "greater-than" function. Conflicting status vectors can always be automatically resolved by performing a pairwise comparison of status values and selecting the larger one in each case.

While the counter-approach provides automatic conflict resolution, it has its drawbacks as well. One replica could force a particular status value by continually incrementing its counter to an arbitrarily large number. However, the 100% correct solution requires that each status value have an accompanying version vector of its own, which would cause the data structure to occupy $O(N^2)$ space, where N is the number of replicas. Our counter-approach has proven acceptable in practice, and seems like a good tradeoff of functionality for data structure size.

6.2.3 Local data availability

The set of all locally stored files at a given volume replica forms a forest of trees. If the forest is permitted to be disconnected, communication with another volume replica would be required to traverse through the intermediate directories, not stored locally, that logically connect the disconnected trees. During periods of network partition or disconnection, locally stored data would no longer be accessible, forcing users to pay close attention to the physical mapping of the tree structure onto the set of volume replicas.

To ensure local availability, the selective replication implementation automatically enforces *full backstoring*. This invariant requires that each locally stored file have its parent directory stored locally as well (the invariant does not apply across volume boundaries). Figure 6.2 illustrates full backstoring as applied to the partial volume replica from Figure 6.1. To store the prepared PowerPoint™ presentation (`talk.ppt`) and the week's mail (`Mail`), the user must store the intermediate directories (`Reports`, `Current`, `Personal`). However, there is no need for any files under `Src`, `Misc`, or `Old`, so these subtrees are not stored locally.

A common alternate solution employs *prefix pointers* [Welch *et al.* 1986], a second directory structure used to connect the user's disconnected namespace. While prefix-pointer solutions avoid storing intermediate directories, they must maintain an independent directory structure and integrate it into the user's namespace. Full backstoring avoids the complexity of dual directory structures, and is therefore simpler to implement. Additionally, full backstoring utilizes the system's directory structure and therefore requires no changes to application libraries, unlike the prefix-pointer solution. Since directories are typically much smaller than files, full backstoring consumes only a small percentage of the available disk space.

Full backstoring is not a complete panacea, however. Both Unix hard links and the `rename` system call pose potential problems for full backstoring. The problems and their solutions will now be explained.

Multiple hard links

Unix hard links pose problems for full backstoring, because they allow the assignment of multiple names to a single file object. Given one file name, it is difficult to find all hard links to the same file without scanning all volume replicas. Therefore, the system only guarantees the full backstoring of one link; additional paths

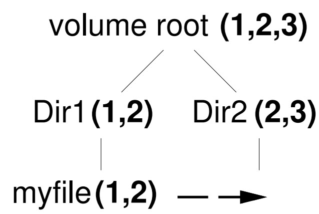

Figure 6.3: An example illustrating the problem with renaming objects in a selectively replicated environment. Each file system object is appended with a list of numbers in parentheses indicating what replicas physically store the given object. When `myfile`, which is stored at replicas 1 and 2, is moved to its new location under `Dir2`, the full backstoring invariant is no longer preserved.

can be fully-backstored at the user's request using the replication tools discussed in Section 6.3. However, this problem is not serious in practice, because hard links are rare, and many modern systems do not even support them. Symbolic links, which are far more common, cause no problems for full backstoring.

Rename system call

When a file object is renamed from one portion of the namespace to another, replication factors may have to be altered to preserve full backstoring. Consider the situation in Figure 6.3. When `myfile` is moved from `Dir1` to `Dir2`, a mismatch arises in replication factors—`myfile` is stored at replicas 1 and 2, while `Dir2` is stored at replicas 2 and 3. Full backstoring is therefore broken at replica 1, and must be restored to guarantee data availability.

There are basically three methods of resolving the problem:

1. The replication factor of the object being moved could be altered to match the new position in the namespace; that is, the resulting replication factor of `myfile` is exactly what would occur if it was a new file created in the new position. This is depicted as Solution `A` in Figure 6.4.

2. The replication factor of the object could remain unchanged, and the intermediate directories between the new position and the volume root could be changed to support the full backstoring invariant. This is depicted as Solution `B` in Figure 6.4.

3. A combination of approaches one and two, resulting in changing the replication factors of both the file object and the intermediate directories, depicted in Figure 6.4 as Solution `C`.

We omit all solutions that are violations of the file system semantics, such as creating two separate and independent files.

Solution `A` adapts the file to its new position in the namespace. While it restores full backstoring in a straightforward manner, it has the unfortunate consequence that certain replicas may suddenly have their replica deleted without warning. Replica 1 suffers from this problem. It unknowingly loses `myfile` after replica 2 moves it to `Dir2`. (Replica 2 is the only replica that could have moved the file, because replica 1 does not store the target directory, and replica 3 does not store the object in question.)

Selective replication is not intended as an information-hiding or security mechanism within the volume. Therefore there is no "good" reason to have it silently disappear at replica 1. In fact, since replica 1 stored the object, there is a good chance it would desire a copy of it under the new name as well.

Solution `B` and `C` are related in that no existing storage site loses its replica; they differ in who should store it given its new position in the namespace. Solution `B` keeps the object's replication factor the same. Solution `C` takes into account the new position in the namespace and adds replicas accordingly, as if the object were newly created in the new position. Both have their advantages and arguably either is correct. However, it seems that solution `B` is more appropriate, given that the new storage sites added in Solution

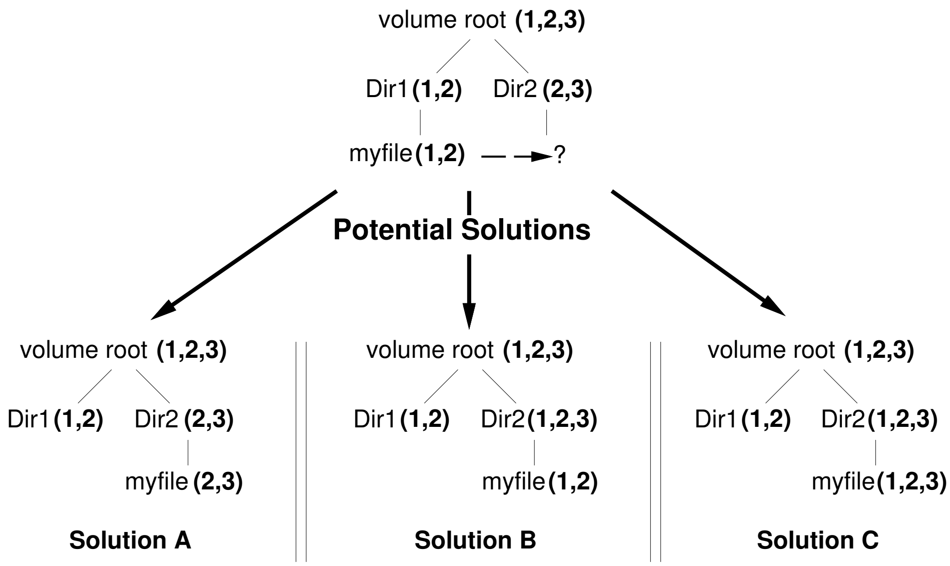

Figure 6.4: An example illustrating the potential solutions to the rename problem from Figure 6.3. Each file system object is appended with a list of numbers in parentheses indicating what replicas physically store the given object. Solutions `A`, `B` and `C` indicate three possible ways to enforce the full backstoring invariant.

`C` (e.g., replica 3) did not believe the object important enough to originally store under its old name. Its position in the namespace does not affect or change this decision. Baring new information, solution `B` is the most correct given only the existing information regarding what replicas deem the object `myfile` "interesting."

Unfortunately, the existing RUMOR implementation suffers from undue complexities in the actual code where rename operations are detected, making the implementation of either Solution `B` or `C` extremely difficult, if at all possible. Were we to re-implement the underlying replication system from scratch, solution `B` would be the desired approach. However, the current implementation utilizes solution `A` in an effort to speed and simplify development.

It should be noted that the equivalent problem occurs when moving objects between volumes, since volumes are replicated independently of one another. However, this problem is significantly different in that volumes are often used as security perimeters and boundaries. Moving data from one volume to another may, in fact, be an attempt at data hiding. Additionally, volumes are heavyweight objects, and creating new volume replicas is a larger and more heavyweight task than adding full backstoring for a different path in a given volume. As such, the equivalent of solution `A` is, in fact, the appropriate functionality when moving objects between volumes.

6.2.4 Namespace maintenance

Selective replication allows individual files to be stored at any physical volume replica; specifically, files may be stored at fewer locations than those of their parent directory. Users who only replicate a particular subtree (or subtrees) locally store only the file system information corresponding to that subtree (or subtrees), such as in Figure 6.2. However, continuing with the example, the given replica does not know the physical location of the other subtrees, such as the `Src` subtree. We believe it is essential that access to remotely stored data be transparent when network connections exist.[4] Selective replication should maintain name

[4] Transparent remote access is not currently possible because the underlying implementation of RUMOR does not presently support it. Nevertheless, the selective replication implementation maintains enough local information to support it.

transparency.

We use an additional status value to locally maintain just the status vector for a particular file without storing file data. Since the status vector indicates which volume replicas store file data, it provides a fast and easy method of locating file replicas; additionally, the status vector is itself small, so it imposes little storage overhead. The new status value is referred to as *attributes-only*.

The system maintains the invariant that, for all locally stored directories, all directory entries are also stored locally in at least the *attributes-only* state (they could be stored in a *data* state). Thus, in Figure 6.2, `Src`, `Misc`, and `Old` would be maintained locally in an *attributes-only* state. Nothing would be stored for the file `prog.c`; it does not need to be maintained in an *attributes-only* state. Storing the status vector for `Src` is enough to locate a replica of the subdirectory, and therefore a replica of `prog.c`.

The status vector is used only as a hint to the physical storage locations. Since it is optimistically and independently maintained, partitioned updates can create temporary inconsistencies. We can either attempt to maintain consistency on the *attributes-only* replicas or treat them like a cache. Our implementation does both. Consistency is periodically maintained on these *attributes-only* replicas; however, since consistency is maintained optimistically, there are still scenarios where the *attributes-only* replicas are inconsistent. We therefore treat the replicas like a cache, and resort to a multi-level polling and cache-replacement solution if the cache is invalid [Ratner 1995]. Treating the *attributes-only* replicas like a cache means that there is zero overhead or performance impact during the normal reconciliation case. An occasional, special reconciliation maintains periodic consistency.

The directory-entry invariant basically requires that directories are always *entirely replicated*. One cannot store only "part" of a directory; if a directory is locally stored, then the entire directory including all directory entries are locally stored. Selective replication simply allows control over which of those directory entries have physical objects locally.

6.3 Replication Controls

A volume-based replication service has a single policy with regard to file creation: all files are replicated at all volume replicas, known as *full replication*. As such, the only necessary controls are those that create, add or delete volume replicas. In contrast, robust selective replication implies multiple policies regarding file creation. Furthermore, users' desires and data demand patterns change, implying the necessity for dynamic addition and deletion of file replicas. This section discusses replication policies as well as the selective replication utilities.

6.3.1 Replication policies

Reasonable default replication policies are necessary both for novices and more sophisticated users. By default, one could store new files wherever the parent directory resides, which has proven reasonable in practice. However, since volume roots are always fully replicated, their children, both directories and files, would be created fully replicated as well. In the absence of further mechanism, everything would be fully replicated. Therefore, we provide user-specified replication *masks*, or mappings onto the set of volume replicas. On a per-directory basis, users can specify where that directory's children should be stored, constrained only by the full backstoring invariant. For instance, if a directory were replicated (i.e., physically stored) at replicas 1, 2, and 3, the replication mask provides a "yes/no" bit for each replica, indicating whether or not new children created under the directory should be stored at the corresponding replica. Since the replication mask is itself a replicated data structure (the implementation folds the "yes/no" bit into each status value) the resulting replication pattern is equivalent regardless of what replica creates the new file. Table 6.3 illustrates an example.

Changing the replication mask does not affect the storage status of existing children in the directory. Additionally, the user can override the mask by physically generating a new file at a replica that is marked "no new children." Even though it is marked as not storing new children, since the user explicitly created

Replication Mask	Result of Object Creation
{{1,yes}, {2,no}, {3,yes}}	New objects stored only at replicas 1,3
{{1,yes}, {2,yes}, {3,yes}}	New objects stored at replicas 1,2,3
{{1,no}, {2,no}, {3,yes}}	New objects stored only at replicas 3

Table 6.3: An example of the use of replication masks. The masks are shown as an array of tuples {volume replica identifier, mask bit}.

Replication Mask	Result of Object Creation at Replica 1
{{1,yes}, {2,no}, {3,yes}}	New objects stored only at replicas 1,3
{{1,yes}, {2,yes}, {3,yes}}	New objects stored at replicas 1,2,3
{{1,no}, {2,no}, {3,yes}}	New objects stored at replicas 1,3

Table 6.4: Object creation takes into account the replication mask as well as the physical replica that created the object. The ideal depicted in Table 6.3 cannot always be enforced.

the object at that replica, we override the mask and include that replica as a data-storing site. Table 6.4 illustrates the scenario.

The mask solution is a simple approach that provides powerful control in many scenarios, but clearly does not solve the entire problem. For example, a software development directory may contain both source and object files, and the user may wish to only store source files at specific replicas. Optimally, a configuration file should be provided that allows users to specify where files should be replicated based on regular expression pattern matching on the name. In the above example, the configuration file could be used to indicate that specific replicas do not store files matching the typical pattern for compiled objects, such as {`*.o`}. Similar to update/update conflict-resolution files [Kumar *et al.* 1993, Reiher *et al.* 1994], these configuration files would be specified on a per-directory or per-volume basis. A per-directory specification seems to make more sense, as the mapping remains with the directory whenever the directory is moved, and the regular expressions are generally shorter.

Unfortunately, the existing implementation does not provide the optimal level of control. For a variety of reasons, incorporating such a configuration file would require a major redesign of much of the underlying Rumor code: specifically the code that handles new-file detection. We have therefore omitted it from the implementation, though it is an important component of the complete solution, and one which would be included had we the time to redesign the underlying Rumor implementation.

6.3.2 Replication policies between wards

With the Ward Model, there are two opposing camps with respect to the selective replication user model. The first camp says that the Ward Model should be transparent to the user, and therefore the introduction of wards should not change the basic user model—it should remain exactly as described above. The second camp argues that wards represent a fundamental change, and that where users want objects replicated changes as the users change their physical geographical position.

We believe that the Ward Model should be transparent to users, and especially transparent to the replication decisions with respect to where objects are physically stored. Both selective replication and the Ward Model are not mechanisms for information hiding or security/privacy features; these should be handled in other ways, such as creating additional volumes for secure information and using security-enhancements like Truffles [Reiher *et al.* 1993b]. If the user wants different replication behavior based on physical geographic location, this is best handled with tools that query the Ward Model for information and supply the replication system with specific decisions, rather than building a more rigid structure into the Ward

Model itself.

The Ward Model therefore does not impact the above replication policies at all. The default behavior and replication masks still apply, abiding by the structure of the Ward Model. Replicas specify the replication masks only within the ward (or wards) to which they currently belong. Ward masters are responsible for storing the inter-ward replication mask information.

6.3.3 Dynamic replica deletion

Users may also need to dynamically change the replication factors of individual files; that is, *delete* and *add* file replicas.[5] These actions can occur locally or remotely on another replica's behalf. That is, we allow replica `R` to specify that a given file object should be added or dropped at some other replica `S`. Since the addition or deletion occurs optimistically, `S` may not be knowledgeable of the action until some time later, when it learns of it via synchronization. We will discuss the more complicated case of deletion first, followed by that of addition.

There are three basic deletion issues: the potential loss of updates, the resolution of the create/delete ambiguity [Fischer *et al.* 1982] and the removal of full backstoring information. All of these are examined below.

Potential loss of updates

Replica deletions should only affect storage locations, not the file's existence or physical data. Note that a replica deletion is different than physically removing a file. When removing a file, the file's data should be reclaimed; we *want* the operation to affect the file's existence—it should no longer exist. When removing a file replica, we only want to remove one physical storage location; we do not want the operation to remove the file at all locations. Guarding against data loss (removing a version that is not the globally latest version) is handled by the remove/update detection mechanism, explained in Chapter 7.

To guard against data loss in the file deletion case, we must ensure that another replica stores a version greater than or equal to the one being deleted. In general, it is not possible to determine which replica has the most recent data given only local information. We therefore guarantee safety by requiring that another replica first reconcile the file with the replica being deleted. Called a *reverse reconciliation*, this process is not always possible, however, as network partitions can make other replicas temporarily inaccessible. Nevertheless, it is sometimes necessary to allow replica deletions in these cases, despite the potential loss of updates. When disks become full, all work ceases, and often the possibility of losing an update is less important than the certainty of not being able to continue working. We therefore allow users to bypass reverse reconciliation when required, though the practice is not recommended for novices.

Reverse reconciliation also guards against accidently dropping the last remaining replica. The semantics of a replica deletion provide that data loss never occurs: only the file *replica* is removed, not the file itself. As such, we disallow dropping when the replica being dropped is the last remaining replica.

However, the user may not be aware that the replica is the last in existence. Additionally, without reverse reconciliation, the system itself may be unaware. For instance, if replicas 1 and 2 were the last two replicas, and 2 dropped its replica without contacting replica 1, then replica 1 incorrectly believes that replica 2 still stores a physical replica. Requiring a reverse reconciliation provides a correctness guarantee that the last remaining replica will not accidently be dropped.

Create/delete ambiguity

Even with reverse reconciliation, replica `R` may be unaware that replica `S` deleted its replica; perhaps because replica `S` performed a reverse reconciliation with replica `T`, and the information has not yet propagated to `R`. In such environments, where distributed actions are performed without global consent, the potential for create/delete ambiguities arises [Fischer *et al.* 1982]. The general description of a create/delete ambiguity

[5] "File" in this context refers to all file system objects, including directories and symbolic links.

Replica R	Replica S	Time
drop directory `foo`	—	t_1
—	create `foo/new`	t_2
reconcile with `S`	—	t_3

Table 6.5: A series of operations at two replicas, resulting in a new object being created at replica `S` that thinks it should also be stored at replica `R`, though `R` has dropped its replica of the parent directory.

is that in the absence of additional information, object deletion at one site appears indistinguishable from object creation at another. In the context of file replica deletion, the ambiguity manifests itself as follows. A file **`problem-file`** has two replicas, `R` and `S`. Unbeknown to replica `S`, we delete `R`'s replica. (Perhaps `R` performs reverse reconciliation with some third replica.) When replicas `R` and `S` communicate, two scenarios are possible:

1. **`problem-file`** could be a new file created at `S` (to be stored also at `R`)

2. **`problem-file`** could be an existing file whose replica was previously deleted at `R`.

The inability to distinguish the two cases is the essence of the create/delete ambiguity. Replicas `R` and `S` cannot decide if the file should be added by `R` or was at some point in the past deleted by `R`.

Guy [Guy *et al.* 1993] describes a solution for resolving the create/delete ambiguity in the context of garbage collection of distributed file system objects; we adopt his solution here. Guy maintains a temporary record of the action and executes a distributed algorithm to guarantee both that all replicas learn of the action and that the record is eventually removed. In our case, the record of the action is a new status, in addition to the three already mentioned, called the *dropping* state. When one or more file replicas change state to *dropping*, all data-storing replicas participate in a distributed algorithm. The *dropping-notification* algorithm guarantees that all replicas correctly learn of the transition by one or more replicas into the *dropping* state. The algorithm also guarantees that, after everyone has learned of the transition, the state changes from *dropping* to *nothing*. When sites change their local state to *nothing*, they remove the entire status vector, thus cleanly completing the replica deletion.

Create/delete ambiguities can also occur on an object's children if the object being dropped is a directory. Consider what happens when replica 1 deletes its local replica of a directory at the same time that replica 2 creates a new child underneath that directory, depicted in Table 6.5. The new object **`foo/new`** is marked as being stored at replica 1, although replica 1 no longer stores the parent directory. To correctly maintain the full backstoring invariant, either the parent directory needs to be re-stored at replica 1, or the new object **`foo/new`** needs to know that replica 1 does not store a replica. Note that the child's create/delete ambiguity only occurs for newly created children. The children that were known at replica 1 when it dropped the directory were marked as dropped at the same time.

The solution employed tracks the progress of the dropping-notification algorithm and during reconciliation explicitly informs remote replicas that may not be knowledgeable of the drop action. Recall that the dropping-notification algorithm has two phases. The first phase guarantees that all replicas correctly learn of the transition to the *dropping* state, and the second phase guarantees that, after everyone has learned of the transition, the state changes from *dropping* to *nothing*. Our solution uses the knowledge of the two phases, and during reconciliation explicitly informs sites that are not marked as having completed phase 1. We notify these sites that the directory has been dropped, so each remote site can correctly modify the data structures for any new children of the directory.

Full backstoring

When deleting a file replica locally, we want to additionally remove any of the intermediate directories that were added to support local availability, as described in Section 6.2.3. Without removing them, the local

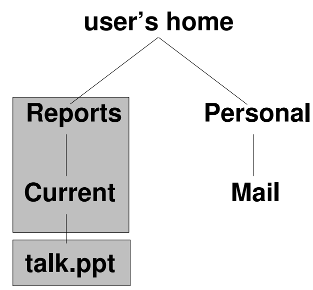

Figure 6.5: When dropping the local replica of `talk.ppt`, the local replicas of directories `Current` and `Reports` are also dropped, since they are only locally maintained for full backstoring and local data availability.

directory structure would eventually become fully replicated, which is contrary to the goals and motivations for selective replication. Therefore, we inspect the file system tree from the deleted file to the volume root, and any directories that are locally replicated solely for full backstoring support are deleted as well. Figure 6.5 illustrates an example. The full algorithm for dropping the required path is in [Ratner 1995].

6.3.4 Dynamic replica addition

Adding file replicas is simpler than deleting them, because there is no risk of data loss, and there is no create/delete ambiguity to resolve (by design, the lack of a delete record implies add). The full backstoring invariant (Section 6.2.3) must be maintained, however, as otherwise the new data could be inaccessible during network partitions. Therefore, all intermediate directories on the path between the volume root and the to-be-added file must be added as well. Fully backstoring the intermediate directories is trivial, however. The full backstoring invariant implies that the remote site which actually stores the file in question must also store the intermediate directories, so the entire action can be accomplished by communicating with only one other replica. It does, however, have performance implications when performing the addition over a low bandwidth or otherwise poor quality network.

6.4 Consistency Maintenance

Reconciliation details have already been described in Chapter 5, but that description purposely avoided selective replication issues. Without selective replication, any two communicating sites are guaranteed to store the same set of objects. (If some site `S` doesn't currently store a specific object, it is a transient situation caused because `S` is ignorant of the object's existence, as opposed to an explicit statement that `S` is not a storage site.) With selective replication, it may no longer be possible to reconcile all of one's data with just one other replica; multiple replicas may have to be queried to synchronize each local object with some remote replica of that object. This section describes the details regarding synchronization between two selectively replicated volume replicas. It is broken into two sections: pre-reconciliation startup and reconciliation processing. A discussion of the reconciliation topology can be found in Chapter 4. We conclude with a state diagram indicating the flow of control and highlighting the additions and modifications made to support selective replication.

6.4.1 Pre-reconciliation startup

Chapter 5 describes how each volume replica scans its file system, infers the user's modifications, and generates filelists summarizing the state of the volume. The local replica, or target, constructs a filelist containing everything it locally stores. This filelist is the list of objects to be reconciled. The remote replica, or source, constructs a filelist of only those objects that it stores *and* it believes the target replica stores as well; i.e., those objects that it believes are stored in a *data* state at the target. Objects participating in create/delete resolution algorithms, such as those being garbage collected and those in the *dropping* state, are included as well if the target replica needs to participate. Having the source replica send attributes for objects not relevant to the target replica is simply a waste of bandwidth, and therefore a waste of both time and money to the user.

The source replica uses its local status vectors to make decisions regarding the objects stored at the target replica. As discussed earlier, since the status vector is optimistically maintained, inconsistencies may arise. The inconsistencies are detected and resolved during the reconciliation process, described below.

6.4.2 Reconciliation processing

Once the target replica has both filelists (the target or local filelist and the source or remote filelist), it begins to reconcile the volume. Entries are removed from the remote filelist in series, the corresponding entry is found on the local filelist, and the two are compared. There are three cases to consider:

1. The object may exist on both filelists
2. The object may exist only on the local filelist
3. The object may exist only on the remote filelist.

The actions taken in each case are discussed in turn. When the last entry on the remote filelist has been processed, reconciliation with that particular remote replica is complete. If all entries on the target filelist have been reconciled at least once, then the local volume is completely reconciled. Otherwise, another remote volume must be queried, and the process repeats. The choice of a remote volume is made according to the synchronization topology discussed in Chapter 4.

Both filelists

If the object exists on both filelists, the two replicas can be directly compared. We compare the two sets of attributes and perform tasks such as version vector and other attribute comparisons. By comparing the attributes, the process reaches a decision regarding which replica has the most recent data and other relevant attributes. The resulting decision can be one of four possibilities:

- The two replicas have identical data and attributes
- The local replica has data and/or attributes that are more recent than the remote replica; the local replica is said to *dominate* the remote replica
- The remote replica has data and/or attributes that are more recent than the local replica; the remote replica *dominates* the local replica
- Both replicas have updates that are unknown at the other; the two replicas are said to *conflict*

If the two replicas are identical, the object is ignored and reconciliation proceeds with the next entry from the remote filelist. If the local replica dominates the remote replica, no action is taken, because reconciliation is a pull-only process. The remote replica will eventually obtain the new data by initiating its own reconciliation process. In the two remaining cases (remote replica dominates local and the replicas conflict), data is obtained from the remote replica. A data-request message is sent to the local *server* process asking it to obtain data for the given object. The local server communicates with the remote server and eventually receives and installs the data. If the remote replica dominated the local replica, the data is installed in place of the existing, out-dated version. If the two replicas conflict, the data is preserved in a hidden place for use by the conflict resolution tools [Reiher *et al.* 1994].

In the cases where the data is the same and only the attributes need updating, we perform the local modification without requesting data or any additional information from the remote site.

Local filelist only

Objects on the local filelist may not have a matching remote counterpart. The scenario can arise for one of three reasons:

1. The object has completed a distributed create/delete resolution algorithm at the remote site and no longer exists there. The two distributed create/delete resolution algorithms are garbage collection (Chapter 7) and the dropping-notification algorithm (Section 6.3.3).

2. The file is not known at the remote replica; it has not yet learned about the object.

3. One of the two status vectors at the replicas contains old information for the target and source replicas. Either the target (local) replica or the source (remote) replica is incorrectly constructing the intersection of the two replica sets, because their local information is out of date.

The completion of a create/delete resolution algorithm can be detected using local information regarding the algorithm status. Both algorithms are two-phase in nature and provide that no replica `R` completes until all replicas know that replica `R` has at least started the algorithm. Additionally, using only local information the local replica can detect whether or not an object should be participating in one of these two algorithms. Therefore, given the algorithm invariant and the local state information, the local replica can decide if the remote replica completed one of the distributed algorithms (case 1) or if the remote replica is simply unknowledgeable regarding the object (case 2). If the algorithm completed at the remote replica, the local replica completes it as well. If the object is unknown to the remote replica, the local replica does nothing; the remote replica will learn of the object's existence when it initiates reconciliation.

The inconsistencies in the status vector must also be resolved (case 3). The selective replication implementation in Ficus dynamically requests the attributes of the remote object and simply compares the two status vectors [Ratner 1995]. The dynamic request is possible and feasible because Ficus is a distributed file system and as such has access to the remote site's file system. Roam (and Rumor) do not have this capability. Additionally, the dynamic request makes implicit assumptions about the transport mechanism: namely that it is fast and inexpensive with low latency. Mobile computing encounters various different types of communication medium, with varying degrees of bandwidth, latency, and cost. The system must adequately handle the entire gamut of transport possibilities. We therefore designed a new solution.

The new solution allows the inconsistencies caused by case 3 to be resolved using only local information. We add to the remote filelist objects stored at the local site in *attributes-only* states. In other words, the source sends not only objects it believes are stored at the target in *data* and *dropping* states, but also those that it believes are stored in an *attributes-only* state as well. By doing so, the target knows that any file objects expected but not sent by the source must be unknown to the source—that is, the source has not yet learned of the object's existence. In Ficus there were two possibilities: either the local replica could have incorrectly thought the remote replica stored data, or the remote replica could have incorrectly thought that the local replica was not interested in the status of the object. The new solution removes the first possibility from ever occurring, because for the remote replica to *stop* storing data it must change its state to either *attributes-only* or *dropping*, both of which are included on the remote site's filelist. If the remote replica does, in fact, store the objects but believes them irrelevant to the local site, the remote replica will update its status vector when it learns the new information via its own eventual reconciliation. Therefore, there is no confusion in case 3. It is either the case that the source replica is ignorant of the object or that the source replica does not believe the target replica is interested in the object's status, and both scenarios are remedied when the source replica initiates its own reconciliation. The local (target) replica need not take any action.

The solution provides reconciliation correctness at the added expense of sending additional attributes from source to target. However, the additional overhead is minimal, because replicas are only maintained in *attributes-only* states for one directory level below what is physically stored on disk. Additionally, the size of the *attributes-only* replicas is fairly small. Furthermore, the solution has two major advantages. It provides reconciliation correctness without the dynamic request for attributes that Ficus uses, and it provides a built-in mechanism for the reconciliation of *attributes-only* replicas.

Remote filelist only

Lastly, the object may exist only on the remote filelist and not have a matching local counterpart. Like the local-filelist-only case, the scenario can occur for one of three similar reasons:

1. The object has completed a distributed create/delete resolution algorithm at the local site and no longer exists there.
2. The file is not known at the local replica; it has not yet learned about the object.
3. One of the two status vectors at the replicas contains old information for the target and source replicas. Either the target (local) replica or the source (remote) replica is incorrectly constructing the intersection of the two replica sets, because their local information is out of date.

Note that the three reasons are identical to those in the local-filelist-only case with the role of the local and remote sites reversed. Each of the three scenarios is resolved in a similar manner as in the local-filelist case.

Cases 1 and 2 are handled in an identical manner to that described in the above local-filelist-only case. The only difference in the processing of case 3 concerns the action performed by the local (target) replica. In the local-filelist only case, the target replica did not need to take any action in case 3, because the situation would be rectified by reconciliation at the source replica. However, here the target replica resolves case 3 in the opposite manner: it always requests data from the remote (source) replica and installs the object locally.

Recall that the source sends not only objects it believes are stored at the target in *data* and *dropping* states, but also those that it believes are stored in an *attributes-only* state as well. By doing so, the target knows that any file received but not expected must be an object it is supposed to physically store (except if the object falls into case 1 above). It could be a new object that the target replica is ignorant of, or it could be the result of a remote file replica addition (a replica addition physically performed at some unrelated replica but done on the target's behalf). In both cases the target replica requests data from the source and installs the object locally. As in the local-filelist-only scenario, we prevent case 3 from generating difficulties by requiring the source to include *attributes-only* objects in the filelist sent to the target.

6.4.3 Flow of control

The flow of control of the reconciliation process, as seen by the target replica, is illustrated in Figure 6.6. As discussed in Chapter 4 under reconciliation topologies, the target replica contacts remote replicas until either all of its local data has been reconciled or the set of accessible remote replicas has been exhausted. The additions required by selective replication are highlighted in bold, to distinguish them from the control flow diagram for the volume-only solution in Figure 5.4.

All information into and out of each replica first passes through the server on that replica. The request for and communication of the remote filelist is shown as a direct connection between the two processes, as that is the eventual state. Prior to this, however, the request for the remote filelist is given to the local server, which communicates with the remote server. The remote server initiates a process on the remote machine, and each server "hands-off" its half of the connection to the appropriate process, thereby removing itself from the communication path.

6.5 Selective Replication Summary

We have previously argued that peer models are necessary in a wide variety of environments, and perhaps essential for mobility. However, often the peer solutions are simply unusable in practice without selective replication, due to the inherent costs of peer-based models. Selective replication allows peer solutions to be effectively and efficiently utilized in scenarios that otherwise would prove impractical.

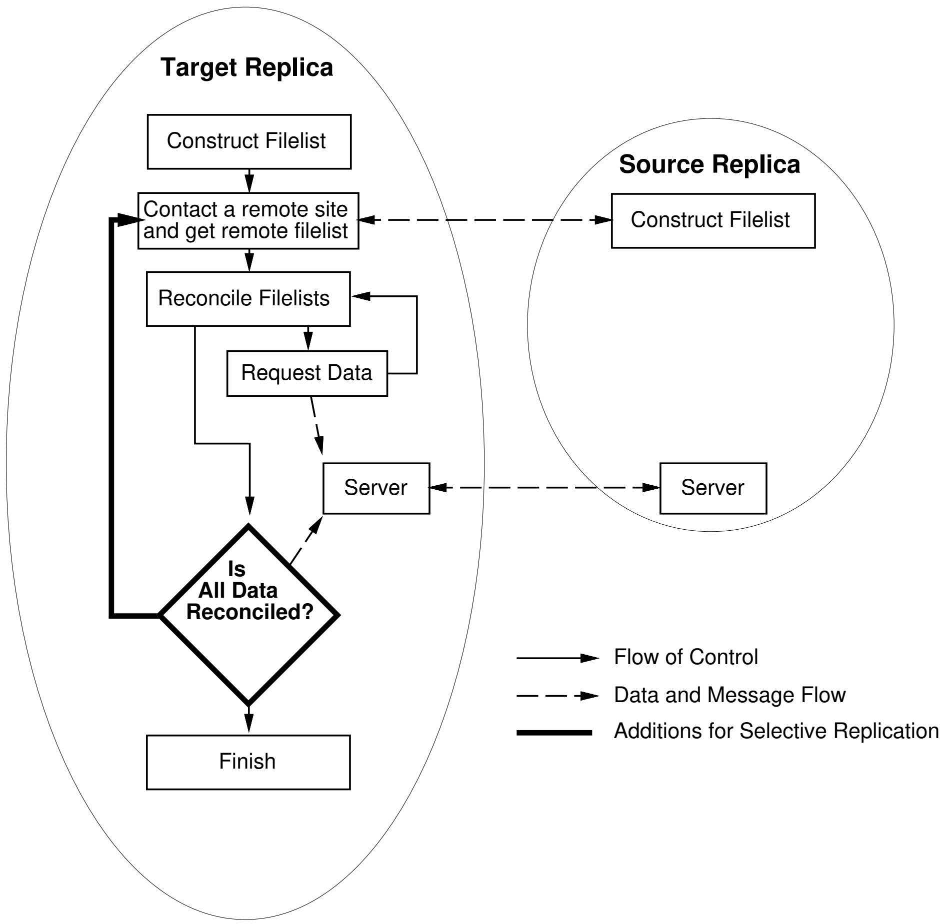

Figure 6.6: Flow of control diagram for a given reconciliation process, illustrating the control flow additions for selective replication support (marked in bold). The diagram is otherwise the same as the control flow diagram in Figure 5.4. Both control and data flow are indicated. Only one source replica is indicated, though multiple source replicas may be queried to reconcile all of the data stored at the target.

Furthermore, the success of file hoarding systems like SEER [Kuenning *et al.* 1997] in peer contexts absolutely requires the ability of the replication substrate to make selective decisions. SEER watches file accesses and asks a replication substrate to hoard the necessary files on a laptop prior to network disconnection, to ensure correct operation during the disconnection period. However, doing so requires that the replication substrate be capable of selecting individual files or sets of files from multiple volumes. Selective replication is therefore an integral part of any peer replication service designed for a mobile world.

Chapter 7

Garbage Collection

Garbage collection is the deallocation of file system resources following object removal. Resources such as disk space occupied by file data and the associated file system objects must be returned to the general resource pool so they can be re-used. Actual deallocation is straightforward, but determining the correct time to do so is difficult—there are both semantic and correctness issues. The semantics affect the user-observed behavior with regard to data loss, *dynamic naming*,[1] and long-term communication delays. Correctness issues occur because premature garbage collection causes the removal of user-visible data; at the same time, garbage collection must complete as quickly as possible, because the resources being collected cannot be re-used until the algorithm terminates. Garbage collection is therefore both an important and complex aspect of replicated file systems.

This chapter describes the garbage collection algorithm used in ROAM. It first describes our basic model of the file system, and then discusses in more detail the semantics and correctness issues behind garbage collection. We then describe a previous algorithm used in FICUS and RUMOR and its associated semantics, illustrating why it is inappropriate for mobile computing. We conclude with a discussion of a new algorithm and semantics, as well as an analysis of that algorithm, a proof of correctness, and algorithm pseudo-code.

7.1 File System Model

We adopt the same model as Guy [Guy 1991]. Briefly, the model provides users with a persistent storage service for a collection of files or objects. Each object has a logical name or set of names by which users access file data. Names can have different replication factors from each other and from the underlying object itself. New names can only be created from an existing name, although name creation occurs asynchronously and optimistically. Only one replica need be contacted to create a new name.

Name removal leaves an object inaccessible when no user-accessible name exists for the object. Since new names can only be generated using an existing name, a globally inaccessible object is permanently inaccessible.

We assume that Byzantine behavior does not occur. Long-term communication delays may occur, barring communication for indeterminate periods of time, but replicas themselves are always truthful in their responses.

7.2 Garbage Collection Semantics

A garbage collection algorithm provides specific semantics or guarantees to the user with respect to data loss. These semantics convey user-understandable knowledge about how the algorithm works and when physical

[1] Operating systems like UNIX allow the dynamic creation of multiple names to the same file with the `link` command. More importantly, most operating systems allow dynamic `rename` operations.

Replica R	Replica S	Time
create `name1`	—	t_1
—	learn about `name1`	t_2
—	rename `name1` to `name2`	t_3
remove `name1`	—	t_4

Table 7.1: A series of operations at two replicas, resulting in the object being locally inaccessible at replica `R` but still not globally inaccessible. Time in the table flows downward.

data will be reclaimed. For instance, consider the Ficus and Rumor *no lost updates* semantics [Guy *et al.* 1993]. These semantics guarantee that data will be preserved as long as the file is globally accessible, and that no replica will remove data until it becomes globally inaccessible. For example, if the object's most recent update exists at replica `R` but the only active name exists at replica `S`, the semantics ensure that eventually the name (known at `S`) will refer to the globally most-recent version of the data (known at `R`). Reconciliation itself guarantees that the new name is eventually known at all replicas, along with the proper data version. Since data is preserved at all replicas until the object is globally inaccessible, the *no lost updates* semantics provide the maximal level of global guarantees.

Other degrees of garbage collection semantics exist. The minimal semantics must provide basic correctness (i.e., avoid garbage collecting before the file becomes inaccessible), but need not provide any higher-level guarantees regarding when data versions are removed. Our new algorithm, as described below, will take a middle position between the two extremes.

7.3 Garbage Collection Correctness

Garbage collection must occur as quickly as possible because it has user-visible side effects, but must not complete before the object being collected is globally inaccessible. Removing the data and resources for a "live" object causes disastrous effects for users—they suddenly lose all data. Detecting when the object is globally inaccessible is therefore one of the main tasks of the garbage collection algorithm.

However, detecting global inaccessibility is often difficult, especially in optimistically replicated file systems. Dynamic naming and long-term communication delays imply that a file may be inaccessible at replica `R` and simultaneously accessible through a new name at replica `S`. Table 7.1 illustrates how such a situation could occur. `R` of course will eventually learn of the new name, but until that time `R` believes the object to be locally inaccessible. Local inaccessibility therefore does not imply global inaccessibility, and `R` cannot in general garbage collect until it can be demonstrated that the object is, in fact, globally inaccessible. Of course, when each replica can garbage collect also depends on the semantics of the particular algorithm.

Note that dynamic naming is different from physically copying the object in question and creating a second name or removing the file and creating a new one with the previous name. In both cases, copying and re-creating the same name, the resulting objects are physically distinct from the original object. Any relation, either by name or by data, is merely semantic and does not imply a causal relationship from the point of view of garbage collection of the original object.

Additionally, garbage collection must resolve the create-delete ambiguity [Fischer *et al.* 1982, Guy *et al.* 1993]. When an object exists at one replica and not at the other, it cannot be determined without additional information whether the object should be created at the second replica or removed at the first. Problems occur if the ambiguity is not resolved. Files that the user removed can re-appear in the namespace if the incorrect action is chosen (create instead of remove). On the other hand, new objects created at one replica will never propagate to the other replicas if the *other* action is always chosen (remove instead of create). The garbage collection algorithm, therefore, cannot complete independently at each replica; cooperation must exist to resolve the create-delete ambiguity during the entire process.

Furthermore, garbage collection must occur as quickly as possible. The resources being collected cannot be re-used until the process completes. In the case of disk space, the disk effectively shrinks during the process by the size of the garbage being collected plus any state data used by the algorithm itself. As such, users indirectly perceive the time garbage collection requires as a "usability" criterion of the system. Garbage collection inefficiencies are noticed by the users, who become dissatisfied with the system as a whole and tailor their activity to act in a "garbage-collection-friendly" manner. Personal experience with other replicated file systems such as Ficus and Rumor illustrates this point all too well—disks occasionally became completely full of "garbage" waiting to be collected, blocking all writes to the disk and sometimes requiring expert assistance to make progress.[2] Other people specifically tailored their activity to avoid the full-disk problem. For instance, some users would specifically truncate their data before removing it, causing incorrect behavior when used in conjunction with dynamic naming, and nullifying any benefit from the garbage collection semantics.

Finally, the garbage collection process must correctly identify and detect *remove/update* conflicts. Remove/update conflicts occur when a replica removes the last global link to a file while not storing the latest data version. We identify the latest data version after all replicas have learned of the removal of the last global link. Therefore the update to the data does not have to occur in real time before the name removal, only in virtual time before the updating replica learns of the name removal. We resolve remove/update conflicts by preserving the updated version of the data and allowing user access to it via a special name.

7.4 Garbage Collection in Ficus and Rumor

Ficus and Rumor provide the *no lost updates* semantics, using a two-phase, coordinator-free, distributed solution [Guy *et al.* 1993, Ratner *et al.* 1996a] that is essentially a distributed implementation of the two-phase commit protocol [Gray 1978, Lampson *et al.* 1979]. The algorithm verifies that no participant completes before all participants are knowledgeable that completion is imminent. Any site with a locally inaccessible object initiates garbage collection on that object; the algorithm either stops when a new name is discovered, or consensus is reached that the object is globally inaccessible and garbage collection completes. Algorithm details can be found in [Guy *et al.* 1993] and [Ratner *et al.* 1996a].

The algorithm performs adequately in Ficus and Rumor; however, its semantics often cause disks to become full of garbage waiting to be collected, since data is preserved at all replicas until algorithm completion. Even in a mostly connected environment, developers were forced to write special tools to remove the data and generate free disk space independent of the garbage collection algorithm. Additionally, Ficus and Rumor users often truncated large files prior to deleting them, specifically to avoid potential garbage collection problems on those objects later.

Ficus and Rumor provided the *no lost updates* semantics because it was originally thought to be cheap to maintain the information, and the designers believed that information should not be discarded without global confirmation of an explicit user action. However, as experience with the two systems demonstrated, maintaining the extra information was not cheap, and mobility only makes it more expensive.

Mobility greatly exacerbates the full-disk problem, due to three observations:

- Disconnections are more common, more frequent, and typically longer in duration in a mobile environment.
- Mobility will most likely increase replication factors (see Chapter 2), increasing the time required for garbage collection to complete at all replicas.
- Mobile users have limited options available to them when they are isolated and their disk becomes full. In general, they cannot contact other users or other replicas to decide the "correct" action to remedy the situation.

[2] The author was one of the main architects of Ficus and Rumor, and has logged several man-years of experience with both. Initially, the Roam source code was replicated with Rumor, and the garbage collection problem constantly caused the laptop's disk to become full.

From these observations, it seems clear that we cannot maintain data at each replica while garbage collection executes. Garbage collection will execute much more slowly, given the larger number of replicas and the more frequent and longer network disconnections. Additionally, mobile users will default to truncating data before removing it or utilizing expert tools such as those described above to deal with the full disk problem. Both options nullify any advantage gained from the powerful *no lost updates* semantics, and in some cases can cause errant and incorrect behavior. Mobile computing requires a different algorithm with different semantics.

7.5 A New Algorithm

We have developed a new garbage collection algorithm that allows the immediate reclamation of data once the last *local* name for an object has been removed, and maintains "markers" or records of the object to resolve the create/delete ambiguity. Even with the aggressive removal of data, we still reliably detect all remove/update conflicts and present understandable and defensible semantics to the user. We will first describe the semantics, and then provide an overview of the algorithm. Algorithm analysis can be found in Sections 7.6 and 7.7. Algorithm details and pseudo code are provided at the end of this chapter (Section 7.9).

7.5.1 New semantics

The algorithm provides what we call *time moves forward* semantics, or simply TMF. Similar to Goel's work with optimistic consistency models [Goel 1996], we guarantee that time as seen by file names always moves forward. Each name for an object always refers to data versions that increase with time. That is, each name never refers to a data version i at time t_1 and then later refers to a data version $i-e$ at time $t_2 \geq t_1$. In this context, the *no lost updates* semantics could be said to provide a time moves forward guarantee on all names, present and future, for a given object. We omit the dependency between different names for the same physical object, and provide the guarantee on each name independently. We believe *time moves forward* to be a defensible, understandable, and potentially more correct position, as explained by the following algorithm analysis and discussion.

In short, we discovered that with the *time moves forward* semantics, we could guarantee that updates would never be lost without having to pay the expense of the *no lost updates* semantics.

7.5.2 Algorithm overview

Whenever a given replica has no local names for an object, it removes file data. A record of the action is preserved for create/delete disambiguation, and we execute a two-phase consensus algorithm to remove the record. Since the record itself is a small number of bytes, and the file data is reclaimed immediately, the disk space and other resources used by the algorithm are practically unnoticeable to the user.

During synchronization, when each new replica `R` learns that its last local name has been removed, `R` checks its data version against the version that was initially removed. If `R`'s data version is more recent, `R` preserves the data as a remove/update conflict and notifies the user; otherwise `R` immediately reclaims the disk space for the file and participates in the two-phase consensus algorithm on the object record. The disk space occupied by the file is therefore reclaimed as quickly as possible. Table 7.2 illustrates a simple example.

7.5.3 Algorithm description

Upon the deletion of the last local name for an object, the replica physically removes the file data, preserves a record of the object for create/delete disambiguation, saves the version vector at the time of removal (known as the saved-vector), and initiates a distributed consensus-forming algorithm regarding global inaccessibility.

When a given replica `R` learns of the name deletion via reconciliation with replica `S`, it also removes the name. If the name was `R`'s last local name as well, `R` compares its data version (i.e., version vector) with

Replica R	Replica S	Time
remove `name1` and reclaim disk space	—	t_1
—	recon: learn of the name removal and reclaim disk space	t_2

Table 7.2: A simple example illustrating that the new garbage collection algorithm reclaims disk space as quickly as possible. Replica `S` cannot reclaim disk space until it learns of the name removal, and as soon as that occurs, disk space is reclaimed at `S`. Assume that the file has only the single name **`name1`**, and that both `R` and `S` originally had the same data version. Time in the table flows downward.

Replica R	Replica S	Time
update `name1`	—	t_1
—	remove `name1` and reclaim disk space	t_2
recon: learn of the name removal, but notice that the local version dominates, so preserve the local data in the orphanage	—	t_3
—	recon: get orphanage entry and data	t_4

Table 7.3: A simple example illustrating the detection of a remove/update conflict and the preservation of data. `S` removes `name1` and reclaims the disk space, but `R` has independently updated it. `R` therefore preserves the file data, when it discovers the remove/update conflict at time t_3. `S` discovers the remove/update conflict at time t_4. Time in the table flows downward.

`S`'s saved-vector. If `S`'s saved-vector dominates or is equivalent to `R`'s version vector, `R` removes its file data, preserves a record of the object, and participates in the consensus-forming algorithm. If `R`'s version vector dominates or is in conflict with the saved-vector, a *potential* remove/update conflict has been detected. The updated data version is preserved for the user in a special place known as the *orphanage*—a special directory for files with no names.[3] Table 7.3 illustrates the detection of a remove/update conflict and the preservation of data.

The remove/update conflict detected in the above manner is only a potential conflict. Information regarding the global accessibility or inaccessibility of the object is required to determine if the conflict is real or simply a facet of the algorithm. Remove/update conflicts caused by interactions with dynamic naming are called *pseudo*, because they are not true remove/update conflicts—the object was never globally inaccessible.

However, dynamic naming rarely occurs [Floyd 1986a], meaning that the percentage of pseudo conflicts is expected to be low. Additionally, the inherent value in detecting a remove/update conflict is in recovering the data for the user instead of requiring it to be reconstructed manually. As such, the value in reporting a remove/update conflict decreases with time—typically the problem will have either been ignored or already solved if the user is not informed quickly enough. We therefore believe it is important, even critical, and potentially more correct to report potential remove/update conflicts to the user the instant they are detected, rather than waiting for global confirmation.

Even when multiple names for the object exist globally, and the conflict is pseudo rather than real, the pseudo remove/update conflict is still important to the user. For instance, consider what happens when

[3] The use and name of the orphanage is adopted from Ficus.

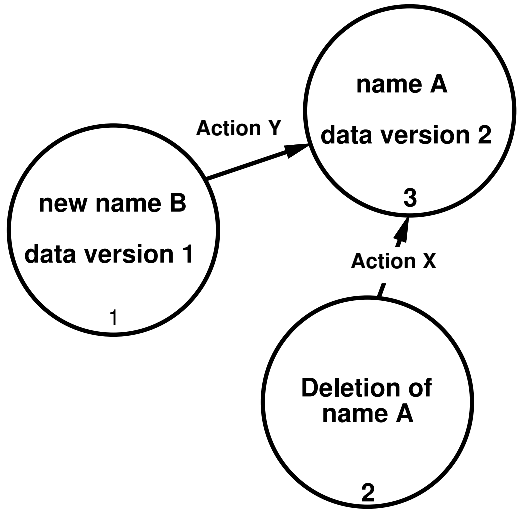

Figure 7.1: An example showing data loss during a pseudo remove/update conflict. Replicas are indicated by circles with their replica identifier, and arrows indicate the propagation of new information. Data version 2 is strictly newer than data version 1. If synchronization action `X` occurs before action `Y`, replica 3 will detect a remove/update conflict before removing the most recent version of the data. If action `Y` occurs before action `X`, no remove/update conflict exists. In both cases, data version 2 is preserved.

a user at replica `R` removes the last local name for a file, and assume that `R` stores the most recent data version. If there are no names globally, the data should be reclaimed; however, if a new name exists at some replica `T`, then the file is technically globally accessible, and therefore should not be reclaimed. The Ficus *no lost updates* semantics guarantee that the latest data stored at `R` would eventually be paired with the new name known at `T`. However, we argue that the expense of the operation—preserving the data everywhere until it is globally determined that no names exist—is overly excessive in a mobile or large scale environment. Furthermore, given that the user at replica `R` *knew* what data version he or she was removing, the correctness of the *no lost updates* semantics is arguable, as discussed in Section 7.7. The TMF semantics therefore make a compromise. We make a "best effort" attempt to ensure that the latest data version is paired with the globally active name or names, but at the same time we preserve the ability to reclaim the data at each replica as quickly as possible. Therefore, we cannot guarantee that such a pairing will always occur: due to the inherent randomness of network disconnections and reconciliation intervals, we cannot deterministically guarantee a specific ordering between propagation of the new name and propagation of the removal of the old name. If the latter occurs before the former, the most recent data version will be removed. In this case, saving the data and reporting the behavior as a remove/update conflict is beneficial to the user, although the actual conflict is caused more by the behavior of the algorithm than user behavior.

For example, see the illustration in Figure 7.1. If replica 3 synchronizes with replica 2 before synchronizing with replica 1 (i.e., action `X` occurs before action `Y`), replica 3 detects a remove/update conflict. The conflict is actually pseudo, since a new name exists at replica 1, but replica 3 does not yet know about the new name. If action `Y` occurs before `X`, replica 3 will not detect a remove/update conflict. In other words, we cannot guarantee that, given all combinations of reconciliation patterns and network partitions, data version 2 will eventually be attached to name `B`. However, in the cases where it does not occur, we guarantee that a remove/update conflict will be detected and data version 2 will be preserved. Therefore, the user never loses data under any scenario.

It should be noted that in the case of *remove/remove* conflicts (multiple replicas simultaneously remove the same name), the saved-vector is taken to be the larger of the two values. Most remove/remove conflicts are not actual conflicts, and the data is collected normally. In the case of conflicting saved-vectors, both are preserved, and any given replica's version vector must dominate or conflict with both to qualify as a potential remove/update conflict.

The replica that physically hosted the user's deletion removes the underlying file data as soon as the user's action is detected (i.e., the next time the file system is scanned). The other replicas remove file data during synchronization. When replica `R` synchronizes with replica `S` and encounters an object for which both replicas have no local names, it performs the preceding check for a potential remove/update conflict. If none is detected, `R` immediately removes its file data. The disk space occupied by the object is therefore freed as quickly as possible—it cannot, by definition, be removed at any replica prior to that replica learning of the name removal, and it is removed as soon as the last local name removal occurs. In this sense, the algorithm is optimal.

Each replica participates in a consensus-forming, distributed two-phase algorithm to determine global inaccessibility and remove the object record, which is small in size compared to the object's data. The two-phase algorithm for record removal operates in parallel with the propagation of the name removal and occurs during synchronization. The two phases guarantee that each replica knows both that everyone else is participating in garbage collection (phase 1) and that everyone else knows that it itself is participating (phase 2) [Guy *et al.* 1993]. Once any replica has completed the two-phase protocol, it may remove the record itself, completing the garbage collection process. If the object is not globally inaccessible, the two-phase algorithm will detect the new name and terminate. In this case, each replica preserves the object record and eventually learns of the new name and data for the object.

Guy has previously demonstrated that we require two phases to correctly remove the record and resolve the create/delete ambiguity [Guy *et al.* 1993].

7.5.4 Garbage collection and wards

As described in Chapter 4, garbage collection operates semi-independently between the wards, with the ward masters managing the cooperation. That is, garbage collection occurs on a ward-by-ward basis. Each ward completes when all of its members are ready to complete. The ward master participates in the garbage collection of the object as part of two wards, although each ward executes the algorithm independently.

7.5.5 Algorithm assumptions

The algorithm assumes that creation of a new name for an object requires communicating with a data-storing replica for the object itself. Otherwise a new name could be created while all object replicas remove their data. Fortunately, this is not an unreasonable assumption. Many systems, such as Ficus, Rumor, and standard Unix, require such a condition.

Additionally, the algorithm makes minimal assumptions about the available connectivity between all replicas. We require that all data replicas are *gossip connected* such that information from any one replica eventually reaches every other replica, although direct interconnections may not exist and all replicas may not be simultaneously available. More formally, let the set of data replicas be the set Φ. Φ is gossip connected if for all pairs of replicas $R, S \in \Phi$ there exists a finite series of replicas $X_1, X_2, \ldots, X_n$ such that a direct path exists:

- at time t_0 from R to X_1
- at time t_k from X_k to X_{k+1} for $1 \leq k < n$ and $t_0 < t_k < t_{k+1}$
- at time t_n from X_n to S, $t_{n-1} < t_n$

Guy calls this set of conditions *time connectivity* [Guy *et al.* 1993].

7.6 Algorithm Analysis

Assume there are `N` replicas of a specific file, labeled R_1, R_2, R_3,...,R_N. With no lack of generality, let replica R_1 delete its last local link to the file; the file becomes locally inaccessible at R_1, and it removes data and initiates garbage collection. Under these conditions, either replica R_1 stores the most recent version of the data, some other replica R_j stores the most recent version, or there are conflicting versions. ("Stores" the most recent data version includes the results of any concurrent file system activity that may occur during the garbage collection process.) In each of these cases, there are two sub-cases to consider: either only one name exists for the object, or multiple names exist. This yields six cases total to consider; each will be analyzed in turn.

7.6.1 Replica R_1 stores most recent version

Replica R_1 stores the most recent version of the data. There are two sub-cases to consider: all replicas know the file only by a single name (the one which replica R_1 just deleted), or some replica or replicas have dynamically created new names or will create new names before they learn that the original name has been removed.

Only one name

All replicas identify the file by a single name, which replica R_1 just deleted. All replicas will eventually learn that replica R_1 deleted the only name for the file, and the file will become globally inaccessible. Since replica R_1 stored the most recent version of the data, and it performed the remove, there is no remove/update conflict. Therefore the file can be completely garbage collected.

Multiple names

Some replica R_i has created or will create a new name for the object. All replicas will eventually learn that replica R_1 deleted its only name for the file, but since at least R_i has additional active names, the file is not globally inaccessible. Therefore, all replicas will eventually have the file accessible by the new name(s). The resulting data version of the file can be any version between R_i's version and R_1's version, depending on the particular reconciliation pattern. For example, suppose there is a replica R_d with a data version greater than (or equivalent to) R_i's, but less than R_1's. If R_d learns of the removal of the name before it learns about the new name(s) generated at R_i, R_d will detect a (pseudo) remove/update conflict, and preserve its data specially for that purpose. However, if R_d learns of the new name(s) before it learns of R_1's removal, then its data version will be preserved.

Thus, we cannot definitively say which version of the file data will be accessible by the new name. However, we guarantee that time always moves forward regarding the data version seen by the new name(s). The data version eventually stored at all replicas under the new name is guaranteed to be no older than the one stored at replica R_i, the replica with the new name.

7.6.2 Replica R_j stores most recent version

Replica R_j, $j \neq 1$, stores the most recent version of the data. As above, there are two sub-cases to consider: all replicas know the file only by a single name (the one which replica R_1 just deleted), or some replica or replicas have dynamically created new names or will create new names before the garbage collection algorithm reaches them.

Only one name

All replicas identify the file by a single name, which replica R_1 just deleted. All replicas will eventually learn that replica R_1 deleted the only name for the file, and the file will become globally inaccessible. When a replica that stores a data version greater than R_1's learns of the removal, it will notice that its data version dominates R_1's, and therefore a potential remove/update conflict has occurred. All such replicas will save their versions of the file data. As all replicas perform the same action and garbage collection progresses, eventually the most recent version of the data will be saved, and will be preserved for the user as a result of the remove/update conflict. The "potential" remove/update conflict in this case is a true remove/update conflict.

Multiple names

Some replica R_i has created or will create a new name for the object. All replicas will eventually learn that replica R_1 deleted its only name for the file, but since at least R_i has additional active names, the file is not globally inaccessible. If a replica R_d that stores more recent data than R_1 learns of the name removal before it learns of the other active names for the file, it will detect a pseudo remove/update conflict. Alternatively, if R_d learns of the other active names generated by R_i before learning of R_1's name removal, it will know there is no remove/update conflict. Whether or not R_d detects a remove/update conflict depends on the particular reconciliation patterns. If R_d detects a conflict, it is not a true remove/update conflict in the global sense, but it is certainly a remove/update conflict in the local sense between R_1 (the remover) and R_d (the updater). Reporting it as such is beneficial to the user, and arguably even more correct than the FICUS *no lost updates* semantics which would transparently attach the R_d's data version to the new names generated at R_i. Although the data version is preserved, it is preserved under a new name, possibly unknown to the user at R_1. It seems more beneficial to detect the *local* remove/update conflict and identify it as such.

7.6.3 Conflicting versions exist

Replica R_1 and R_{c1}, R_{c2},...,R_{cm} have conflicting data versions when R_1 removes its last name for the file. Replica R_1 removes its data, essentially "undoing" its part of the conflict. When another replica R_c detects that its version conflicts with the version previously stored at replica R_1, there isn't anything R_c can do, because only part of the conflict (i.e. only R_c's data version) is still available.

As above, there are two sub-cases to consider: all replicas know the file only by a single name (the one which replica R_1 just deleted), or some replica or replicas have dynamically created new names or will create new names before the garbage collection algorithm reaches them.

Only one name

All replicas identify the file by a single name, which replica R_1 just deleted. All replicas will eventually learn that replica R_1 deleted the only name for the file, and the file will become globally inaccessible. Since the underlying object is still in conflict, the algorithm must save at least one of the versions, and more correctly $m-1$ versions. This seems understandable, since the user physically removed the version at R_1, and knew what version was being removed; it is the other $m-1$ conflicting versions that may have been unknown at the time of file removal. Since R_1 removed its data version, we preserve the other $m-1$ conflicting versions that weren't removed, and identify them as remove/update conflicts.

Multiple names

Some replica R_i has created or will create a new name for the object. All replicas will eventually learn that replica R_1 deleted its only name for the file, but since at least R_i has additional active names, the file is not globally inaccessible. The initial m-way conflict will be a $(m-1)$-way conflict under the new name(s). During the garbage collection process, before learning about the new names generated by replica R_i, some of the conflicting replicas might learn about R_1's removal, and therefore report possible remove/update conflicts. In this case the conflicts would be pseudo, and the conflict would essentially be reported twice:

once by the remove/update mechanism, and once by the update/update mechanism (since the object itself is already in update/update conflict).

The removal of the name is treated as a type of resolution of that portion of the conflict—that is, a removal of that portion of the conflict.

7.7 Discussion

It would seem that in each of the six cases, the algorithm behaves in a manner that is both defensible from a correctness point of view and understandable from the user's point of view. Though some may argue that in certain circumstances the *no lost updates* semantics behave more correctly, these semantics cannot deallocate disk space without two full rounds of communication—a delay that is unacceptable in a mobile environment. The *time moves forward* semantics guarantee that the object's data, which constitutes the vast majority of the disk space, is freed as quickly as possible. The semantics therefore seem well-suited, if not optimal, for mobile computing.

In some cases the semantics report what we call pseudo or *local* remove/update conflicts, as distinguished from true or *global* remove/update conflicts. Local remove/update conflicts occur between a pair of sites, rather than between all replicas: although the file is still globally accessible via some name, replica R_1 removed a name that caused more recent data to be removed at replica R_2. We believe this to be acceptable behavior. We cannot predict the patterns of reconciliation, and therefore cannot know whether the name removal or the new name addition will propagate to replica R_2 first. Therefore, in either case we preserve the more recent version stored at replica R_2.

More importantly, detecting and reporting the local remove/update conflict is arguably more correct than the Ficus *no lost updates* semantics, which would transparently attach the data version to the new name. Although the data version is preserved, it is preserved under a new name, possibly unknown to the user at R_2. It seems more beneficial to detect the *local* remove/update conflict and identify it as such, rather than force the user to find the new name.

7.8 Proof of Correctness

The above analysis (Sections 7.6 and 7.7) describes the algorithm's behavior. To prove correctness, we must demonstrate that one of two invariants becomes true in finite time:

1. All replicas have removed or will remove their object record exactly once, thus completely "forgetting" about the object.
2. Alternatively, all replicas have or will learn about a new name for the object.

Both of these invariants are properties of the two-phase algorithm for reclamation of the object record, not the fast reclamation of object data. Guy already demonstrated that the two-phase algorithm is correct under the above conditions in [Guy 1991]; his proof applies here equally well.

7.9 Algorithm Details

We outline the new garbage collection algorithm in pseudo code. First we define the various data structures used, and then present the algorithm.

7.9.1 Algorithm data structures

The following is a list of the data structures used by the algorithm when garbage collecting a single object, many of which are adopted from Guy [Guy 1991]. The only replicated data structure among the list is the saved-vector; the remainder are maintained independently at each replica, though each replica gossips about their values to the other replicas.

saved-vector: a preserved copy of the version vector at the time the last name is removed.

tlink[ ]: a vector, one element for each replica, indicating the total number of names ever seen by that replica. Each element is monotonically increasing, as the element reflects a cumulative total.

reference count C: the number of current names for the object at a given replica.

remove/update bit RU: *RU* is true if and only if a potential remove/update conflict has been detected.

status: The status is an enumerated type, which can take on one of three values: *live, dead, destroyed.* Live objects have active names locally ($C > 0$). Dead objects have no local names ($C == 0$) but still maintain data locally. Destroyed objects are dead objects with the data already reclaimed.

phase1[ ], phase2[ ]: Two bit-vectors, one element for each replica, used for create/delete disambiguation. Each element indicates local knowledge regarding the corresponding replica's garbage collection progress.

7.9.2 Algorithm details

When replica R removes its last local link for the object:

// first destroy data
set saved-vector = current version vector
remove file data
set status = DESTROYED

// now start two-phase algorithm
$\forall$ replicas S, set phase1[S] = phase2[S] = 0
set phase1[R] = 1
set tlink[R] = 1

When replica R_1 reconciles from replica R_2:

(1) If, after processing all name additions and removals from R_2's information,
R_1 removes its last local name for a given object, then R_1 performs the following:
// can't remove data yet; must check for remove/update conflict in (2) below
set status = DEAD
set reference count $C = 0$

// start two-phase algorithm
$\forall$ replicas S, set phase1[S] = phase2[S] = 0
set phase1[R_1] = 1
set tlink[R_1] =1

(2) After step (1), or anytime R_1 and R_2 both have the same object with
reference count $C == 0$, R_1 performs the following:
if status == DEAD
    // check for remove/update conflict
    if R_1's version vector > R_2's saved-vector
        // potential remove/update conflict detected

```
            save data in orphanage
            set RU = true
            perform remove/update conflict notification
        else if R1's version vector conflicts with R2's saved-vector
            // remove/update conflict
            save data in orphanage
            set RU = true
            perform remove/update conflict notification
        else
            // no remove/update conflict that R1 is involved in, so remove data
            set saved-vector = R2's saved-vector
            remove file data
            set status = DESTROYED

            // participate in two-phase algorithm
            ∀ replicas S, set tlink[S] = max(tlink[S], R2's tlink[S])
            set phase1[] = bitwise-or phase1[] with R2's phase1[]
            if ∀ replicas S, phase1[S] == 1
                set phase2[R1] = 1 // start phase2
                set phase2[] = bitwise-or phase2[] with R2's phase2[]
                if ∀ replicas S, phase2[S] == 1
                    // complete garbage collection
                    remove all remaining data structures
    else if status = DESTROYED
        // no need to check for remove/update conflicts any more
        // gossip about the largest saved-vector seen
        set saved-vector = max(R1's saved-vector, R2's saved-vector)

        // verify that R2 didn't temporarily learn of new names
        if tlink[R2] ≠ 0 and tlink[R2] ≠ R2's tlink[R2]
            // R2 learned about new names; restart consensus algorithm
            ∀ replicas S, set phase1[S] = phase2[S] = 0
            set phase1[R1] = 1
            set phase1[] = bitwise-or phase1[] with R2's phase1[]
            ∀ replicas S, set tlink[S] = max(tlink[S], R2's tlink[S])
        else
            // two-phase algorithm
            set phase1[] = bitwise-or phase1[] with R2's phase1[]
            if ∀ replicas S, phase1[S] == 1
                set phase2[R1] = 1 // start phase2
                set phase2[] = bitwise-or phase2[] with R2's phase2[]
                if ∀ replicas S, phase2[S] == 1
                    // complete garbage collection
                    remove all remaining data structures
```

(3) If, in processing objects from R_2's information, R_2 does not store an object R_1 expects it to store, R_1 performs the following:

```
    if status != LIVE and phase1[R2] = 1
        // we know R2 started garbage collection, so now it must have
        // completed; therefore, we can complete too
        remove all remaining data structures
```

(4) If, in processing name additions from R_2's information, R_1 creates a new name for an object in either the DEAD or DESTROYED state,

R_1 performs the following:

set status = LIVE
set C = 1
$\forall$ replicas S, set phase1[S] = phase2[S] = 0
set tlink[R_1] = tlink[R_1] + 1
if status = DESTROYED
    set version vector = all zero's // lets get any copy of data

Chapter 8

Version Vector Maintenance

Version vectors [Parker *et al.* 1983] are the mechanism used by ROAM to track updates and compare data versions. To recap from Chapter 3, the version vector is an array of length equal to the number of replicas. Each replica R maintains its own vector independently, and tracks in position i the number of updates generated by replica S_i that are known at R. In other words, each element is a monotonically increasing counter, and each counter in position i tracks the total number of known updates generated by replica S_i. Since each replica independently maintains its own version vector, each could temporarily have different values for S_i's position, reflecting the fact that some replicas have more recent data than others. Eventually, when all replicas are consistent, all replicas will have equivalent version vectors. Table 3.1 illustrates a simple example.

Version vectors are well known and proven correct [Parker *et al.* 1983]. However, they do not scale well in large systems, given that each replica maintains its own, dedicated position in the vector. Various researchers have commented on the size of the version vector as a primary scaling problem in FICUS, and a simulation of RUMOR demonstrated the scaling problems concretely [Wang *et al.* 1997]. The data structures simply become too large as the number of replicas increases.

In an effort to remedy the scaling problem, we noted two key observations:

1. Updates typically occur in a few isolated "hot-spots," as opposed to concurrently at all replicas.

2. Once all replicas have the same element for replica R, replica R's element is no longer relevant for version vector comparison purposes, and provides no additional information for distinguishing between data versions.

From these two observations, we realized that *dynamic* maintenance of the version vector would dramatically increase its scalability. By dynamic we mean:

1. Rather than pre-allocating a vector element for each replica R, we can instead dynamically *expand* the vector and create R's element when R generates its first update. Zero-valued elements in the vector are insignificant in the comparison routines, and can therefore be trivially removed.

2. Once all replicas have the same value for replica R_j in their vector, R_j's position can be removed from each vector with no loss of distinguishing information. Should replica R_j generate future updates, it can re-expand the vector via dynamic expansion. The challenge however, and one of the main contributions of the dynamic maintenance algorithms, is to provide dynamic version vector *compression* on R_j's element while still allowing R_j to generate updates during the compression process. We cannot, even temporarily, block R_j from generating updates while executing a distributed algorithm to remove its element, because we nullify the whole advantage and purpose of optimistic replication.

We believe that dynamic version vector expansion and compression can dramatically increase the scalability of the version vector while adding minimal cost to the system. As such, it is an important and key

Vector 1	Vector 2	Dominance Relation
{{1,1}, {2,5}}	{{1,1}, {2,4}}	V_1 dominates V_2
{{1,1}, {2,5}, {3,0}}	{{1,1}, {2,4}}	V_1 dominates V_2
{{1,1}, {2,5}}	{{1,1}, {2,4}, {3,0}}	V_1 dominates V_2
{{1,1}, {2,5}, {3,0}}	{{1,1}, {2,4}, {4,0}}	V_1 dominates V_2

Table 8.1: Adding zero-valued elements in the version vector adds no new information and does not change the existing dominance relation. Version vectors are indicated as tuples of {replica identifier, value}. Note that the dominance relation doesn't change, although zero-valued elements are added in both vectors.

distributed algorithm at the core of ROAM. This chapter first describes version vector expansion and then discusses compression.

8.1 Dynamic Vector Expansion

Dynamic version vector expansion allows one to store only non-zero elements in the vector, and provides for the dynamic expansion of the vector when a replica without an existing element generates an update. We first describe why dynamic expansion is important and then describe the changes to the data structure and the mechanism itself.

8.1.1 Motivation

Zero elements in the version vector add no distinguishing information. Given two version vectors V_1 and V_2 and a dominance relation `D` between them, such as V_1 dominates V_2, adding unmatched zero elements (elements in one vector with no corresponding element at the other) to either vector leaves `D` unchanged.[1] Table 8.1 illustrates an example.

Additionally, our experience with replicated file systems like FICUS and RUMOR seems to indicate that some replicas *never* generate updates. It is important, especially as we increase scale, that the non-writers do not adversely affect the size of the version vector. The version vector is, after all, meant to distinguish the series of updates; it makes sense, therefore, to only mention in the vector the replicas that have generated updates.

If we could predict beforehand the set of writing replicas, we could simply make the remainder read-only, and reduce the size of the version vector permanently. However, we cannot, in general, predict the set of writers, and it is important to give each replica the *ability* to generate updates, regardless of how often updates occur, as described in Chapter 2. Therefore, we must preserve the ability for each replica to generate updates while taking advantage of the fact that many replicas will generate no updates.

Sparse-matrix approaches [George *et al.* 1981] could potentially be used to compress the zero elements. However, the non-zero elements could be anywhere in the vector. The writing replicas are not guaranteed to be in one contiguous list, minimizing the benefit of sparse matrix compression techniques. Additionally, the set of updating replicas is typically different for different objects, meaning that sorting the vector, if applied, would have to be done on a per-object basis and constantly re-applied, as more replicas generated updates. While potentially beneficial, the approach seems overly complex.

Instead, the approach we take is to never store zero elements, and to dynamically *expand* the vector when a replica generates its first update. As a result, replicas that never generate updates are never mentioned in the version vector, although they reserve the ability to generate updates at any time. An example is illustrated in Table 8.2.

[1] The topic of version vector dominance is discussed in Chapter 3.

Action	Resulting Version Vector
—	{}
Replica 1 updates	{{1,1}}
Replica 1 updates	{{1,2}}
Replica 7 updates	{{1,2}, {7,1}}
Replica 1 updates	{{1,3}, {7,1}}
Replica 4 updates	{{1,3}, {7,1}, {4,1}}

Table 8.2: An example of dynamic version vector expansion. Initially the object is created with an empty version vector. When each replica updates, it increments its position or creates its position if it doesn't exist. Version vectors are indicated as tuples of {replica identifier, value}.

8.1.2 Changes to the vector definition

Dynamic version vector expansion requires minor changes to the basic definition of a version vector element. A basic version vector is simply a list of counters, one counter per replica. Since new replicas typically join the system optimistically by communicating with only one other replica, version vectors will not always be the same length; however, identifying which element belongs to which replica is vital. The order of replicas in the list can be pre-defined to match a specific ordering, such as specified by a master list or a standard sorting algorithm; the replica identifiers themselves need not be present in the vector. A given replica's position can be identified using the pre-defined mechanism.

However, once we remove the zero elements and institute dynamic vector expansion, the ordering in the vector becomes indeterminate. Master lists cannot be utilized because the set of replicas mentioned in the version vector may be different for each object. Sorting the replicas does not improve the situation, because position i no longer uniquely specifies a specific replica identifier, as the replica identifier corresponding to position i potentially changes whenever a new replica is inserted. Adequately solving the problem requires an associative array indexed by replica identifier.

Using an associative array increases the size of each element by the size of the key. Of course, we are also shortening the version vector by removing zero elements. Since a ROAM replica identifier is 32 bytes long and a version vector element (counter) is 32 bytes long, the break-even point with regard to saving space is half the elements: if omitting the zero elements and using an associative array removes over half the elements, we actually save space.

However, the real benefit from dynamic expansion occurs when used in conjunction with dynamic vector compression, described below in Section 8.2. When both dynamic compression and expansion are used together, the version vectors tend to list only the replicas that are actively generating updates. An object without any active replicas could, in fact, have a completely empty, or null, version vector. Therefore, while the space overhead of the associative array may seem excessive when considered by itself, dynamic version vector management as a whole has the ability to dramatically decrease the size of the version vector.

8.1.3 Implementation

The version vector is implemented in ROAM using an STL[2] `map`, which is an associative array. The vector is initially created with no elements. Each time replica `R` generates an update, `R` attempts to locate its position. If its position exists, `R` increments the value stored there; otherwise `R` expands the vector by creating a new element with the value of one, and adds the new element to the associative array with its replica identifier as the key.

[2] STL, or the Standard Template Library, is a common library package used with C++.

Vector 1	**Vector 2**	**Dominance Relation**
{{1,1}, {2,5}, {3,2}}	{{1,1}, {2,4}, {3,2}}	V_1 dominates V_2
{{1,1}, {2,5}}	{{1,1}, {2,4}}	V_1 dominates V_2

Table 8.3: Removing equivalent elements from all version vectors leaves the dominance relation unchanged. In this case, replica 3's element (value 2) is removed from all vectors, leaving the dominance relation the same. Version vectors are indicated as tuples of {replica identifier, value}.

8.2 Dynamic Vector Compression

Studies of replicated file systems [Kumar *et al.* 1993, Reiher *et al.* 1994] illustrate that conflict rates are generally quite low, meaning that concurrent writing of the same object (within the synchronization time-window) rarely occurs. Experience with replication systems like FICUS and RUMOR seems to expand upon this notion. Objects tend to have "hot-spots" or small collections of writers within the set of replicas. While the hot-spots may change over time, it is rare to see a widely replicated object that is consistently updated by everyone. File system analyses and simulation data seems to support the hot-spot notion.

Analyses of file usage in existing systems suggest that both concurrent read and write sharing is quite rare [Floyd 1986b]. In the referenced examination of an academic UNIX system, of all files read during a seven-day study period, under 7% were read by more than one user during the next week (and the vast majority of the files read by multiple users were news articles and hence read-only; only 1.3% of files owned by normal users were read by multiple readers). Update sharing was even rarer. Just 2.4% of all files written (and less than 1% of user-owned files) were updated by multiple writers. Similar results were found for directories.

Analyses of commercial systems suggests similar conclusions. The re-analysis of trace data collected in the 1970s [Smith 1981] from three commercial IBM timesharing environments found that from 3% to 12% of the files accessed are updated by multiple users [Kure 1988]. A more recent study looked at traces taken in three different commercial environments. In studies of productivity environment data taken at Locus Computing [Kuenning *et al.* 1994], Wang found that replicated objects tend to have only a small set of common writers at any one time [Wang *et al.* 1997].

In short, while we must provide the *ability* for everyone to generate updates, it seems rare that a significant percentage of the replicas are trying to do so at the same time—a hypothesis we call the *hot-spot hypothesis*. The hot-spot hypothesis provides intuition into the expected update patterns, and illustrates that the elements for the *cold* replicas, the ones not in the current hot-spot, play only a minor role in the version vector. When a cold replica generates an update or even a burst of updates, its element quickly stabilizes at all replicas since, by definition, cold replicas only rarely generate updates. Once the element stabilizes it no longer plays an important or distinguishing role in the version vector comparison algorithm, and can be removed everywhere with no loss of versioning information. Table 8.3 illustrates an example.

When removing the cold replica's element, only the actual number of updates generated by the cold replica is lost; we do not lose any versioning information with respect to the object itself. The physical number of updates is of dubious value anyway, given that the element is only required to be a monotonically increasing counter, and need not represent the actual number of updates. Multiple file system updates can be coalesced into one element increment, provided that each distinct object version which is locally generated and seen by any other remote replica is tagged with its own unique version vector value. Both RUMOR and ROAM, for instance, use update coalescing to generate version vector elements, since the systems are not in the critical path of normal file system operations and therefore cannot distinguish a single update from a series of updates. More information regarding how RUMOR (and ROAM) formulate version vectors can be found in [Reiher *et al.* 1996, Salomone 1998].

Note that the *hot* versus *cold* distinction is on a per-object, per-replica basis. A given replica will typically be in the hot-spot for some objects and not for others.

The situation is therefore ripe for dynamic version vector compression. We periodically execute a distributed algorithm to remove the elements generated by the cold replicas. Only the cold replicas are good candidates for compression, since the hot replicas are expected to generate more updates "soon." Of course, during the element-removal process we still must retain the high availability of optimistic replication systems. Therefore, we must allow any replica, including the one whose element is being removed, to still generate updates. Retaining the ability for any replica to generate updates during compression proves to be the biggest challenge.

We will first describe the algorithm requirements, challenges, and a brief overview. Then we will describe the changes to the version vector definition, discuss the algorithm assumptions and our model of the system, and describe the details of the algorithm itself. We conclude with a series of remarks regarding the algorithm, including a method to generalize the algorithm by removing the key assumption and the best times to initiate compression.

8.2.1 Algorithm requirements

First and foremost, the algorithm must be safe. For any two version vectors V_1 and V_2 and a dominance relation `D` between them, if the comparison of V_1 and V_2 yields `D` before compression, then barring other updates the comparison must also yield `D` after compression.

Additionally, we must not prevent updates by any replica, hot or cold, during the algorithm's execution. Even while we are attempting to remove replica `R`'s element, we must allow updates by `R` to maintain the high availability provided by optimistic strategies. In the event that `R` generates an update, the compression algorithm may have no effect (in that it doesn't remove `R`'s element), but it must not cause incorrect behavior.

8.2.2 Algorithm challenges and simplifications

There are several distributed algorithms for achieving consensus on a given value [Dwork *et al.* 1988, Lynch 1996, Fischer 1983, Chandra *et al.* 1991, Lamport 1989, Oki *et al.* 1988, Skeen 1981], but dynamic version vector compression poses some additional challenges while also allowing specific simplifications.

There are two key algorithmic challenges. First, communication must be restricted between version vectors V_1 and V_2 when the former one has removed an element and the latter has not; incorrect behavior such as false conflict detection can otherwise occur. Often the communication restriction may simply be a mechanism for letting V_2 know that it must complete compression prior to vector comparison. Nevertheless, such a communication restriction must occur.

Second, compression occurs in place, meaning that the value of the element being compressed may change during or even after consensus has been reached. In contrast, garbage collection (Chapter 7) forms consensus on a globally stable and unchanging condition.

Additionally, we allow one important simplification. Our compression algorithm ignores the issue of failed machines, because the underlying Ward Model detects and propagates the identity of failed machines (discussed in Chapter 4). Our algorithm simply accesses the failed-machine information that is already stored at each replica.

8.2.3 Algorithm overview

The algorithm works on a per-element basis, although multiple elements can be removed at once. For each element being removed, the algorithm achieves consensus on a value to be subtracted from the existing element. After consensus is reached, each replica subtracts the common value from the specified element. If the resulting element is zero, the entire element is removed from the vector. The consensus algorithm ensures that no replica with element x ever attempts to subtract a value greater than x.

We tag each version vector (not each element) with a sequence number that counts the number of times compression has occurred, to ensure that vectors V_1 and V_2 do not accidently compare elements e_1 and e_2 when one of them has been compressed and the other has not. The in-place issue is resolved by associating a "spare" position with each element being compressed, allowing consensus to be achieved independently of new updates.

8.2.4 Data structure modifications

There are three substantial changes to the version vector structure. The first is the addition of a "spare" element. Since it is only used during compression, it does not generally increase the size of the vector. The second is a one-phase bit vector used to detect the correct completion of the compression algorithm. The third is a sequence number attached to the version vector to prevent communication between vectors when one has compressed and the other has not. Each will be discussed in turn. Note that the bit vector is described as if each individual element has its own. The algorithm description will later expand upon this basic definition and illustrate that multiple instances of the algorithm can be easily coalesced, so that the entire version vector requires only one bit vector.

The spare element

A typical version vector element e is simply a monotonically increasing value $\{a\}$.[3] Dynamic vector compression additionally requires that while compression on a given element is occurring, the element take the form $\{a, b\}$, where b is referred to as the "spare" element. When compression starts, elements of the type $\{x\}$ are changed to $\{0, x\}$. Consensus is achieved using the "b-value" (the value in the b position for the element) while future updates increment the a-value; in this way we allow updates during the compression process. The b-value can be dynamically allocated, so it occupies no additional space except when compression is occurring.

Since we changed the basic format of a version vector element, we must modify the compare function to account for the b-value. The basic approach used is to sum a and b, and use that value as the element's total for comparison purposes. The sum $a + b$ accurately reflects the current version, because the element had the value b when compression started, and since that time has received a more updates. Specifically, there are three cases to look at when comparing two elements e_1 and e_2 (assuming that the version vector sequence numbers match, for otherwise we cannot compare the two vectors):

1. Both e_1 and e_2 are of the form $\{a\}$.

   In this case neither element is undergoing compression. We compare $e_1 = a_1$ and $e_2 = a_2$ by comparing a_1 and a_2.

2. Exactly one of e_1 and e_2 is undergoing compression.

   Without loss of generality, assume that e_1 is the element undergoing compression. Then $e_1 = \{a_1, b_1\}$ and $e_2 = a_2$, and we compare $a_1 + b_1$ against a_2.

3. Both e_1 and e_2 are undergoing compression.

   In this case, $e_1 = \{a_1, b_1\}$ and $e_2 = \{a_2, b_2\}$. We compare $a_1 + b_1$ against $a_2 + b_2$.

The one-phase bit vector

We cannot complete compression until all replicas have achieved consensus on the b-value. We use a one-phase bit vector to determine when consensus has occurred. Each participant indicates knowledge and agreement by setting its corresponding bit in the bit vector. In general, one phase is not enough to determine consensus [Guy *et al.* 1993]. However, in our case we only require one phase. Unlike the scenario with general consensus problems, the replica whose element is being compressed, called the *generating replica*, cannot, by design, change its "vote" (i.e. b-value) once it is set. As described above, new updates simply increment the a-value. The purpose of the bit vector, therefore, is to guarantee that everyone else eventually

[3] Dynamic vector expansion (Section 8.1 above) adds a replica identifier to the element, but we ignore the replica identifier for now.

learns of the correct b-value. Since we cannot be assured that each replica will directly communicate with the generating-replica, we must ensure that the compression information survives long enough to be gossiped to each replica. The rules for setting bits in the bit vector are described more fully in Section 8.2.7, but basically bits are not set unless the communicating replicas agree on the value of b.

The sequence number

To block comparisons between vectors that have undergone different numbers of compression "cycles," we add a sequence number to the version vector. The sequence number counts the number of times compression has completed on any element in the vector. Comparisons can only occur between vectors with equivalent sequence numbers, although different sequence numbers may still yield important information. For instance, if vector V_1 has completed compression and V_2 has not, (V_2 waiting on communication and participation from V_1), then V_2 can conclude from the sequence-number differential that V_1 has finished compression and therefore it can also finish.

8.2.5 Algorithm assumptions

We make the initial assumption that only replica `R` can initiate compression on its element, except when replica `R` no longer exists. We refer to the replica that "owns" a particular element as the *generating* replica, since it is the replica that generates updates for that element. Under this assumption, two replicas will either agree on the b-value (the consensus value) or else one of the b-values will be zero (that replica doesn't know compression has begun), thus simplifying algorithm presentation. In Section 8.2.11 we remove this assumption, which leads to situations where $b_1 \neq b_2$ and neither b_1 nor b_2 is zero.

Additionally, we assume that Byzantine behavior does not occur. Long-term communication delays may occur, but replicas themselves are truthful in their responses.

8.2.6 System model

We assume that replicas are allowed to move between wards or more generally hierarchical groups without requiring special "pre-motion" actions. If ward motion required communicating with all affected ward masters, the version vector could perhaps be maintained in a ward-specific manner, with the ward masters maintaining consistency between the wards. However, requiring pre-motion actions greatly reduces the usability of the system, as described in Chapter 2. We therefore assume that pre-motion actions are not required, and that any replica can move between wards at any time.

We make the same minimal assumptions regarding the available connectivity between replicas as in the garbage collection algorithm (Chapter 7). Although long-term communication delays may occur and all replicas may not be simultaneously available, we require that all replicas be *gossip connected.*

8.2.7 Algorithm details

We present the algorithm in four stages. First we describe the algorithm as operating on a single version vector element in a non-hierarchical (i.e. non-ward) model. Second, we add a hierarchical model without considering motion between groups in the hierarchy. We then add support for motion between groups, and conclude by generalizing the approach to compress multiple elements at once.

Basic algorithm

We will first describe the compression algorithm, and then apply it to two different examples at the conclusion of the description. The basic compression algorithm operates on a single version vector element in a non-hierarchical (non-ward) model. When replica `R` initiates compression on its element, the element changes form from $\{x\}$ to $\{0, x\}$. Additionally, replica `R` sets its bit in the one-phase bit vector.

The compression information is now gossiped from replica to replica according to the following rules. Assume that replica R_1 pulls information from R_2 as part of synchronization. The version vectors V_1 from R_1 and V_2 from R_2 are compared, looking at matching elements e_1 and e_2 from each vector, respectively. Assume that replica R_2 knows that compression has begun, and therefore e_2 has the form $\{a_2, b_2\}$; R_1 may or may not be knowledgeable, and therefore e_1 either has the form a_1 or $\{a_1, b_1\}$. If R_2 does not know compression has started, R_1 takes no action with regard to the compression algorithm. Under these assumptions, the rules are:

1. If e_1 has the form a_1, R_1 changes e_1 to have the form $\{0, a_1\}$. R_1 then applies the rules below.
2. R_1 can set its own bit in its bit vector whenever it knows that compression has begun (i.e., e_1 has the form $\{a_1, b_1\}$).
3. As part of synchronization, if V_2 dominates V_1 and new data is installed at R_1, then V_1 is set to be V_2. Specifically, this copies the b-value b_2 when $b_2 > b_1$. (The a-value is copied as well). R_1 then applies rule 4.
4. If $b_1 = b_2$, then R_1 merges R_2's bit vector with its own using the bit-wise "or" function.
5. When all bits are set in the bit vector, R_1 can complete compression on element e_1 in vector V_1.

These rules guarantee that when any replica has all bits set in the bit vector, all replicas will reach or have reached consensus on the b-value. The completing replica changes the element $e = \{a, b\}$ by subtracting b from the total value for the element $a + b$. If $(a + b) - b = a = 0$, then the entire element is removed; otherwise e simply becomes a.

Since R is the only replica that increments its element in the version vector, no replica S can ever have a larger element for R than replica R. Additionally, no replica S can ever have a non-zero b-value without learning that compression has begun and therefore having communicated (directly or indirectly) with replica R. Therefore, once a replica S learns that compression has begun on replica R's element, S must have the highest b-value in the system, guaranteed by rules 1-3, assuming they are applied atomically. If two version vectors are in conflict and have different b-values, then by rule 4 the bit vector will not be completely set until the conflict is resolved (unless the conflict only involves the a-values, in which case we can achieve consensus independent of conflict resolution). Therefore we can be assured that all replicas have achieved consensus on the same b-value.

Each replica, when it completes compression, increments its sequence number on the associated version vector by one. When replica R_1 reconciles with replica R_2, and, for a given version vector, V_2's sequence number is greater than V_1's, R_1 knows that R_2 has completed compression, and therefore R_1 can complete compression as well on V_1.

Example 1: A simple application of the basic algorithm can be seen in Table 8.4. Initially, at time t_0, replicas 1 and 2 have the same version vector, while replica 3 has an older version. Compression begins at time t_1. Replica 2 decides to compress its element, which changes at replica 2 from 3 to {0,3}. It additionally creates its bit vector and sets its own bit. At time t_2, replica 1 reconciles from replica 2. Replica 1 learns that compression has begun and changes its form for replica 2's element from 3 to {0,3}. Replica 1 creates its bit vector, sets its own bit, and additionally sets replica 2's bit, since replica 1 and replica 2 have the same b-value. Replica 3 then reconciles from replica 1 at time t_3. It changes its element for replica 2 from 2 to {0,2}, creates its bit vector, and sets its own bit. Replica 3 then notices that replica 2's version vector dominates its own, so it pulls over the new file data and the new version vector at time t_4. Since replica 3's b-value now equals replica 2's, replica 3 additionally sets replica 2's bit in its bit vector. At time t_5 replica 1 reconciles from replica 3 and, since replica 1 and 3 agree on the b-value, replica 1 sets replica 3's bit. Replica 1 now has a completely set bit vector, so it completes compression at time t_6 and increments its cycle number. The a-value for replica 2 is zero, so replica 2's element is completely removed. At time t_7 and t_8, replicas 2 and 3 similarly reconcile with replica 1 (3 could reconcile with either 1 or 2) and complete compression, because they are a cycle behind.

Time	Replica 1	Replica 2	Replica 3
t_0	{{1,2}, {2,3}, {3,1}}:0	{{1,2}, {2,3}, {3,1}}:0	{{1,2}, {2,2}, {3,1}}:0
t_1		{{1,2}, {2,**{0,3}**}, {3,1}}:0 bit vector:{0, **1**, 0}	
t_2	{{1,2}, {2,**{0,3}**}, {3,1}}:0 bit vector:{**1**, **1**, 0}		
t_3			{{1,2}, {2,**{0,2}**}, {3,1}}:0 bit vector:{0, 0, **1**}
t_4			{{1,2}, {2,**{0,3}**}, {3,1}}:0 bit vector:{0, **1**, **1**}
t_5	{{1,2}, {2,{0,3}}, {3,1}}:0 bit vector:{1, 1, **1**}		
t_6	**compress element 2** {{1,2}, {3,1}}:**1**		
t_7		**compress element 2** {{1,2}, {3,1}}:**1**	
t_8			**compress element 2** {{1,2}, {3,1}}:**1**

Table 8.4: A simple example of version vector compression. For each important action, the version vector and bit vector (if applicable) are indicated for each replica. The cycle number is indicated as a separate number set apart from the version vector by a colon. The example is explained in depth in Section 8.2.7 under Example 1.

Time	Replica 1	Replica 2	Replica 3
t_0	{{1,2}, {2,3}, {3,1}}:0	{{1,2}, {2,3}, {3,1}}:0	{{1,2}, {2,2}, {3,1}}:0
t_1		{{1,2}, {2,**{0,3}**}, {3,1}}:0 bit vector:{0, **1**, 0}	
t_2	{{1,2}, {2,**{0,3}**}, {3,1}}:0 bit vector:{**1**, **1**, 0}		
t_3			{{1,2}, {2,**{0,2}**}, {3,1}}:0 bit vector:{0, 0, **1**}
t_4			{{1,2}, {2,**{0,3}**}, {3,1}}:0 bit vector:{0, **1**, **1**}
t_5		**2 updates** {{1,2}, {2,{**1**,3}}, {3,1}}:0 bit vector:{0, 1, 0}	
t_6	{{1,2}, {2,{0,3}}, {3,1}}:0 bit vector:{1, 1, **1**}		
t_7	**compress element 2** {{1,2}, {3,1}}:**1**		
t_8		**compress element 2** {{1,2}, {2,1}, {3,1}}:**1**	
t_9			**compress element 2** {{1,2}, {2,1}, {3,1}}:**1**
t_{10}	{{1,2}, {2,1}, {3,1}}:**1**		

Table 8.5: A more complicated example of version vector compression. For each important action, the version vector and bit vector (if applicable) are indicated for each replica. The cycle number is indicated as a separate number set apart from the version vector by a colon. The example is explained in depth in Section 8.2.7 under Example 2.

Example 2: Example 2 begins just as Example 1 does, except that at time t_5, in the middle of the consensus algorithm, replica 2 generates an update. The update gets applied in the a-value of 2's element. At time t_6, when replica 1 reconciles from replica 3, replica 1 has all bits set in its bit vector and completes compression, even though it is not knowledgeable of replica 2's update. Replica 1 removes replica 2's element completely. At time t_8 replica 2 reconciles from replica 1, notices the difference in cycle numbers, and realizes it too can complete compression. However, replica 2 has a non-zero a-value, so when it completes compression it replaces {1,3} with just the value 1. Replica 3 similarly does the same when it reconciles with 2 and realizes that compression can complete. Replica 3 also obtains 2's update during the process. Replica 1 adds replica 2's element back (and obtains the accompanying data version) when it reconciles at time t_{10}.

Hierarchical model

We add to the basic algorithm by allowing a hierarchical replication model. We assume a fully-generalizable, N-level hierarchy. At each hierarchical level, replicas are organized into wards, each of which has a ward master. Ward masters are organized into higher-level wards, which themselves have higher-level masters. The hierarchy continues upward until eventually, at the top of the hierarchy, is a group with no masters.

The only change from the basic algorithm concerns the form of the bit vector. Instead of a single bit vector, each element being compressed has M hierarchical bit vectors, depending on how many hierarchical levels the particular replica belongs to. For instance, a normal ward member still only has one bit vector, but a replica that belongs to wards in M hierarchical levels would have M hierarchical bit vectors. In a two-level implementation, all replicas have a single bit vector except ward masters, which maintain two bit vectors. The hierarchical bit vectors are ranked lowest to highest corresponding to the hierarchical model itself, with the "top" of the hierarchy being the highest hierarchical level.

We define the lowest-order bit vector that is not completely set as the *active* bit vector, and define the hierarchical level corresponding to the active bit vector as the active hierarchical level. The preceding rules 2, 4 and 5 from above change to (changed conditions in bold):

2. $\mathtt{R}_1$ can set its own bit in its **active** bit vector whenever it knows that compression has begun (i.e., e_1 has the form $\{a_1, b_1\}$).

4. If $b_1 = b_2$ **and replica $\mathtt{R}_2$ is a member of the active hierarchical level**, then $\mathtt{R}_1$ merges $\mathtt{R}_2$'s **hierarchical** bit vector with its own using the bit-wise "or" function.

5. When all bits are set in the bit vector **and $\mathtt{R}_1$'s ward master, if one exists, has compressed**, $\mathtt{R}_1$ can complete compression on element e_1 in vector V_1.

The algorithm operates correctly because the first replica $\mathtt{R}$ to complete is a replica at the highest level of the hierarchy, as guaranteed by rule 5. Since bit vectors are set in ascending hierarchical order, the only way for $\mathtt{R}$ to complete is for all hierarchical levels beneath it to be ready to complete. Thus everyone knows about compression, and it is safe to compress. As in the basic algorithm, when replica $\mathtt{R}_1$ reconciles with replica $\mathtt{R}_2$, and, for a given version vector, V_2's sequence number is greater than V_1's, then $\mathtt{R}_1$ knows that $\mathtt{R}_2$ has completed compression, and therefore $\mathtt{R}_1$ can complete compression as well on V_1.

Motion between groups

We add to the hierarchical algorithm support for mobility, and allow replicas to move between wards. We support two types of motion operations, as described in Chapter 4: ward overlap and change.

The only change from the above hierarchical algorithm deals with maintenance of the hierarchical bit vectors. If replica $\mathtt{R}$ performs a ward overlap operation, nothing special need occur, since replica $\mathtt{R}$ is still a member of its original ward, and therefore still under the "control" of its original ward master. $\mathtt{R}$'s original ward master still communicates on $\mathtt{R}$'s behalf, meaning that $\mathtt{R}$'s original ward master will not set its bit in its higher-level bit vector until $\mathtt{R}$ is knowledgeable that compression is occurring and stores the same b-value.

If replica `R` performs a ward change, the bit vectors of the new ward expand to incorporate the new member `R`, and the bit vectors of the old ward shrink to remove `R`. For `R` to perform the move it must reconcile with some member of the new ward. Therefore, if `R` is not knowledgeable that compression has begun but the new group is knowledgeable, then `R` learns about compression upon entry to the new group, obtaining a b-value consistent with at least one other replica in the group. Since any replica that is knowledgeable that compression has started must have the largest b-value in the system, correctness is ensured.

If `R` is already knowledgeable that compression has begun, it simply carries this knowledge into the new group. Again, since any replica that is knowledgeable that compression has started must have the largest b-value in the system, correctness is still ensured.

Simultaneous compression of multiple elements

The algorithm as described so far works on a single element, requiring each element to have its own set of bit vectors. Since we could potentially have multiple instantiations of the algorithm simultaneously running, each on a different element, it seems to make more sense to merge the multiple instantiations into one.

We have two methods of enabling simultaneous compression of multiple elements. The first would be simply to maintain a bit vector per element. Since each element maintains its own bit vector, we can easily separate the progress of the various algorithms, and we can compress each element independently. Even though the bit vectors are dynamically allocated at compression time, maintaining multiple bit vectors might prove expensive.

The alternative method modifies the basic algorithm by adding an additional phase. Version vector compression becomes a two-phase algorithm; the first phase establishes consensus on the set of replicas whose elements are being compressed, and the second phase ensures consensus on the b-values. As is often the case in distributed algorithms, the tradeoff between the two approaches is one of space versus time. If it is rare that multiple elements will be compressed simultaneously, than the former solution is clearly better. However, supporting the concurrent compression of multiple elements in a general manner is most efficiently (in space) performed with the latter solution.

For simplicity, we have chosen the latter (two-phase) solution. We modify the bit vector to have two bits per element instead of one. The first phase guarantees consensus on the set of replicas being compressed; if, during the algorithm, some replica `R` learns that additional elements are being compressed, `R` restarts the algorithm on the merged set of elements, clears all its bit vectors, sets its own bit according to rule 2 above, and continues. The two-phase nature guarantees that when any replica completes, all replicas will eventually complete on the same set of elements.

Termination is guaranteed only if we can guarantee that the first phase (consensus forming on the set of elements being compressed) will complete. We institute a termination guarantee by restricting the number of elements that can be compressed at once to some finite number n: the algorithm can only restart at most $n-1$ times. We therefore place an absolute upper bound on the first phase of the algorithm, and can guarantee that the overall algorithm will eventually finish.

8.2.8 Proof of correctness

To prove the algorithm correct, we must show that two conditions hold. First, all replicas must eventually agree on the same b-value. Second, all replicas must eventually complete compression.

We can prove that all replicas will agree on the same b-value, because only the generating replica, the replica that "owns" a given position, initiates compression on its element. Since the generating replica `R` is the only replica that directly increments the value in `R`'s element, no replica `S` can ever have a larger value for `R`'s position than replica `R`. Furthermore, once `R` initiates compression, further updates by `R` only increment the a-value; the b-value is set upon the initiation of compression. It follows, therefore, that any replica `T` that is knowledgeable of compression on `R`'s element must have the same b-value as `R`. `T` cannot have a larger value, as that is expressly impossible as described above. Additionally, `T` cannot have a smaller b-value unless it is zero, in which case `T` does not even know that compression has begun. Assuming `T` knows that compression has started, it must have communicated either with R or some intermediary S, such that S has communicated directly with S_1, S_1 has directly communicated with S_2, and some S_n has directly communicated with R. When T learns that compression has begun, it therefore knows exactly R's version at the time R initiated compression, and therefore has the same b-value.

We can similarly prove that all replicas eventually begin compression. Since we assume that all replicas are *gossip connected*, as specified in Chapter 3, it is possible for each replica to learn in finite time that compression has begun. Since each replica periodically synchronizes, incorporating remote information into its own, and since all replicas are guaranteed to be available infinitely often, information propagates from one replica to all replicas in finite time.

Each replica will learn that compression has begun, and each replica, upon learning about compression, sets its own bit in its bit vector. Furthermore, we have already demonstrated above that all replicas have the same b-value. Thus, when replica R synchronizes with S, R will merge its bit vector with S's, causing the propagation of all bits S has set to R's bit vector. After synchronization, R's bit vector reflects the combination of R's and S's information. Therefore, within finite time some replica T will have every replica's bit set in its bit vector; T then completes compression and increments its cycle number. As knowledge of the new cycle number propagates throughout the system, all other replicas complete compression as well. All replicas are gossip connected, and information generated at one replica is guaranteed to eventually flow to all other replicas; therefore the new cycle number, indicating the end of compression, will propagate to all replicas, and all replicas will complete compression.

8.2.9 Sequence number management

Since all file replicas participate in the algorithm, no replica can ever be more than one compression cycle behind any other replica. Therefore, instead of a full sequence number, it suffices to have a tri-state sequence number, which is smaller in size and additionally can never overflow.

8.2.10 Communication restriction

As mentioned above, communication needs to be restricted between replicas R_1 and R_2 when their version vectors V_1 and V_2 have completed a different number of compression cycles. However, communication (i.e., synchronization) is never actually restricted. All replicas participate in the algorithm, and no replica R_1 can ever be more than one compression cycle behind any other replica R_2. In such a situation, R_1 has enough information to complete compression; communication is then permitted with R_2. Communication is therefore never actually restricted between replicas; rather, the sequence number is only used as an indication that the restricted replica should complete compression before performing synchronization.

8.2.11 Removing the key assumption

The simplifying assumption in the preceding algorithm is that only replica R can initiate compression on its element. (R in this case is the generating-replica.) The assumption makes practical sense, given that replica R is the best authority concerning when compression should begin on its element. There is little advantage in compressing an element that is being updated, since its value will only be reduced, as opposed to removing the entire element. Replica R is the only replica that knows when its update-generation behavior has ceased, so it is the only authority regarding when to initiate compression on its element. Additionally, the algorithm restriction allows us to make strong claims regarding any replica S's b-value for R's element; specifically that any replica S either has a zero b-value or R's b-value.

However, for full generality we may desire an algorithm without such a restriction: that is, an algorithm that allows any replica S to initiate compression on any replica R's element. Clearly, the fully general algorithm is more complex. The main difficulty arises in forming consensus on the b-value. Replica S, for instance, may store the value n_s for R's element when it decides to initiate compression on R's element, while replica `R` stores the value $n_r > n_s$.[4] Correctness dictates that consensus must be globally formed on either n_r or n_s (or some value in-between).

We can relatively easily form consensus on n_s by modifying the rule governing how elements begin compression. Instead of changing the element from x to $\{0, x\}$, to form consensus on n_s we must change the element to:

$$\begin{array}{ll} \{0, x\} & \text{if } x \le n_s \\ \{x - n_s, n_s\} & \text{if } x > n_s \end{array}$$

We can guarantee that consensus will correctly be formed on n_s using the same argument as above: consensus is formed on the b-value stored by the initiating replica (`S`) at the time compression is started. However, forming consensus on n_s is only mildly interesting, because if $n_r > n_s$ then the element being compressed is not completely removed, but only reduced in ordinal value.

It is more effective to form consensus on n_r, although also more difficult. To do so, we essentially "restart" the algorithm when necessary. Whenever replica $\texttt{R}_1$ reconciles from $\texttt{R}_2$ and discovers that $b_2 > b_1$, replica $\texttt{R}_1$ updates its version vector to reflect the latest b-value (and the accompanying data), resets all bit vectors, and sets its own bit according to rule 2. The bit vectors represent a set of sites that store a common b-value; when the b-value changes, we must reset the bit vectors. Since the bit vectors are set in lowest to highest hierarchical order, and the first replica to complete is at the top of the hierarchy, we are guaranteed that when any replica completes, all replicas will have the same b-value.

A given b-value stabilizes when its generating-replica learns that compression has begun. Therefore, in the basic algorithm, when only the generating-replica can initiate compression on its element, the b-value is stable upon initiation of compression, and the bit vector only serves to guarantee propagation of the information to all parties. With the more general, advanced algorithm, the b-value may change during the course of the algorithm. Essentially, information about compression propagates from the algorithm initiator to the generating-replica and then out to all replicas. The general algorithm therefore simply adds an extra "phase": distribution of the request for compression to the generating-replica. In a standard optimistic approach, we assume that the generating-replica will have the same b-value, and we therefore set bits in the bit vector during this initial "phase." If the generating-replica has the same b-value, then our optimism provided substantial benefit, as many replicas already have the bits correctly set; otherwise, the replicas simply reset their bit vectors. The optimistic approach during the first "phase" generates no extra work in the case where our optimism was unjustified, and when it was justified provided substantial savings in time. Our optimism is based on the fact that the element being compressed corresponds to a cold replica, and therefore it is not expected to receive further updates.

8.2.12 When to initiate compression

Since compression requires communicating with all data-storing replicas of the object, i.e., all replicas that maintain a version vector, and compression of multiple elements requires restarting the algorithm, there can be a significant penalty when compression is initiated at an "inopportune" moment. Ideally, compression on `R`'s element should not be initiated until `R` has entered a quiescent (cold) state, and its update activity has at least temporarily ceased. Compression of hot replicas should be avoided. Of course, differentiating between hot and cold replicas is not always easy; neither is it easy to determine when a burst of updates has stopped. Heuristics to determine these events are an ongoing area of future work.

[4] The different values are due to the effects of optimistic replication and time delays involved with periodic synchronization.

Chapter 9

Performance Analysis

In the previous chapters we have described the ROAM architecture, the various algorithms and controls, and the implementation structure. However, a useful system must not only be designed well: it must also perform well. If the cost of use outweighs the benefits the system provides, users will generally avoid using the system in favor of more cost-efficient mechanisms, even if they involve manual control. We must therefore characterize the performance of the system and, where possible, justify that ROAM does not impose an unacceptable burden upon its users.

Our performance analysis looks at several factors. First, we measured the disk space overhead. ROAM requires additional disk space for its data structures and mechanisms, but from the user's perspective, anything other than the actual user files is overhead. Second, we measured system performance during synchronization in a variety of environments, and demonstrated the scalability of the model with a hybrid-simulation approach. Finally, we measured system performance during ward motion. We conclude with some comments on real use.

9.1 Disk Space Overhead

ROAM, like RUMOR before it, stores its non-volatile data structures in lookaside databases within the volume but hidden from the user. From the user's perspective, anything other than his or her actual data is overhead and effectively shrinks the size of the disk. Minimal disk overhead is therefore an important and visible criterion for user satisfaction.

Additionally, ROAM is designed to be a scalable system. The Ward Model should support hundreds of replicas with minimal impact between wards. Specifically, the creation of a new replica in ward `X` should not affect the disk overhead of the replicas in other wards.

We therefore measured the disk overhead of ROAM using two different volumes, and compared it to RUMOR's overhead, since RUMOR is the only other user-level peer replication package available. We also measured the disk overhead caused by new ward creation. We then analytically generated equations to describe the overhead as the number of files, types of files, number of replicas, and number of wards change. Finally, we looked at the effect of selective replication on the disk overhead. Each case will be described separately.

Overhead is measured in Linux using `du`, which measures size in terms of 1KB disk blocks. An empty directory occupies one 1KB disk block, as does a very small file with only a few bytes of information.

9.1.1 Volume one

We first measured a "typical" user data volume, consisting of 307 files, of which 38 are directories, and totaling 13.6MB of data. Some directories have only a few files; the root directory (the volume root) has 151. The largest file is 1.4MB, and some files are only a few bytes long.

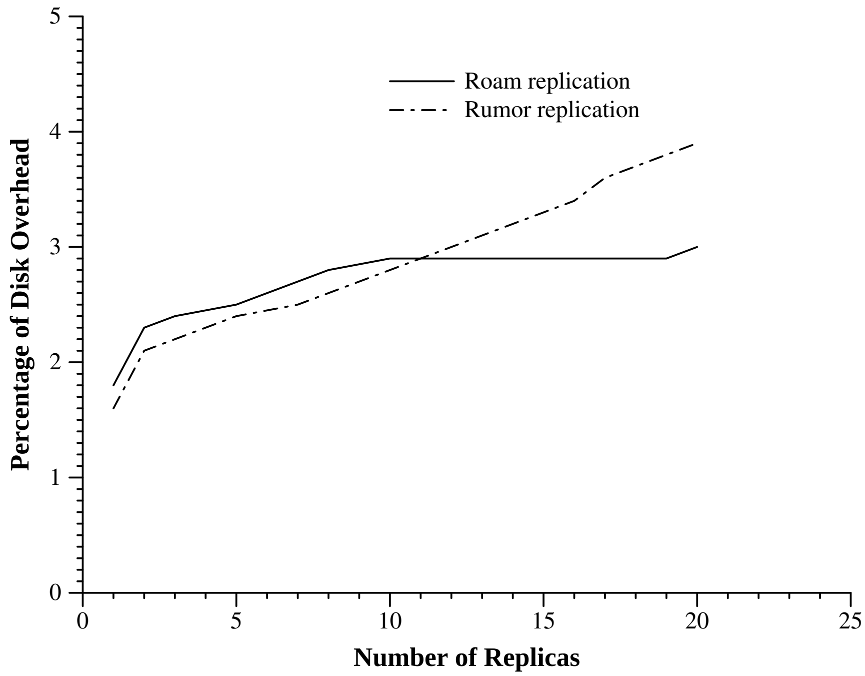

Figure 9.1: Roam and Rumor disk overhead for volume one. Disk overhead is measured as a percentage of the size of the user's data, which is 13.6MB. To demonstrate the effect of wards, new wards are created in Roam's case at replicas 10 and 20. Replicas 1-9 belong to ward one, replicas 10-19 belong to ward two, and replica 20 belongs to ward three.

Figure 9.1 shows the disk overhead as a percentage of the total user data, as the number of replicas increases. Except in the case of a single replica, the overhead is measured at replica 2. The figure indicates that users pay approximately a 2% overhead when creating the first replica, and the overhead increases linearly with the number of replicas at less than a .1% increase per additional replica. The large jump between the first and second replicas is due to a design flaw; when we correct the flaw, the curves will keep the same slope but will cross through the initial point corresponding to replica 1, rather than rising sharply at replica 2. That is, each data point for replicas 2 through 20 will shift downward by approximately .3 for both Rumor and Roam.

The overhead increases linearly as the number of replicas increases. Roam, however, is designed to be a scalable system; wards are meant to isolate the effects of increased scale. To verify Roam's scalability, we create a new ward at replica 10. Replicas 1-9 belong to the first ward, and replicas 10-19 belong to the second ward. Since the new replicas belong to the second ward, the overhead at replica 2 (or any other replica in the first ward) does not increase with the additional replicas. The overhead increases again at replica 20 because replica 20 belongs to a third ward, and each replica pays a small per-ward increase for system reliability. Figure 9.2 shows the per-ward overhead more clearly, and Section 9.1.2 discusses the topic in depth.

Figure 9.1 demonstrates one aspect of the scalability of Roam. While the overhead in Rumor increases linearly, the separation of replicas into wards reduces the linear increase in disk overhead to zero overhead. The addition of replicas 11 through 19 add zero overhead to that measured at a replica in the first ward, because replicas 11 through 19 belong to the second ward.

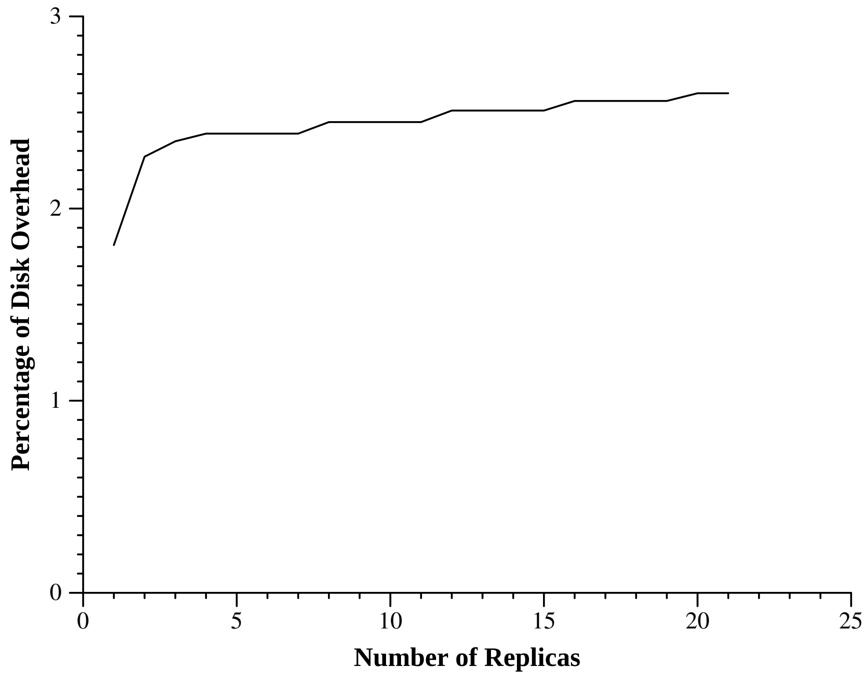

Figure 9.2: The effect of new ward creation on disk overhead. The volume is the same one as in Figure 9.1. We create a new ward every fourth replica. That is, replicas 1-3 belong to ward one, replicas 4-7 to ward two, etc., to a total of 21 replicas in 6 wards.

9.1.2 Per-ward increase

As mentioned above, the addition of each new ward causes a small increase in each replica's disk overhead. Recall from Chapter 4 that during ward master re-election, the newly elected master must "re-connect" itself with the other ward masters to properly join the higher-level ward. To do so, each replica must locally store the identities of the other ward masters, which adds a small amount of overhead as each new ward (and therefore new ward master) is created. Figure 9.2 illustrates the additional overhead. The data set is the same as in Section 9.1.1. There are 307 files, of which 38 are directories, totaling 13.6MB. Every fourth replica marks the creation of a new ward: new wards are formed at replicas 4, 8, 12, 16, and 20. Again, the sharp increase between replicas 1 and 2 is due to a design flaw and is not a real effect. Overhead is measured at a replica in the first ward (replica 2).

As illustrated in the graph, we experience a slight increase in overhead when each new ward is created, and no additional increase for the new members of that ward. The increase caused by new ward creation is approximately .06%. Since the increase is so small, we conclude that Roam can easily tolerate large numbers of wards, although clearly not an infinite number. For example, having 100 wards only adds a 6% overhead at each replica. Note that the per-ward increase is substantially smaller than the additional replica increase (.06% versus .3% from Figure 9.1) that occurs when a new replica is created within the same ward. Roam's scalability therefore depends on the creation of multiple wards, so that no single ward ever becomes too large or contains the majority of the replicas.

9.1.3 Volume two

We also measured the overhead for a second type of volume. We used the `/usr/include` tree from our Linux Slackware distribution, which consists of 374 files, of which 21 are directories, totaling 1.6MB of data.

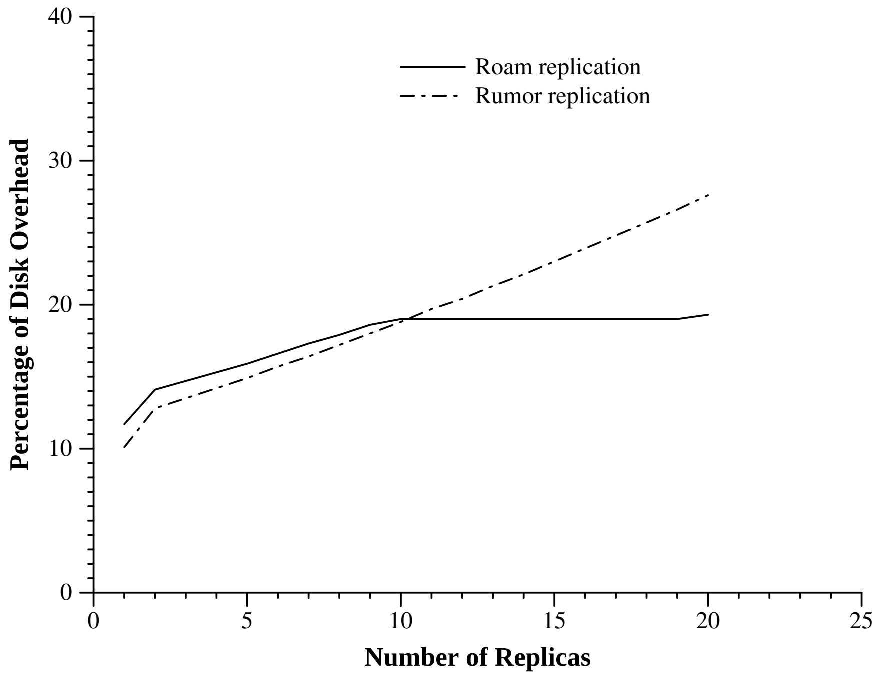

Figure 9.3: ROAM and RUMOR disk overhead for `/usr/include`. Disk overhead is measured as a percentage of the size of the user's data, which is 1.6MB. To demonstrate the effect of wards, new wards are created in ROAM's case at replicas 10 and 20. Replicas 1-9 belong to ward one, replicas 10-19 belong to ward two, and replica 20 belongs to ward three.

As before, we expect a sharp increase between replicas 1 and 2, and we would likewise expect the curve to increase linearly with increasing numbers of replicas until the formation of a new ward.

The overhead is illustrated in Figure 9.3. The overhead starts at roughly 10 to 11% for one replica and increases linearly. Again, we detect the same trends: the increase in overhead becomes zero when a new ward is created at replica 10. We also create a new ward at replica 20.

We experience more relative overhead here than with the previous volume because the average file size is smaller. The number of objects in the two volumes is roughly comparable but the overall size of the second volume is more than ten times smaller. In general, the size of the lookaside databases depends on the number of files, not the size of the files, so the disk overhead should be inversely proportional to the average file size in the volume. From the graph, we see almost 19% overhead at 10 replicas, which unfortunately can be quite visible to the average user. We conclude from this observation that ROAM should not generally be used to replicate large volumes with very small average file sizes, because the overhead becomes noticeable.

Figure 9.3 also illustrates the effects of dynamic version-vector expansion (as does Figure 9.1, although the effect is not as visible due to scale). The slope for ROAM's curve between replicas 1 and 9 (before creating the second ward) is slightly less than that for RUMOR. The smaller slope is due to dynamic version vector expansion in ROAM. Since replicas 2 through 20 have not generated updates, they occupy no space in the version vector, and therefore the overall data structure is comparatively smaller as the number of replicas increases. RUMOR, on the other hand, pays the overhead for storing a zero (and associated replica information) in each version vector for each new replica. With ROAM, if each replica generates an update and we ignore the effects of dynamic version vector compression, ROAM's curve would have the same slope as the RUMOR curve between replicas 1 and 9.

9.1.4 Overhead equations

Of course, we cannot measure the overhead for every possible scenario. We therefore developed a set of equations to characterize the disk overhead. We validated the equations against the above volume scenarios and others, with less than 9% error.

Roam's disk overhead can be characterized by the following formulas:

- Each new directory costs 4.2KB + 30 bytes per object in the directory
- Each new file costs .24KB
- The first replica, even without any user data, costs 57.36KB
- Each additional replica costs 6.44KB + 12 bytes per object stored at the replica
- Each new ward costs 6.44KB

We derived the formulas analytically and then validated them against a number of test cases. We created an empty Roam volume and measured the disk overhead in 1KB blocks at 58KB, matching precisely the analytic result of 57.3KB, since the file system must round up. We created a second replica of the empty volume and measured the increase in overhead at 6KB. The analytic result predicts 6.44KB, which differs from the measured result by 7.3%.

Using the formulas, we also calculated our overhead prediction for the second replica of `/usr/include`. The formulas predict an overhead of 231.9KB; in measurements, we see 254KB. The difference is 8.7%.

The minor differences between the analytic and measured results are due largely to variable-length data structures. The lookaside databases are stored in ASCII to make debugging easier. However, ASCII generates variable length storage. For example, the number "12" requires two bytes in ASCII, while the number "1024" requires four. Roam's data structures store a variety of numbers (for instance, the globally unique file identifier and the inode number), which, when stored in ASCII, occupy variable space. Other variable-length fields must also be stored, such as file names. Names for objects are inherently variable-length; we assume a 10-character name, but names can be essentially arbitrary length, limited only by the native file system.

9.1.5 Overhead analysis and optimizations

The numbers suggest the general trend that overhead increases linearly with the number of files, directories, new replicas, and new wards. Additionally, they demonstrate that the addition of new replicas in one ward does not affect the overhead of the replicas in the remaining wards.

The overhead could be optimized and made smaller in a number of ways. First, the directory cost could relatively easily be reduced to 3.2KB + 30 bytes per object simply by fixing the aforementioned design flaw. Second, an analysis of the overhead indicates that it is roughly split between two major sources. First are the two ASCII lookaside databases that Roam (and Rumor) maintain. The two databases, the *filelist* and the *inode-ffsh-map*, contain the attributes of the replicated objects and the mapping of inode numbers to unique file identifiers, respectively. Combined, they account for approximately half of the overhead. We could compress them when they are not needed, trading off CPU time for disk space. Simple experiments with `gzip` indicate that the databases compress quite well; we typically experience a 7- to 9-fold reduction in size. Alternatively, we could store them in binary, which does not use variable space and is in general more compact than ASCII.

In general, the other half of the overhead comes from the directory structure used by Roam (and Rumor) to maintain its own information. Directories are expensive, costing at least one 1KB each, and Roam maintains two two-level directory structures that map into the user's namespace. Overhead would be reduced if the two directory structures were merged into one.

Of course, the above figures of merit are only generalities. The specific breakdown of overhead depends on the actual user volume. A volume with only one directory but many files would have a larger percentage of overhead occupied by the two databases.

Pattern	Number of Objects	Number of Directories	Size
Small	173	38	2.6MB
Space-saver	295	38	5.75MB
Inode-saver	218	38	11.21MB
Subtree	256	31	12.9MB
Full	307	38	13.6MB

Table 9.1: The five replication storage patterns used for studying the effects of selective replication. Each pattern has a different subset of the 13.6MB user data volume from Section 9.1.1. The complete volume stores 307 files, of which 38 are directories. In each case, the total number of objects *includes* the number of directories.

9.1.6 Selective replication effects

With selective replication, users can store only particular fragments of the volume. Selective replication therefore affects the perceived disk overhead, because as the number of files and size of the user's volume changes, the percentage of space occupied by overhead changes.

We studied the effect of selective replication on the disk overhead using five different selective replication patterns from the original 13.6MB volume from Section 9.1.1. Replication pattern "small" attempts to save most of the space by dropping 134 files totaling 11MB. Replication pattern "space saver" does the same by dropping only 12 large files, totaling 7.85MB. Replication pattern "inode saver" saves inodes while preserving most of the data locally by dropping 89 small files, totaling 2.39MB. Replication pattern "subtree" drops 7 directories containing 51 files and totaling 734KB. Replication pattern "full" drops nothing, and is therefore equivalent to the original, complete volume. These replication patterns are summarized in Table 9.1.

For each replication pattern, we created two replicas, performing the selective replication changes and measuring the results at the second replica. Replica 1 stores a fully replicated volume; replica 2 stores different subsets of the volume, according to the pattern. Figure 9.4 illustrates the results, showing two different graphs. The first graph indicates the percentage of disk overhead as a function of the amount of locally stored user data, which is a measure of the cost of selective replication. The second shows the savings in disk space provided by selective replication (including the overhead), which is a benefit of selective replication.

The results of the graph are not surprising. As already demonstrated in the two different volumes described above (Sections 9.1.1 and 9.1.3), the percentage of overhead is inversely proportional to the average file size. Replication pattern "small" has the smallest average file size; we therefore expect it to have the highest percentage of overhead when comparing replication overhead to the amount of user data. As the average file size gets larger, the overhead percentage becomes smaller.

The second set of bars in the graph illustrates a benefit of selective replication. We preserve access to all files in the volume, while potentially saving the user up to 80% of the required disk space, as in the case of pattern "small." Of course, Roam does not currently support transparent remote access; however, the selective replication mechanism maintains all the information necessary for an eventual remote access mechanism.[1]

The opposite extreme of replication pattern "full" would be to have nothing stored locally ("empty"). Since the user's data occupies zero size, the data point is difficult to show as a percentage of overhead on the graph. Instead, we present actual numbers in kilobytes.

When we drop all user files from the volume, the volume occupies 221 1KB disk blocks. (Recall from the formulas in Section 9.1.4 that the second replica of a completely empty volume with no user data requires 57.36KB + 6.44KB = 63.8KB.) The complete volume contains 13.6MB of user data; the "empty" volume can therefore be stored in only 1.6% of the total size of the user's data. Stored in the 1.6% are the

[1] Ficus, an early predecessor, provided transparent remote access with largely the same selective replication implementation.

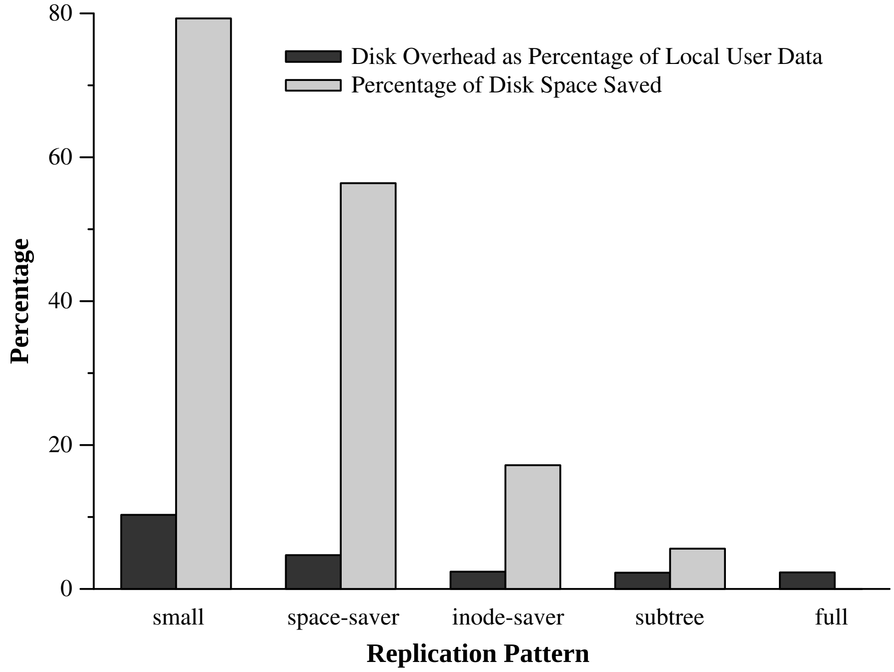

Figure 9.4: Selective replication disk overhead for the 13.6MB user volume from Section 9.1.1. The replication patterns are described in Section 9.1.6 and Table 9.1.

locations of the other replicas and enough information for a transparent remote-access mechanism to locate a data-storing replica of any object in the volume.

9.2 Synchronization Performance

Since ROAM's main task is the synchronization of data, we need to characterize the synchronization performance. If synchronizing two volumes replicas is a painful operation for users, then most users will revert to less costly replication methods, even if they involve manual control or intervention. Since time is a very user-visible cost, we performed a series of experiments to measure the time required to synchronize two volumes replicas.

We performed our experiments with two portable machines. In all cases, we minimized the daemons and processes on the two machines. The first machine is a Dell Latitude XP with a 486DX4 running at 100Mhz with 36MB of main memory. The second is a TI TravelMate 6030 with a 133Mhz Pentium and 64MB of main memory. Reconciliation was always performed transferring data from the Dell machine to the TI machine. In other words, the reconciliation process always executed on the TI machine.

Of course, reconciliation performance depends heavily on the sizes of the files that have been updated. Since ROAM performs whole-file transfers, and any updated file must be transfered across the network in its entirety, we would expect reconciliation to take more time when more data has been updated. We therefore varied the amount of data updated from 0 to 100%, and within each trial we randomly selected the set of updated files. Since the files are selected at random, a given figure of `X`% is only an approximation of the amount of data updated, rather than an exact figure. In all measurements, we used the 13.6MB volume from Section 9.1.1, and unless otherwise mentioned, performed at least seven trials at each data point.

We performed five different experiments, under the above conditions. The first two compare ROAM and RUMOR synchronization performance over a 10MB quiet Ethernet and WaveLAN™ wireless cards

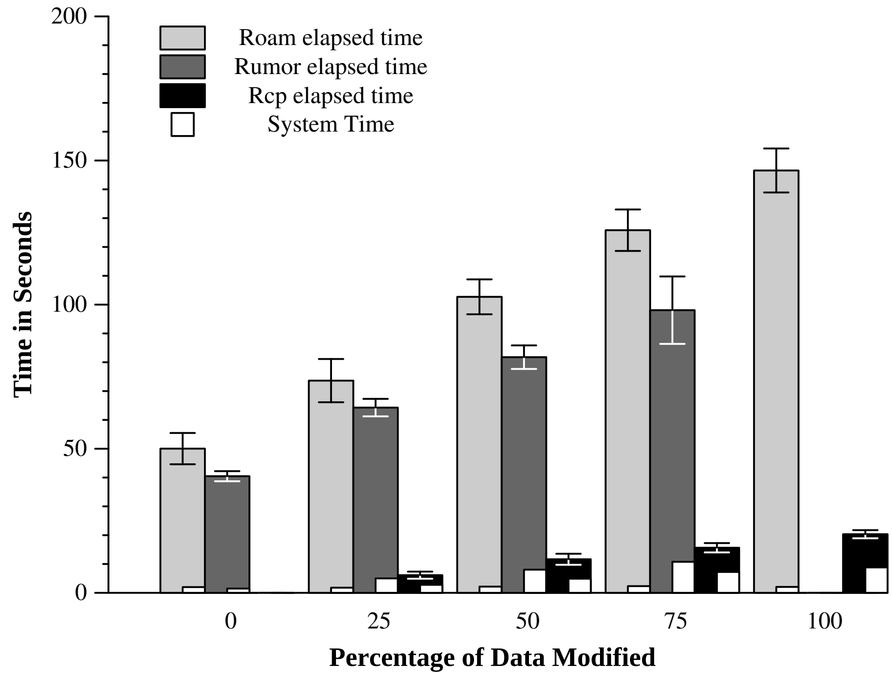

Figure 9.5: ROAM and RUMOR synchronization performance over Ethernet. The volume used is the 13.6MB volume from Section 9.1.1. Five different scenarios are reported, corresponding to 0% to 100% of the data being updated at the remote side. 95% confidence intervals are indicated.

respectively. The third studies the effect of increasing numbers of replicas; the fourth studies the effect of increasing numbers of wards. The fifth looks at the effects of selective replication and different replication patterns on synchronization performance. An experiment designed to illustrate the overall scalability of ROAM is described separately in Section 9.3.

9.2.1 Roam and Rumor over Ethernet

Figure 9.5 displays the relative performance of ROAM and RUMOR over 10MB Ethernet with two replicas of the given volume. As a reference point, the data-transfer time is measured by timing a remote copy command (`rcp`) between the two machines. A data point corresponding to 100% modified using RUMOR is unavailable, because RUMOR bugs prevented reconciliation from completing in that scenario. Each data point represents the mean of at least 14 trials.

The figure illustrates that ROAM is slightly more costly than RUMOR, but not dramatically more expensive. RUMOR elapsed time averages 10-25% better than ROAM, due to several factors in the server architecture (described in Chapter 5). First, ROAM has three processes executing on the machine performing reconciliation, while RUMOR has only one. ROAM separates reconciliation into a recon process and a server process, in addition to the `wardd` which is always running at each host; RUMOR combines the recon and server processes into one. While the `wardd` is only queried once during the reconciliation process (to obtain a socket number for the local `in-out server`), ROAM must nevertheless context switch between the recon and server processes; RUMOR does not pay any context-switching costs.

Second, ROAM generates IPC traffic between the two local processes; for example, to maintain cache consistency. Both the server and recon processes cache the necessary data structures in memory for performance; however, when the recon process updates its data structures, such as when the file-system scanner detects new updates, the server must learn of the changes. An additional source of IPC traffic arises because the recon process and the server process communicate with regard to:

- The objects for which the recon process requested data
- The objects for which the recon process wants acknowledgment of the receipt of new data versions
- The objects for which the server received new data

In general, while the separation of functionality into recon and server processes results in a cleaner and more structured architectural design, it decreases overall performance, due primarily to IPC communication between the two. If we implemented the solution in a shared-memory model, we would expect ROAM's performance to dramatically improve.

Third, once per reconciliation the local `in-out server` forks a separate child `in-out server` process to handle the potential data requests (see the description of the server architecture in Chapter 5.) Our experiments were all performed with a parent `in-out server` already running on the local machine; however, if it was not, the `wardd` would have to spawn a new one. Both `fork` operations add a small but noticeable cost to the elapsed time, even when no data is transferred from the remote site.

With regard to system time, the figure illustrates that, while ROAM's system time remains constant, RUMOR's system time increases linearly with the amount of data transferred. At the 0% level, RUMOR and ROAM spend nearly identical amounts of time in the kernel; however, at the 25%, 50%, and 75% levels, RUMOR spends approximately 3, 3.5, and 5 times more time in the kernel than ROAM respectively. If RUMOR performed correctly at the 100% level, we would expect the trend to continue.

The reason is due to the data transport method. RUMOR invokes a local `popen` of an `rshell` connection to the remote machine, and reads from and writes to that connection. In contrast, ROAM creates a direct socket connection between the local and remote `in-out servers`. RUMOR's method spends a great deal more time in system calls and in the kernel moving buffers around than ROAM's method. As a result, RUMOR doesn't scale very well with respect to increasing amounts of data to be transferred. As the amount of data transferred grows, RUMOR will spend increasingly more time in the kernel, and therefore the kernel will have less time for other processes. Under a "real" workload, when the other daemons and processes were not artificially turned off for measurement purposes, RUMOR's kernel requirements would have a significant effect on the overall performance of the machine.

9.2.2 Roam and Rumor over WaveLAN

We also measured the relative performance of ROAM, RUMOR, and `rcp` when using WaveLAN wireless communication cards, rather than Ethernet. The WaveLAN cards have a maximum data rate of 2MB/second, and use a CSMA/CA media access protocol. Since the communication medium is much slower, we would expect worse reconciliation performance whenever data must be transmitted (i.e. in the 25%, 50%, 75%, and 100% cases). No data is transmitted in the 0% case, and we would therefore expect the reconciliation performance to be largely unchanged.

The results are illustrated in Figure 9.6. Again, a data point corresponding to 100% modified using RUMOR was unavailable, because RUMOR bugs prevented reconciliation from completing in that scenario. Each data point represents the mean of at least 14 trials.

We see largely the same trends as in the Ethernet study. ROAM's performance is slightly worse than RUMOR's, for the reasons described above, and it follows the same general trends as before. The same is true of the system time; RUMOR spends 3-5 times more time in the kernel than ROAM when data has been updated at the remote site. While the elapsed times are in general greater than the Ethernet case, because of the slower data transfer rate, ROAM (and RUMOR) performance is unaffected by the speed of the transport medium when 0% of the data has been updated, since no user data needs to be transferred.

It is interesting to note that, relatively speaking, ROAM's performance improves compared to `rcp`'s at the lower transfer rate. We spend a larger percentage of the time performing data transfer, because the relative speed of the CPU to the transfer medium increased, and therefore we can perform more work while waiting for data to be transfered.

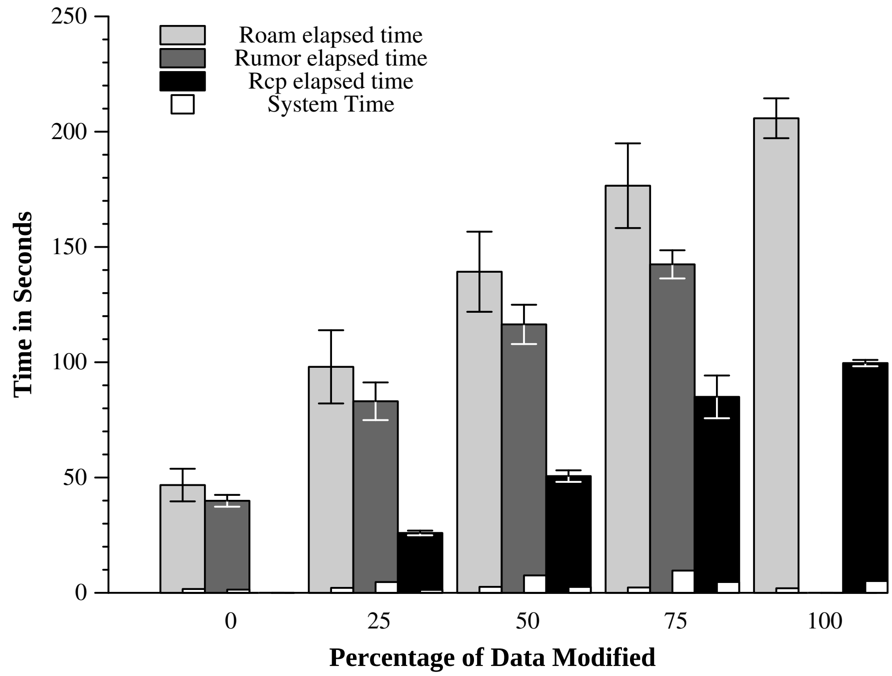

Figure 9.6: ROAM and RUMOR synchronization performance over WaveLAN. The volume used is the 13.6MB volume from Section 9.1.1. Five different scenarios are reported, corresponding to 0% to 100% of the data being updated at the remote side. 95% confidence intervals are indicated.

Additionally, given the performance studies over both Ethernet and WaveLAN, we can compare the relative performance differences of ROAM and RUMOR. A statistical analysis of the difference between ROAM and RUMOR using the formulas for unpaired observations [Jain 1991] indicates that at the 95% confidence level, there is no significant difference in the relative performance of ROAM and RUMOR between the Ethernet and WaveLAN studies. In other words, the difference between ROAM and RUMOR remains constant, independent of the data transfer mechanism or speed. We therefore conclude that the additional time required by ROAM is purely a CPU cost, validating our above analysis with respect to the relative performance of ROAM versus RUMOR.

9.2.3 Impact of multiple replicas

We studied the impact of multiple replicas within the same ward on the synchronization performance. While the previous experiments only measured the performance for a ward with two replicas, we wanted to measure the effect of multiple replicas. We therefore varied the number of intra-ward replicas from two to six. Figure 9.7 illustrates the results.

As demonstrated by the figure, reconciliation performance degrades slightly as the number of replicas increases, but not dramatically. The reasons for the performance degradation are two fold. First, each additional replica increases the size of each file's ward vector. The larger ward vector must be both transmitted from source to target and compared by reconciliation at the target, causing an increase in transmission and compute time. Second, since information regarding the set of volume replicas, their replica identifiers and their physical whereabouts is stored and replicated within the volume itself, each additional replica adds a new file and new directory to the replicated namespace. These two new objects must be scanned by the file-system scanner at both source and target during each reconciliation. Additionally, their attributes are transmitted from source to target and compared by reconciliation at the target, in case the particular vol-

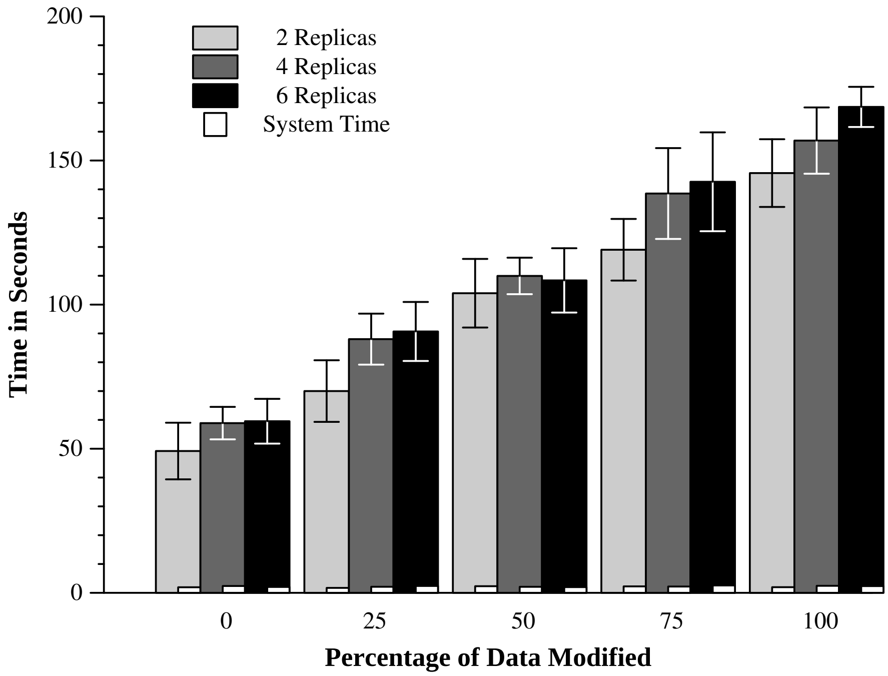

Figure 9.7: The impact of additional replicas on Roam synchronization performance. The volume used is the 13.6MB volume from Section 9.1.1. We vary the number of intra-ward replicas from 2 to 6. Five different scenarios are reported, corresponding to 0% to 100% of the data being updated at the remote side. 95% confidence intervals are indicated.

ume replica has moved physical locations. Regardless of whether the volume replica has moved or not, the two additional file system objects increase both the compute time at source and target and the transmission time from source to target (for the file's attributes, not data). The figure illustrates that the performance degradations are not extreme; nevertheless, it clearly indicates that the system could not adequately support large numbers of replicas if they were all members of the same ward (such as in Rumor, which essentially places all replicas in one "ward").

9.2.4 Impact of multiple wards

We also studied the impact of multiple wards the synchronization performance. While the previous graphs only measured the performance with one ward, we wanted to measure the effect of multiple wards. We therefore varied the number of wards from one to six. Within one of the wards, we placed three replicas, and measured the synchronization between two of them on the previously described portable machines. Therefore, the performance data reflects the synchronization performance for three replicas within a single ward, while we vary the number of additional wards. Each additional ward only contains a single replica.

Three replicas were used instead of just two because we wanted to study just the synchronization performance between normal ward members, and not involve the ward master. Since the ward master's data structures increase proportionately with the number of wards, we would expect that the time required to reconcile with the ward master would depend somewhat on the number of wards; however, our belief is that synchronization between normal ward members would be unaffected.

Figure 9.8 illustrates the results. It demonstrates that the additional wards have essentially no significant effect on the elapsed time for reconciliation between normal ward members. The ward master isolates the ward members from the effects of large scale; the members' data structures do not reflect the additional

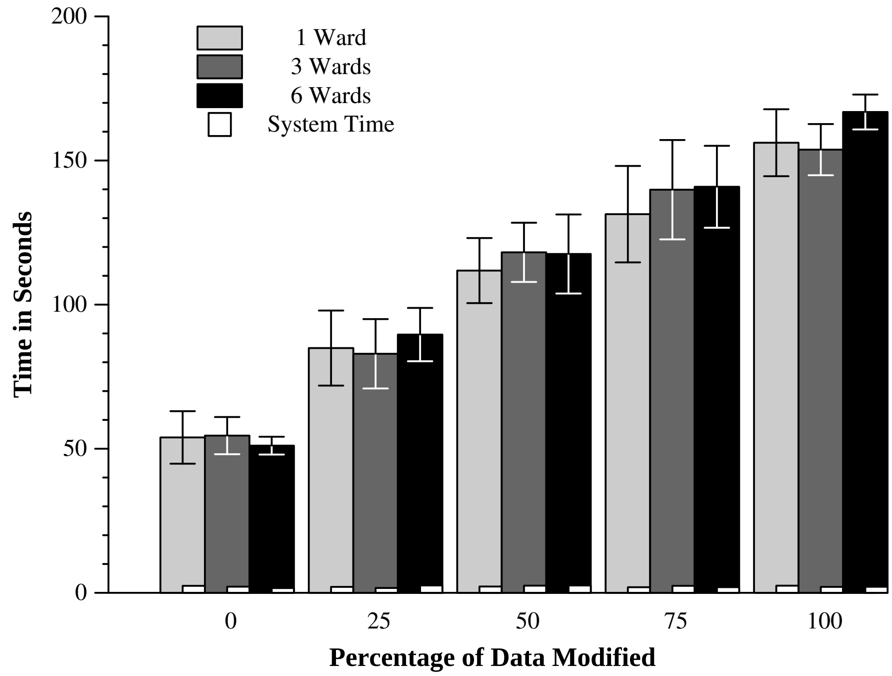

Figure 9.8: The impact of additional wards on Roam synchronization performance. The volume used is the 13.6MB volume from Section 9.1.1. We vary the number of wards from 1 to 6, with the measured ward having 3 replicas; synchronization is between the two non-master replicas. Five different scenarios are reported, corresponding to 0% to 100% of the data being updated at the remote side. 95% confidence intervals are indicated.

wards, so it is understandable that their synchronization performance would not be affected by their presence. Figure 9.8 demonstrates an important aspect of Roam's scalability. While the number of wards may increase, the synchronization performance within each ward remains constant within statistical error.

9.2.5 Impact of selective replication

We previously demonstrated that selective control did not create any additional performance costs for users during normal operation in Ficus, as measured by a series of micro-benchmarks [Ratner *et al.* 1996a, Ratner 1995]. However, it is probably more important to characterize the actual performance of synchronization under different selective replication patterns and workloads. We performed our experiments with the same two portable machines described in Section 9.2 and in the same environment.

We utilized the previous five replication patterns at the local (TI) machine. Additionally, since reconciliation performance depends heavily on the sizes of the files that have been updated, we varied the amount of data updated at the remote site (and therefore the amount to be transfered by reconciliation to the local site) from 0 to 50%. Data files are selected at random for update; they are not pre-determined. We performed 7 runs for each data point; the results are illustrated in Figure 9.9.

From the figure, we see that selective replication saves the user as much as half of the cost of reconciliation, comparing pattern "small" to pattern "full." As one would expect, the amount of data updated at the remote side in pattern "small" makes little difference, since almost none of it is stored at the local replica. In contrast, we see almost a two-fold rise in reconciliation time at pattern "full" between 0 and 50% of the data being modified. In general, the more data modified at the remote side, and the more data locally stored, the longer reconciliation takes. The result seems obvious in retrospect, and is certainly the

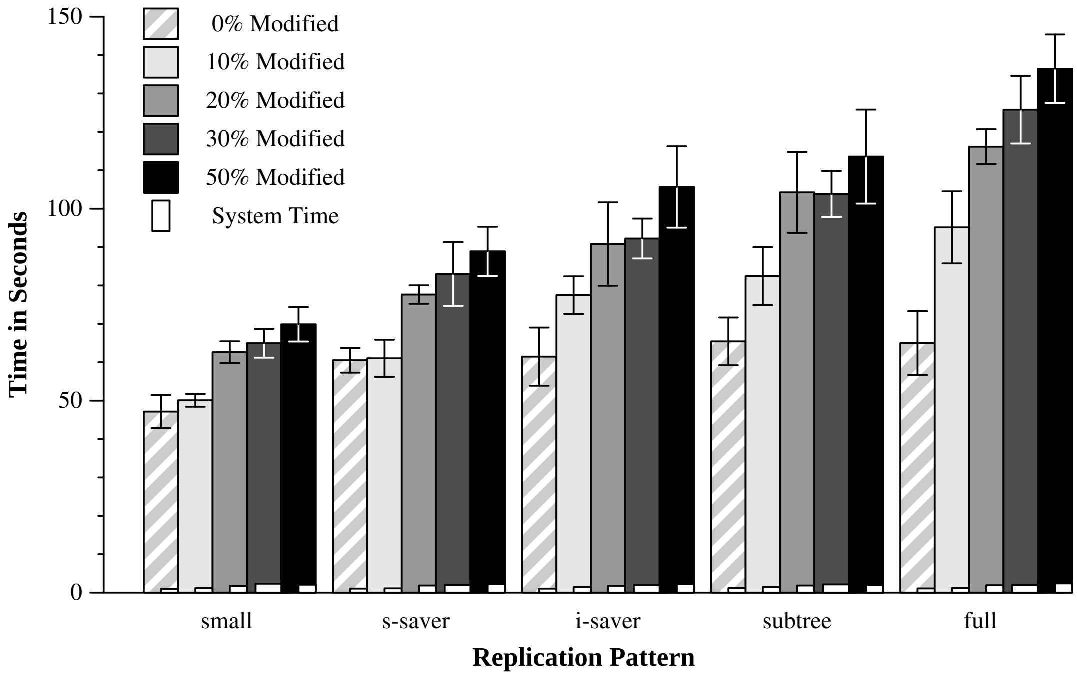

Figure 9.9: The impact of selective replication on reconciliation performance. The replication patterns are described in Table 9.1. 95% confidence intervals are shown. Except where indicated, all measurements are of elapsed time.

way a well-designed system should perform; nevertheless, measurements were required to validate and prove the result. The figure demonstrates a second clear benefit to selective replication, in addition to the disk space savings demonstrated above: reconciliation takes less time. The cost, of course, is that access to the non-local data takes longer as well; how much longer depends on the transport mechanism and the relative location of the data-storing replicas.

We also see from the figure that, even when nothing was modified at the remote site, reconciliation took longer under replication pattern "full" than replication pattern "small." Since more files are stored locally in replication pattern "full," reconciliation spends more time detecting if any updates have been generated locally (to update its local data structures and version vectors), regardless of whether or not updates were applied at the remote site. The fewer files managed and stored locally by the replication system, the better reconciliation performs.

9.3 Scalability

We have already demonstrated some aspects of Roam's scalability, such as in disk space overhead (Section 9.1). However, another major aspect of scalability is the ability to create many replicas and still have the system perform well during synchronization. Synchronization performance includes two related issues. First, the reconciliation time for a given replica in a given ward should be largely unaffected by the total number of replicas and wards. Second, the time to distribute an update from any replica to any other replica should presumably be faster in the Ward Model than in a standard approach (like Rumor), or else we have failed in our task. We will now address both issues in detail.

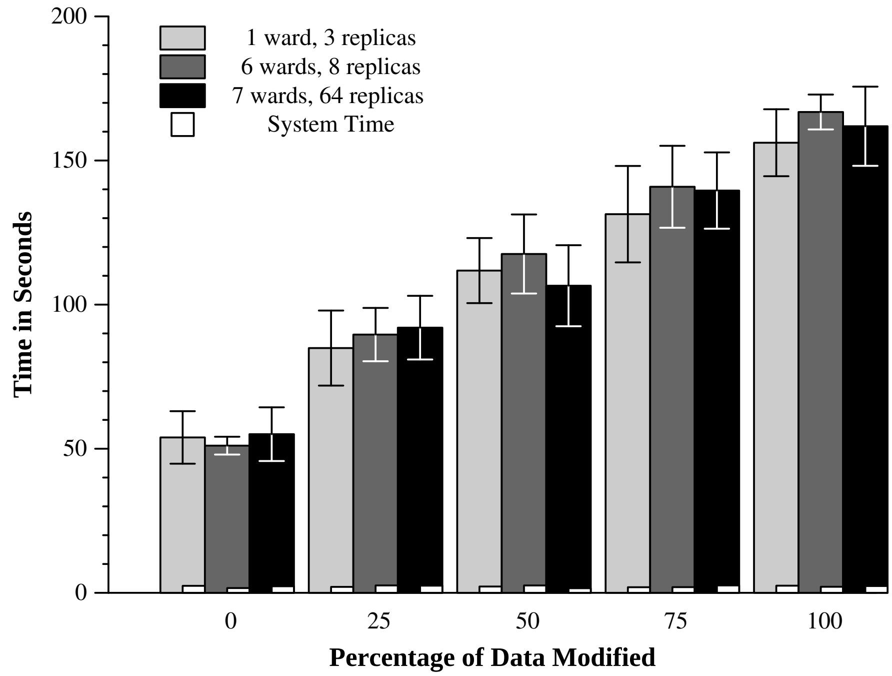

Figure 9.10: A hybrid simulation demonstrating synchronization performance as we vary the number of replicas and wards. Synchronization in the 64-replica case is measured between two members in the one ward of 4. The volume is the 13.6MB volume from Section 9.1.1. The amount of data modified at the remote side varies from 0% to 100%. 95% confidence intervals are indicated.

9.3.1 Synchronization performance

As evidenced by the measurements in Section 9.2, the synchronization performance in both Roam and Rumor becomes worse with the addition of more replicas in the given ward. Clearly, the system would not scale without wards. However, the analysis of multiple wards above (Section 9.2.4) only studied wards with small numbers of members. To properly demonstrate that the Ward Model scales, we need to create a large number of replicas.

We used a hybrid-simulation approach to create 64 replicas of the 13.6MB volume described above. Rather than collecting 64 separate portables, interconnecting them and creating a replica on each, we gained the same effect by creating multiple wards on two 200Mhz Pentium Pro server machines. Six wards of ten members each were created on these two machines. The seventh ward contains four replicas (so we can easily compare our performance numbers against the ones previously measured) on real, portable machines. We have therefore simulated the effect of 64 replicas across 7 wards. The machines are real; in many cases, the network is simulated, by virtue of many of the replicas and wards existing on the same machine. However, we can still measure the performance between two portable machines that are connected by a real, physical network.

Figure 9.10 shows the results. Figure 9.7 previously demonstrated a steady rise in elapsed time as the number of replicas increased; here we demonstrate that multiple wards isolate those effects. Synchronization performance does not significantly change as we move from a 1-ward, 3-replica system to a 7-ward, 64-replica system. We therefore conclude that increased scale does not affect the intra-ward synchronization performance. The intra-ward synchronization performance depends entirely on the number of intra-ward replicas, as opposed to the total number of replicas or wards. The scalability of Roam therefore depends entirely on separating replicas into wards.

9.3.2 Update distribution

Another aspect of scalability concerns the distribution of updates to all replicas. A scalable system would presumably deliver updates to all replicas more quickly than a non-scalable system, at least at large numbers of replicas. Additionally, while it may not perform better at small numbers of replicas, a scalable system should at least not perform worse.

We analytically developed equations that characterize the distribution of updates. Rather than measuring elapsed time, which depends on many complicated factors such as connectivity, network partitions, available machines, and reconciliation intervals, we measured the number of individual, pair-wise reconciliation actions. We assume that there are $\mathtt{M}$ replicas; one of them, replica $\mathtt{R}$, generates an update that must propagate to all other replicas. The following equations identify the number of separate reconciliation actions that must occur, both on the average and in the worst case, to propagate the update from $\mathtt{R}$ to some other replica $\mathtt{S}$.

In RUMOR, since there are $\mathtt{M}$ replicas, $\mathtt{M}-1$ of which do not yet have the update, and reconciliation uses a ring between all $\mathtt{M}$ replicas, we need $\frac{\mathtt{M}-1}{2}$ reconciliation actions on average. The worst case requires $\mathtt{M}-1$ reconciliation actions.

The analysis for ROAM is a little more complicated. Assume that the $\mathtt{M}$ replicas are divided into $\mathtt{N}$ wards such that each ward has $\mathtt{M}/\mathtt{N}$ members. Propagating an update from $\mathtt{R}$ to $\mathtt{S}$ requires first sending it from $\mathtt{R}$ to $\mathtt{R}$'s ward master, then sending it from $\mathtt{R}$'s ward master to $\mathtt{S}$'s ward master, and then finally to $\mathtt{S}$. Of course, if $\mathtt{R}$ and $\mathtt{S}$ are members of the same ward, then much of the expense is saved; however, we will solve the general problem first before discussing the special case.

Under the above conditions, we need $\frac{1}{2}(\frac{\mathtt{M}}{\mathtt{N}}-1)$ reconciliation actions on average to distribute the update between a replica and its ward master, and $\frac{1}{2}(\mathtt{N}-1)$ actions on average between ward masters. From these building blocks, we calculate that, on average, ROAM requires the following number of reconciliation actions:

$$\begin{aligned}
&\frac{1}{2}(\frac{\mathtt{M}}{\mathtt{N}}-1)+\frac{1}{2}(\mathtt{N}-1)+\frac{1}{2}(\frac{\mathtt{M}}{\mathtt{N}}-1)\\
=\;&\frac{\mathtt{M}}{\mathtt{N}}-1+\frac{1}{2}(\mathtt{N}-1)\\
=\;&\frac{\mathtt{M}}{\mathtt{N}}+\frac{(\mathtt{N}-3)}{2} \qquad (9.1)
\end{aligned}$$

Note that when $\mathtt{N}=\mathtt{M}$, Equation 9.1 becomes $\frac{\mathtt{M}-1}{2}$ (RUMOR's performance). Setting $\mathtt{N}=\mathtt{M}$ eliminates any benefit from grouping. However, it is also interesting to note that when $\mathtt{N}=2$, Equation 9.1 *also* becomes $\frac{\mathtt{M}-1}{2}$. Having only two wards does not improve the required time to distribute updates (although it does improve other aspects such as data structure size and network utilization).

In general, ROAM distributes updates faster than RUMOR when $2<\mathtt{N}<\mathtt{M}$ and $\mathtt{M}>3$; otherwise, ROAM performs the same as RUMOR (with respect to update distribution). From the two equations we calculate that the optimal number of wards for a given value of $\mathtt{M}$ is $\mathtt{N}=\sqrt{2\mathtt{M}}$. The above conditions yield a factor of three improvement at 50 replicas, and a factor of five at 200 replicas. With a multi-level implementation, larger degrees of improvement are possible.

The analysis for ROAM also indicates that, in the worst case, ROAM requires $\frac{2\mathtt{M}}{\mathtt{N}}+\mathtt{N}-3$ reconciliation actions.

As a special case, if $\mathtt{R}$ and $\mathtt{S}$ are in the same ward, only $\frac{1}{2}(\frac{\mathtt{M}}{\mathtt{N}}-1)$ reconciliation actions are required on average, and $\frac{\mathtt{M}}{\mathtt{N}}-1$ in the worst case.

9.4 Ward Motion

Recall from Chapter 4 that ROAM supports two different flavors of ward motion: overlapping and changing. Overlapping is supposed to be a lightweight, temporary form of motion that is easy to perform and undo. However, synchronization performance can become worse during overlapping. When the moving replica

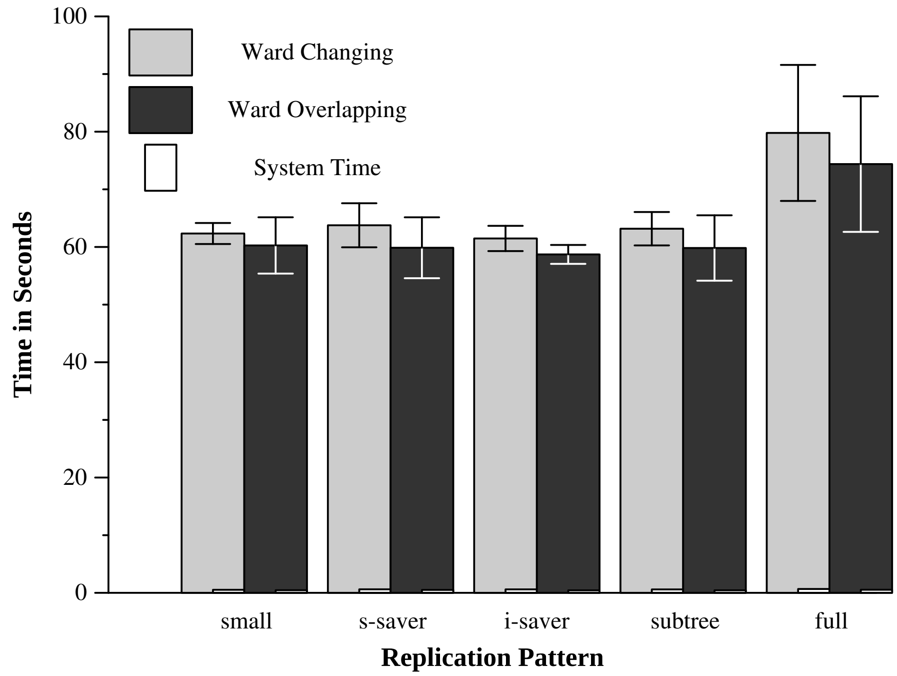

Figure 9.11: Time needed to perform ward motion. Both elapsed and system time are indicated. The volume is the 13.6MB volume from Section 9.1.1. The replication patterns are described in Table 9.1.

stores objects that are not part of the new ward, they must be synchronized with the original ward (or else remain unsynchronized during the presumably short time period). Changing is supposed to be a more heavyweight, permanent form of motion that costs more but provides optimal synchronization performance in the new ward. Here we examine, illustrate, and explain the costs of both forms of ward motion.

9.4.1 Initial operation cost

We measured the time to perform each operation, using the 13.6MB user volume from Section 9.1.1. Additionally, we varied the set of data stored on the moving replica according to the five replication patterns specified earlier in Table 9.1.

For the experiment, we created two wards. The first ward had a ward master and a second replica, deemed the "moving" replica. The second ward only had a ward master. The experiments were performed on 200Mhz Pentium Pros, connected by standard Ethernet. The moving replica's machine has 96MB of memory; the ward master for the second ward is on a machine with 64MB of memory. Each data point reflects the arithmetic mean of six runs; 95% confidence intervals are indicated.

Figure 9.11 illustrates the results. The graph demonstrates that while overlapping is less expensive for the user in terms of elapsed time, the difference is almost insignificant. Changing never takes more than 7% longer to complete. The real difference in cost lies not in the time to perform the operation, but in the effects of the operation on the local replica and on the system as a whole, which are explored more fully in the following section.

In general, the graph illustrates that ward motion is not a tremendously expensive operation for users to perform. Even with the entire volume stored locally, the operation only took 80 seconds on a 13.6MB volume. Ward motion currently performs two one-way reconciliations, since information must be exchanged in each direction. If we implemented true two-way reconciliation, we believe that the required time would be roughly halved. Additionally, much of the overhead involved in ward motion consists of loading Perl programs and spawning remote processes to execute on the remote machine. There is clearly great possibilities for optimization. Nevertheless, we are encouraged by the current results.

9.4.2 Real operational costs

As previously stated, the real difference in cost between overlapping and changing lies in the effects the two operations have on the replicas and the system. Ward overlapping, for instance, generates additional disk overhead at the moving replica, since the moving replica maintains its membership in two wards rather than just one. Ward changing produces no new overhead at the moving replica; it can in fact decrease the overhead, when moving into a ward with less members than the previous ward.

Additionally, all files that are not stored within the new ward must be synchronized with the old ward, or else remain unsynchronized during the duration of the ward overlap operation. After ward changing, all files on the moving replica can be synchronized within the local ward.

On the other hand, ward changing has its costs as well. When the moving replica stores files that are not stored within the new ward, the new ward expands its ward set to include them. The changes in the ward set are then distributed via normal reconciliation to the other ward masters on a "need to know" basis, as explained in Chapter 4. The ward set of the old ward potentially changes as well: if the moving replica stored the only copy within the old ward, then the old ward set shrinks in size. At one extreme, the changes in the ward set must be propagated to all other ward masters, although the propagation occurs via gossiping and therefore it is not the responsibility of one ward master to individually communicate with every other ward master.

We will now more quantitatively describe the real costs associated with ward motion.

Ward overlapping costs

Figure 9.12 shows the increase in disk overhead at the moving replica as a result of the overlap operation. The moving replica stores the 13.6MB user volume from Section 9.1.1, subject to the previous five replication patterns (Table 9.1). We illustrate three different scenarios that differ only in the number of ward participants in the new ward. Scenario 1 shows the effect when the new ward has only one member, regardless of the number of members in the old ward. Scenarios 2 and 3 demonstrate the overhead when the new ward stores four and seven members, respectively.

As the graph illustrates, the disk overhead depends not on the particular selective replication pattern or amount of locally stored data but instead on the number of members in the new ward. The dominant source of overhead comes from the directory structure used to store the information about wards and their participants. The directory structure overhead overwhelms the small increase in data structure size when the additional ward vector is added to each filelist entry (unless the volume contained millions of files). In general, the increase in overhead is basically insignificant when wards have moderate numbers of members.

Additionally, however, we must characterize the additional synchronization cost when ward overlapped. Recall that, when the mobile machine's replica set has a non-empty set difference with the new ward's ward set, some of the mobile machine's data must be synchronized with the old (distant) ward. To measure the cost, we devised the following experiment. We created two wards, $\mathtt{W}_1$ and $\mathtt{W}_2$, each with only one member (the ward master) and applied the previous five selective replication patterns (Table 9.1) to the ward master of $\mathtt{W}_2$. We then created another replica $\mathtt{R}$ as a member of ward $\mathtt{W}_1$ and, using ward overlapping, moved it into ward $\mathtt{W}_2$. Since ward $\mathtt{W}_2$ does not store all files in the volume, replica $\mathtt{R}$ cannot synchronize all of its files solely within ward $\mathtt{W}_2$; communication with ward $\mathtt{W}_1$ is required. For simplicity of presentation, we refer to ward $\mathtt{W}_1$ as ward $\mathtt{W}_{\text{orig}}$ and $\mathtt{W}_2$ as ward $\mathtt{W}_{\text{overlap}}$.

Communication with the old ward is presumably more expensive than communication with the new, local ward. Therefore, our experimental framework has replica $\mathtt{R}$ communicate with ward $\mathtt{W}_{\text{overlap}}$ via Ethernet and with ward $\mathtt{W}_{\text{orig}}$ via WaveLAN. Reconciling via WaveLAN mimics the additional cost of long-distance communication.

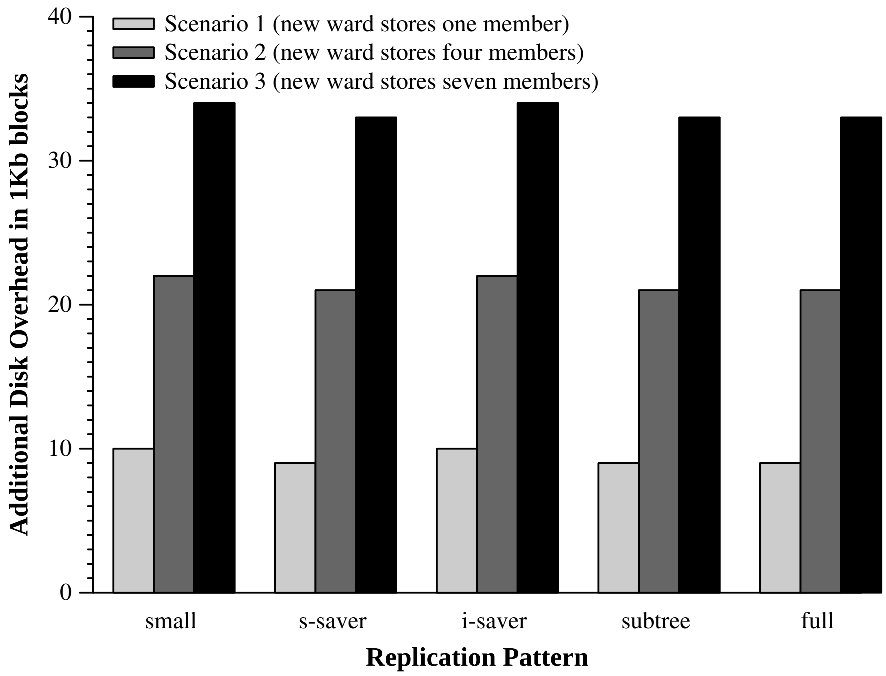

Figure 9.12: The resulting disk overhead after performing a ward overlap operation. The volume is the 13.6MB volume from Section 9.1.1. The replication patterns are described in Table 9.1. The three scenarios depicted are the result of different size wards, as described in Section 9.4.2.

Additionally, the cost of R's long-distance synchronization with the old ward depends on the amount of data that has been updated, and how many of the updates R was able to obtain from reconciliation with ward $\texttt{W}_{\text{overlap}}$. We varied the amount of data updated from 0 to 100%. Updates were applied at random to the complete volume stored within ward $\texttt{W}_{\text{orig}}$. Ward $\texttt{W}_{\text{overlap}}$ obtained some of these updates via reconciliation with $\texttt{W}_{\text{orig}}$ but not all of them, since it selectively stores only some of the volume's files, according to the five replication patterns. When R reconciles with $\texttt{W}_{\text{overlap}}$, it will possibly receive some of the updates; when R reconciles with $\texttt{W}_{\text{orig}}$, it will receive the remainder.

The experiment has one drawback: namely, that the time as measured is overly conservative. Given the constraints of the measurement mechanism, the time indicated includes that of performing a local scan at replica R, a scan at the ward master $\texttt{W}_{\text{orig}}$, and the transmission cost. However, when R actually performs reconciliation, it only scans its file system once, prior to reconciliation with $\texttt{W}_{\text{overlap}}$, and does not scan its local file system again. Our measurement therefore includes the time for an extra local scan. However, the absolute measurements are not as important as the relative performance between data points, especially since the use of WaveLAN is only to mimic the effects of long-distance transmission, and is not intended to be indicative of the exact transmission cost. Furthermore, since cost of the local scan at replica R is constant for all data points, we can easily infer the overall trends from the experiment.

We performed 7 trials for each data point in our experiment, and indicate 95% confidence intervals. As before, reconciliation was performed using the Texas Instruments TravelMate and Dell Latitude XP portable machines. R is on the TI machine; the ward master of $\texttt{W}_{\text{orig}}$ is on the Dell. Figure 9.13 illustrates the results.

The elapsed time is essentially constant in the 0%-modified case, regardless of how much data is stored with ward $\texttt{W}_{\text{overlap}}$. Replica R pays a small cost to contact $\texttt{W}_{\text{orig}}$ and inquire about updates; since there are none, reconciliation performance is constant. In the "small" case, reconciliation performance degrades rapidly as more data is updated, since most of that data must be transferred from the old ward. Performance

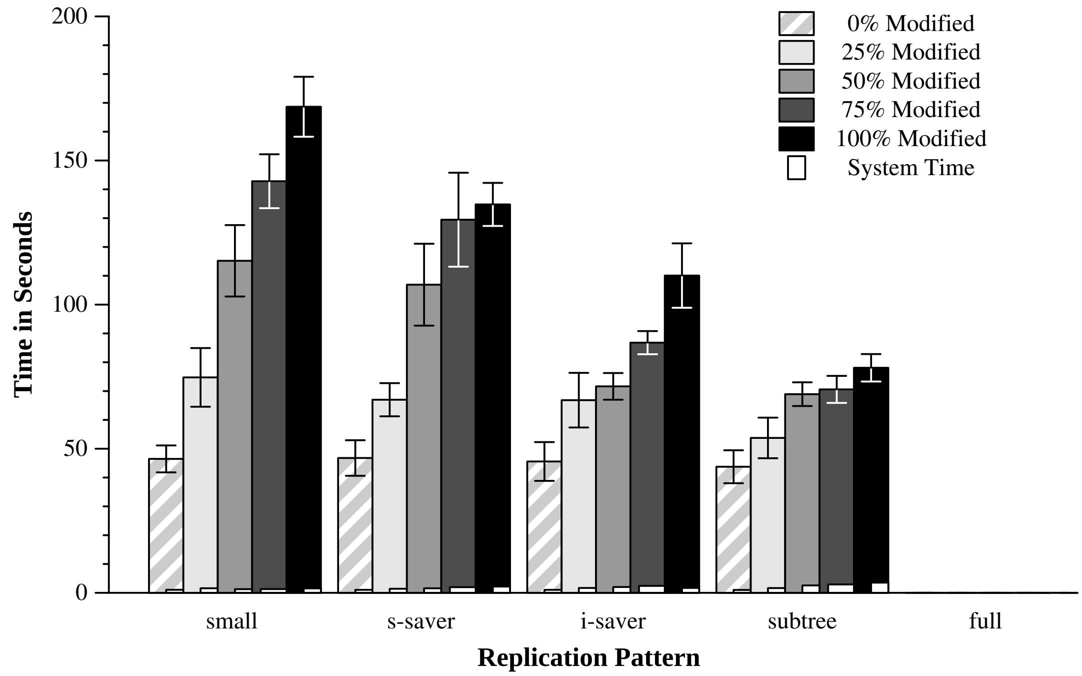

Figure 9.13: Synchronization performance with the old ward master, after ward overlapping and reconciling with the new (overlapped) ward master. The volume is the 13.6MB volume from Section 9.1.1. The replication patterns are described in Table 9.1 and are applied to the new ward master. Reconciliation is performed over WaveLAN, as described in Section 9.4.2.

also degrades under replication patterns "space-saver", "inode-saver", and "subtree", although not as much in each case. Since more of the updates can be obtained from the new, local ward master, less of them need to be transferred via reconciliation with the old ward. For example, in the "subtree" case, the new ward stores most of the files in the volume; only a few of them need to be fetched from the old ward. Performance therefore degrades much more slowly than in the "small" scenario, even when the same amount of data has been updated. Additionally, we notice that, when most of the data has been updated, reconciliation takes longer under the "space-saver" pattern as opposed to the "inode-saver" pattern. Since in the "space-saver" case we drop a few very large files, reconciliation must pay the (large) transmission time to send their updates, while in the "inode-saver" case, we must only transmit a number of very small files from the old ward.

In the "full" case, reconciliation time is 0, regardless of how much data has been updated. Since the new ward stores the complete volume, all updates can be obtained via local reconciliation; reconciliation with the old ward is not required.

In summary, assuming the new ward does not locally contain everything stored by the mobile replica, users pay a small, constant cost during synchronization, even when nothing has been updated. Of course, the size of the cost depends on transmission speeds, latency, and other network characteristics. The user additionally pays for each file that must be transferred from the old ward: the larger the file, the higher the cost, due to transmission time.

However, it should be mentioned that since ward overlapping is a temporary phenomenon, users have the option of not paying the additional reconciliation cost at all. Instead, they can decide not to receive updates from the old ward until they return (or else upgrade to ward changing, in which case all updates can be received through the local ward). The data that would normally be synchronized with the old ward slowly diverges. The cost of out-of-date data depends on the value of the updates to the particular user. If

PowerPoint were updated at the old ward, many users might choose not to pay the cost of long-distance transmission while ward overlapped. However, if the updated file were a sales report or another critical file, users may well value the update more than cost of transmission.

Ward changing costs

The ward set at the new ward potentially expands as a result of ward changing, and the ward set at the old ward potentially shrinks. Both changes need to be propagated to the other ward masters. Since the changes occur asynchronously and via gossiping, the associated costs are difficult to measure directly. Instead, we characterize the *amount* of changed data, rather than the *time* required to distribute the changes.

The ward set at the new ward expands by the number of file objects stored at the moving replica that were not previously stored within the new ward. For each file object added to the ward set, the ward master must store an entry in its lookaside databases (the filelist and inode-ffsh-map), described in Section 9.1.4. Each inode-ffsh-map entry requires approximately 35 bytes per object, and each filelist entry requires approximately 225 bytes per object. Therefore, each new file added to the ward set causes an additional 255 bytes to be stored at the ward master. The ward master need not store an actual copy of the file data (it can store a virtual replica), but if it does store a replica, the required disk space increases by the size of the object.

The other ward masters must learn about the changes in the ward set, which amounts to propagating to them the 255 bytes per new object. The changes are propagated to the ward masters in a "need to know" fashion, so that there is no cost placed on ward masters that don't care about the given change; however, the worst case from a cost-analysis occurs when distributing the changes to all ward masters.

If we assume the ward set increases by $\mathtt{n}$ objects, then $255\mathtt{n}$ bytes must be distributed to all other ward masters. The cost is not extraordinary; an increase in 100 files causes only an overhead of 25.5KB. Nevertheless, the overhead must be distributed to all ward masters. Given $\mathtt{m}$ ward masters, we effectively introduce a network overhead of $255\mathtt{n}(\mathtt{m}-1)$ bytes. Assuming fifty ward masters and our previous 100 file increase, the network overhead amounts to 1.25MB. In other words, the network must in the worst case deliver 1.25MB.

Of course, the actual network cost depends a great deal on the reconciliation topology. With a broadcast protocol, the network expense would be basically independent of the number of ward masters. With most topologies, such as the Roam's adaptive ring, each $255\mathtt{n}$-byte overhead is generally transfered over a different network link; therefore the cost is not really as imposing as it seems. Additionally, since we utilize a gossip-based form of communication, and the distribution occurs during normal reconciliation, each ward master could potentially add new information to the $255\mathtt{n}$-byte overhead. If both ward masters $\mathtt{M}_1$ and $\mathtt{M}_2$ were expanding their ward sets, and the expansions happened to be on the same $\mathtt{n}$ files, each ward master simply adds its new information to the existing $255\mathtt{n}$-byte overhead, rather than generating an additional set of network overhead. In general, if $\mathtt{M}_1$'s ward set increased by $\mathtt{n}_1$ files, and $\mathtt{M}_2$'s ward set increased by $\mathtt{n}_2$ files, with $\mathtt{p}$ files being common to both ($\mathtt{p} \leq \mathtt{n}_1$ and $\mathtt{p} \leq \mathtt{n}_2$), then the combined overhead in bytes is

$$255\mathtt{p} + 255(\mathtt{n}_1 - \mathtt{p}) + 255(\mathtt{n}_2 - \mathtt{p}) < 255\mathtt{n}_1 + 255\mathtt{n}_2 \text{ for } p > 0$$

which must be distributed to all $\mathtt{m}$ ward masters. Gossiping thereby allows the costs to be combined in a non-additive manner.

Additionally, since the overhead is distributed during normal reconciliation, each ward master potentially includes information about new updates. For example, if all $\mathtt{n}$ files had updates that needed to be distributed to the other ward masters, the extra overhead attributable to ward changing would be zero, since the information must be transfered anyway to maintain consistency. In general, if $\mathtt{p}$ of the $\mathtt{n}$ files had updates to be distributed, then the overhead is $255\mathtt{p}(\mathtt{m}-1)$ bytes.

9.5 Experience with Real Use

For over four months, all ROAM development has occurred under ROAM. The sources, object files, and executables were replicated between office machines and laptops using ROAM as the replication substrate. We used RUMOR early in the development cycle, before ROAM was very robust, but RUMOR generated problems than benefits, due mainly to garbage-collection issues. The volume containing the sources, objects files, and executables totals nearly 200MB. Each time a `make clean` operation was performed, to remake the executables from scratch, RUMOR would not garbage-collect any of the disk space until a full two rounds of garbage collection could complete. Thus, after a `make clean` operation, the RUMOR volume requires nearly 400MB of disk space, until the extra 200MB could be properly garbage collected. The laptop exhausted its disk space multiple times, causing problems ranging from minor headaches to actual data loss. With ROAM's new garbage-collection semantics, the volume actually resides within approximately 200MB, and once we switched replication substrates to ROAM, we no longer experienced any disk space problems.

Multiple people within the UCLA community are actively using ROAM for their replication requirements. Additionally, a SEER [Kuenning *et al.* 1997] interface is being developed, so that SEER could use ROAM as its replication substrate for predictive hoarding of files on portable computers. However, few if any people are actively creating multiple wards, so the scalability features of ROAM have been largely untested in real use, mostly due to the practical and social issues involved with large-scale deployment of software. Although our simulations and tests indicate that ROAM should behave quite nicely, real experience in large scale environments is clearly required before we can adequately comment on ROAM's use in such scenarios.

Chapter 10

Related Work

ROAM contains a number of important facets. First and foremost it is a replication system, but additionally it contains important distributed algorithms such as those governing garbage collection, version vector management, and selective replication. The related work, therefore, covers a number of different areas. We will first discuss a number of related replication systems, and then discuss the previous work on garbage collection, version vectors, selective replication, and scaling.

10.1 Replication Systems

CODA, LITTLE WORK, BAYOU, DECEIT, LOCUS, FICUS, RUMOR, and others are all examples of optimistic replication systems that have come out of the research community. Additionally, there are commercial and share-ware replication systems, such as LOTUS NOTES™ and RDIST. Each system will be discussed briefly.

10.1.1 Coda

The CODA file system [Satyanarayanan *et al.* 1990, Kistler *et al.* 1991] is an optimistically replicated file system constructed on a client-server model, as opposed to the peer architecture proposed by ROAM. CODA provides replication flexibility akin to selective replication at the clients, but not at the replicated servers, which are traditional peers. Additionally, CODA clients cannot directly inter-communicate due to the inherent restrictions of the client-server model. The rigidity of the model dramatically simplifies the consistency algorithms at the cost of limiting the system's utility for mobility. It is often expensive, if at all possible, for clients to contact the server; in these cases (and often in many other scenarios) the clients should be able to directly synchronize among themselves. In CODA, they cannot.

CODA claims good scalability in the number of clients, but no data has ever been published. CODA permits multiple replicated servers, but again no data has ever been published with regard to how well the server architecture scales. All known CODA installations use tightly connected servers; it unclear how well CODA operates when the servers are much more geographically distributed or when the servers are not as well connected. Additionally, it is unclear whether or not CODA lends itself to a mobile environment where a client connected to server `A` could travel and re-connect to a remote server `B`, when `A` and `B` are not well-connected.

One area in which CODA is clearly superior is the low-bandwidth scenario. CODA has greatly optimized the communications and synchronization between client and server, especially in environments with weak connectivity [Kistler *et al.* 1992, Mummert *et al.* 1995]. Some of the same ideas could be applied in ROAM; however, more research is required in optimizing communication protocols for weak connectivity in peer models.

10.1.2 Little Work

The Little Work project [Honeyman *et al.* 1992] is similar to Coda, but modifies only the clients, leaving the AFS [Howard *et al.* 1988] servers unaltered. Congestion caused by client's slow links is reduced in a variety of ways, including client-side modifications of AFS, its underlying RPC, and other congestion avoidance and control methods. However, again clients cannot directly communicate, hindering the usability of the system in dynamic, mobile environments. Additionally, the method used to populate the client with the necessary data is sub-optimal compared to other strategies [Kistler *et al.* 1992, Kuenning 1994].

10.1.3 Bayou

The Bayou system [Terry *et al.* 1995] is another replicated storage system, replicating databases rather than file system objects. Like Roam, it is based on the peer-to-peer model and provides support for application-dependent resolution of conflicts. However, unlike Roam, Bayou does not attempt to provide transparent conflict detection. Applications must specify a condition that determines when a conflicting access has been made, and must specify the particular resolution process.

Bayou provides *session guarantees* [Terry *et al.* 1994] to improve the perceived consistency by users. Additionally, Bayou establishes strong guarantees about its data—writes can be classified either as *committed* or *tentative*. Similar research has been done in optimistic file systems [Goel 1996] and could be applied to Roam to offer many of the same types of guarantees.

Bayou does not provide any form of selective replication. The databases must be entirely replicated at all storage sites.

Bayou supports mobility in the same way as all systems based on the traditional peer-to-peer model. They allow direct any-to-any communication, but suffer the same scaling problems as Rumor and Ficus when faced with mobile environments.

10.1.4 Deceit

The ISIS environment's Deceit file system [Siegel *et al.* 1990, Siegel 1992] places all files into one "volume" and allows each individual file to be replicated independently with varying numbers of replicas. In this sense it provides selective replication. However, Deceit employs a conservative approach to replication, namely a writer-token mechanism, and therefore cannot provide the high availability offered by optimistic mechanisms. In addition, Deceit cannot tolerate long-term network partitions; instead, only a related failure known as a *virtual partition* [Siegel *et al.* 1990], which eventually corrects itself, is tolerated. These factors make the Deceit system unsuitable for the environments discussed in Chapter 2.

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

Our selective replication implementation solved the local availability problem by enforcing the system invariant of full backstoring (Chapter 6). Deceit instead implements the directory structure using an in-memory binary tree of all hard links, backed up to non-volatile storage. Similar to the "prefix-pointer" mechanism [Welch *et al.* 1986], Deceit's solution has the advantage that it saves the system from locally storing the intermediate directories, but has the disadvantage of added complexity, both to maintain the list and to garbage collect from it. Furthermore, the in-memory tree can potentially absorb much of main memory.

10.1.5 Locus

The university Locus operating system [Popek *et al.* 1981, Walker *et al.* 1983] provides volumes and allows selective replication within the volume. However, the approach taken toward replication is not strictly optimistic. While Locus allows concurrent updates across partitions, the mechanism used to merge partitions is complex and does not scale, because it always attempts to maintain a perfect picture of partition membership. Locus can not, therefore, support mobility.

10.1.6 Ficus

Ficus [Guy *et al.* 1990a, Page *et al.* 1991] is one of the intellectual and physical predecessors of Roam, and Roam therefore shares many of the same characteristics as Ficus. Both are based on a peer model, although Roam is based on the Ward Model which scales vastly better than the traditional peer model implemented in Ficus. Both provide selective replication control. While each maintains consistency with a periodic reconciliation process, Ficus additionally uses a best-effort, real-time propagation of updates. Update propagation helps maintain stronger consistency in well-connected environments, but has the disadvantage that it reduces the performance of the native file system during normal operation.

Ficus is aimed at a distributed Internet environment, and works well within its intended target. However, it is an unsuitable system for mobile use. It suffers from scaling problems, and its garbage collection algorithm does not remove file data until a two-phase algorithm has executed between all replicas. Although it does provide the strong *no lost updates* semantics, we argue that these semantics are both unnecessary and too costly for mobile use.

Ficus is tightly integrated in the file system, and as such porting between systems is not straightforward. In contrast, Roam is entirely at the user level, and therefore trivial to port between different Unix versions. Additionally, since Roam is designed in a platform-independent manner [Salomone 1998], it should not be difficult to port it to non-Unix systems.

Ficus also provides transparent remote access, a feature that Roam does not currently offer. Objects that are not stored locally as a result of selective replication are transparently accessed from other replicas if a network connection exists; if not, a special error code is returned to the process. Roam does not provide this type of transparent remote access for the reasons discussed in Chapter 6. Additionally, Ficus dynamically and transparently mounts whole volumes from remote replicas into the local namespace. Transparent remote access is an area of important future work for Roam. Ficus offers it by being tightly integrated in the kernel, which provides a functionality versus performance tradeoff. Ideally, Roam would provide the same functionality, but in a lightweight manner.

10.1.7 Rumor

Rumor [Reiher *et al.* 1996] is the direct predecessor of Roam; in fact, much of Roam's implementation is directly based on modified Rumor code. Rumor was designed to be a direct adaptation of Ficus, only at the user level instead of integrated in the kernel. As such, it shares many of the same characteristics and problems as Ficus. It is based on the traditional peer model, and relies upon periodic reconciliation to maintain consistency. Rumor provides selective replication control, although only as a result of the Roam development. Rumor's chief problem is scaling. It shares the same scaling and garbage collection problems as Ficus, described above.

However, Rumor was the first optimistic peer replication system to operate completely at the user level, and partly for that reason pioneered peer replication for mobile use. It is a portable system that can easily be installed on different Unix platforms. Additionally, its *platform-independent* design [Salomone 1998] makes porting Rumor to other operating systems straightforward and easy. From one vantage point, Roam could be seen as the next generation of Rumor, both in its implementation and its underlying algorithms.

10.1.8 Harp

The HARP file system[Liskov *et al.* 1991] implements replication for a UNIX client-server environment using a primary-copy concurrency control mechanism. HARP achieves high performance and reliability by combining write-behind logging techniques with an uninterruptible power supply that allows logs to be forced to non-volatile storage after a power failure. However, being based on a conservative concurrency protocol and a client-server model makes it unattractive for mobile environments. Furthermore, the notion of using uninterruptible power supplies is completely unsuitable for mobile computers that are often untethered.

10.1.9 Tait and Duchamp

Tait and Duchamp designed an algorithm for replica management in mobile file systems [Tait *et al.* 1991] based on the client-server model. They allow the servers to be geographically separated, and allow clients to dynamic change their primary server. Their "dynamic client-server" model is therefore similar to the Ward Model, but with some substantial differences. While they do cluster replicas and allow a client to change its cluster, the algorithms to do so are more complex. Additionally, their system does nothing to solve the client-to-client interaction problem, as clients still cannot directly synchronize with other clients.

10.1.10 Lotus Notes

LOTUS NOTES [Corporation 1989] provides a database-oriented filing environment with a built-in replication service. Files are stored as forms in databases, and entire databases can be replicated.

NOTES databases provide a revision control service that is closely related to replication. Each "note" consists of a document and a series of responses to it. NOTES is aimed at an append-only environment, as many common modes simply append updates as additional responses. While it is possible to update the document itself, conflicting updates are resolved by making one an actual update to the document itself and one a response to the document. It is therefore not possible, in general, to guarantee that a given update actually updates the document itself.

LOTUS NOTES relies upon peer-to-peer reconciliation to periodically synchronize the databases. At synchronization time, new documents and responses are propagated. Additionally, LOTUS NOTES incorporates flexible, rule-based selective replication. A new document may be excluded from replication due to size, creation or modification date, or other factors.

LOTUS NOTES provides a simple and useful replication service. However, it is completely separate from any file system and requires a great commitment in infrastructure. It works well with documents that lend themselves to append-only usage, such as logs, but its universality is quite limited.

10.1.11 rdist

RDIST [Cooper 1992] is a relatively simple replication package based on the master-slave model. One replica is designated the master and all others are slaves. The synchronization protocol makes the slaves identical to the master, regardless of the file system activity that has occurred at the slaves. Therefore, updates can only be reliably performed at the master; those performed at the slave are "un-done" during synchronization. RDIST is useful in well-connected environments for maintaining mirrored copies of files, but was not intended as a fully functional replication service; it is hardly surprising that RDIST is not applicable to a mobile scenario with multiple sources of updates.

Advanced RDIST users often bypass the system's limitations with a "revolving master" scheme, in which they manually keep track of the site with the most recent updates, and designate that site as the current master. Doing so can be a potentially harmful mode of operation, since data loss results if the most recent master is forgotten or mislabeled. Additionally, such a model forces the user to participate in the replication process more than should be necessary.

10.2 Garbage Collection

Every optimistically replicated file system must have some method of resolving the create/delete ambiguity and performing garbage collection. ROAM maintains records of the deleted objects and uses a fully distributed, coordinator-free algorithm to guarantee that everyone learns of the deletion and removes the record. The garbage collection algorithm (Chapter 7) operates semi-independently between wards, reclaiming user data at the optimal speed. Additionally, the garbage collection algorithm is proven correct in all hypothetical scenarios of dynamic naming and long-term communication delays. ROAM's solution is not the only one, however.

CODA and DECEIT both have much simpler garbage collection algorithms. However, their algorithms are not appropriate in mobile environments. DECEIT's solution is not guaranteed to operate correctly in environments with common network partitions [Siegel 1992]. CODA's solution (at the servers) uses logging of all directory operations and a log wrap-around technique when the log becomes full [Kumar 1994]. The log assists synchronization by removing the create/delete ambiguity: since the log contains complete update histories, the logs can be directly compared, and the correct actions can be determined. Both systems preserve extra information; ROAM stores it in an object record, while CODA stores it in a common log file. The systems differ, however, in how the extra information is removed. CODA uses the simpler approach of log wrap-around to reclaim space in the log. When the log reaches its fixed size, existing entries at the beginning are overwritten. Logs are cleared as a result of successful synchronization (i.e., no conflicts).

The viability of CODA's solution depends entirely on how often the logs become full. Log wrap-around results from one of two scenarios: a long disconnection time or a plethora of file system activity. The actual length of time, or the actual quantity of file system activity, required to generate log wrap-around depends on the physical size of the log. Whenever log wrap-around does occur, however, it causes great headaches for the user. Each directory with an overwritten entry in the log is marked in update/update conflict and must be manually resolved, requiring user involvement and temporarily barring access to the objects underneath the directory.

CODA has not yet experienced many problems with its log wrap-around approach [Ebling 1997]; however, its servers are tightly integrated and typically well-connected. In fact, the only time CODA has experienced problems with log wrap-around is when a server died and it took a few days to begin functioning again [Ebling 1997]. Should the same strategy be applied in a mobile environment where disconnections are commonplace, it seems that the log wrap-around strategy would result in a large number of unnecessary conflicts, causing unneeded problems for mobile users.

FICUS and RUMOR both use a garbage collection algorithm similar to ROAM's; in fact, ROAM's algorithm is the intellectual descendent of the one developed for FICUS [Guy *et al.* 1993]. FICUS and RUMOR provide the stronger *no lost updates* semantics, while ROAM provides the weaker *time moves forward* semantics. The two semantics are related in that *no lost updates* is the result of applying *time moves forward* to all names for a given object, past and future. We argue in Chapter 7 that the strong guarantees made by *no lost updates* are unnecessary and potentially incorrect in the face of selective replication. We additionally argue that that the ability of the *time moves forward* semantics to immediately reclaim file data is more important than any potential benefit gained by the *no lost updates* semantics.

ROAM's garbage collection algorithms are related in spirit to Goel's work on *view consistency* for optimistic replication systems [Goel 1996]. Goel provided higher-level constraints on the replica selection mechanism to guarantee that users never accessed older data than they had previously seen at some other replica, which is similar to the guarantees ROAM provides on the garbage collection of objects. In fact, the term *time moves forward* originated in Goel's work.

The two-phase algorithm for record reclamation and create/delete disambiguation is adopted from Guy [Guy *et al.* 1993] and is related to the general topic of forming consensus in distributed environments. Heddaya et al. [Heddaya *et al.* 1989] use a two-phase gossip protocol to manage distributed event histories of updates to object replicas. However, their solution does not address the problem of completely forgetting that a history exists; it only addresses forgetting items in the history.

Wiseman's survey [Wiseman 1988] of distributed garbage collection methods includes several techniques based on reference counting, but none are designed for use on replicated objects, and none are directly applicable to imperfectly connected networks.

Finally, the issue of garbage collection is related to the general "gossip" problem, in which each node in a graph must communicate a unique datum to every other node in the graph. A variety of researchers have done work in this area [Hedetniemi *et al.* 1988].

10.3 Version Vector Management

Version vectors are basically an implementation of Lamport clocks [Lamport 1978]. Since Parker et al.'s original work [Parker *et al.* 1983], very little research has been done on the version vector. Ramarao attempted to disprove the correctness of the version vector [Ramarao 1987], but his refutation stemmed from a misunderstanding, and the version vector has gained global acceptance as the staple behind optimistic replication. Some have commented on the poor scalability of the version vector, but no one, until now, has improved upon it.

Our version vector compression algorithm forms consensus on the value of a given element and atomically removes it from the vector. As a consensus algorithm, it is closely related to Guy's garbage collection algorithm [Guy *et al.* 1993] and two-phase commit protocols [Gray 1978, Lampson *et al.* 1979]. However, unlike the earlier algorithms, our algorithm can afford to be one-phase, rather than requiring two. Although we require two phases to first form consensus on the set of elements being compressed, actual consensus on the b-value can be performed in one phase. (The alternative approach to simultaneous compression of multiple elements, discussed in Chapter 8, only requires one phase total.) The generating replica, the replica whose element is being compressed, cannot alter its "vote"; therefore, one phase is enough for consensus. Additionally, our algorithm is framed in a hierarchical model, and we allow mobility between groups in the hierarchy. Previous consensus algorithms did not address these notions.

Our compression algorithm must also perform consensus "in place." We cannot restrict or block updates during the algorithm, even if they increment the element being compressed. Many consensus algorithms [Guy *et al.* 1993, Lamport 1989, Oki *et al.* 1988, Dwork *et al.* 1988] address the in-place issue by being resilient to changes only during certain phases of the algorithm, and restricting when participants can change their value. We solve the in-place issue with the simple approach of using a spare element, thereby avoiding the complexity of handling value-changes while never restricting updates during any phase of the algorithm.

A number of algorithms consider consensus-forming in the face of machine failures [Skeen 1981, Lamport 1989, Oki *et al.* 1988, Dwork *et al.* 1988]. Our version vector compression algorithm addresses the issue of failed machines in a different manner. Rather than designing a more intricate and resilient protocol, we rely on Roam and reconciliation to propagate the information regarding which machines have failed (discussed in Chapter 4, Section 4.5). Version vector compression uses the underlying system knowledge with regard to failed machines, and therefore does not require the knowledge to be bundled into the algorithm.

Nevertheless, there are some similarities with the above, more robust algorithms. Specifically, the idea of algorithm restarts and multiple "rounds" is similar to the Paxos algorithm [Lamport 1989].

10.4 Selective Replication

Many replication systems based on the client-server model provide file granularity control over which objects should be physically stored at the clients. Coda and Little Work are two such examples. They recognize the need for fine-grain control at the file level. However, building such control within the client-server model is significantly easier and provides less functionality than Roam's selective replication in the peer model.

Alternatively, some have argued specifically against file-level granularity in the replication service, as in the case of the Echo distributed file system [Hisgen *et al.* 1989, Hisgen *et al.* 1990]. The authors argue the necessity of a large-granularity grouping mechanism for performance reasons, and specifically argue against file-level granularity. However, Echo utilizes quorum-based replication techniques with a primary synchronization site, and needs to actively and periodically perform elections to re-establish the primary for each replication unit. In our optimistic context, we do not have such real-time performance constraints, and can therefore afford to provide the more flexible selective replication controls.

The university Locus operating system was perhaps the first service to provide selective replication in a peer context. However, Locus did not scale well.

Both Ficus and Rumor support selective replication (Rumor only as a stage in the development of Roam). The Ficus solution was an initial implementation, and noticeable improvements in the efficiency of the algorithms were made when implementing selective replication in Roam, especially in the garbage collection of selectively replicated files.

10.5 Scaling

Many systems have been designed to provide good scalability, and clustering participants into groups and forming a hierarchy is a typical approach. The client-server model uses such an approach, since servers can themselves be grouped together and act as clients to a meta-server. The Internet Protocol design is based on a hierarchical solution, as are a number of other designs aimed at large environments and good scalability. Roam borrows the idea of grouping, electing one participant from the group to speak for the entire group, and constructing hierarchies from a large body of existing research.

Additionally, and perhaps more importantly, most if not all previous hierarchical solutions do not allow movement between clusters, especially clusters from different "subtrees" in the hierarchy. The Ward Model is specifically designed with an inherent peer ability, allowing motion from any ward to any other ward, regardless of their physical location within the hierarchical structure.

Chapter 11

Future Work

During any project of this magnitude, a number of ideas, possibilities, and extensions arise that are not practical to implement within the available time frame. Here we outline a number of improvements and possible future additions to Roam that one might desire.

11.1 Ward Placement

The Ward Model provides an architecture capable of grouping replicas into wards and allowing motion between wards. It does not, however, address the issue of optimal ward placement with respect to parameters like the degree of file sharing and dynamic changes in network connectivity and quality. Higher-level daemons should be constructed that analyze the layout of ward membership and dynamically use the Ward Model's controls to reconfigure the set of wards.

One potentially very important area of research concerns the automatic detection of hot-spots: the set of replicas that are actively updating a specific object or collection of objects (Chapter 8). Automatically detecting hot spots would help the version vector compression algorithms because we could more easily differentiate hot from cold replicas, and would therefore easily identify the candidates for compression. Equally importantly, however, is the potential interaction between hot-spots and ward placement. An interesting idea would be to group all replicas in the current hot-spot in one ward. Since the hot replicas are the ones actively updating a given object, one would ideally want to maintain the tightest degree of synchronization and consistency between them. It is unclear whether or not the hot-spot approach would lead to improved reconciliation topologies (i.e., if consistency could be maintained more efficiently). Additionally, there are added complexities, in that the hot-spot is a file granularity notion, and the ward is a volume granularity construct. Nevertheless, it appears to be an interesting direction for future thought and possible performance improvements.

11.2 Handling Machine Failures

As mentioned in Chapter 4, Section 4.6, the fully general realization of the Ward Model requires more robust, efficient, and aggressive ward master re-election algorithms. Currently, the failure of a ward master is detected lazily; a more aggressive mechanism would be required.

One possibility borrows an idea from Deceit. Deceit allows users to specify that a specific number of replicas should always exist of a given object. We could use a similar procedure to replicate ward masters, creating more replicas at higher levels where redundancy is more important. As mentioned in Chapter 4, the Ward Model fully supports multiple ward masters.

A second idea uses a variant of network heartbeats [Andrews 1991, Aguilera *et al.* 1997]. If the ward members maintained better contact with the ward master, they could more quickly detect ward master failures (or the creation of network partitions that separate them from the ward master). Ward master re-election could therefore occur more quickly.

Additionally, the procedure used to handle the failure of normal ward members requires human intervention to initiate the decision process regarding when a machine has failed. Chapter 4 describes a fully automated solution to the problem, but it is not currently implemented.

11.3 Remote Access

Like Ficus, Roam requires a transparent remote access capability that, when network-connected, provides access both to non-local volumes and to individual files that are not locally stored as a result of selective replication. The latter is probably more important than the former, at least with respect to the transparent requirement. While the "hooks" exist in Roam to locate remote replicas storing the objects, the mechanism to actually redirect access to a remote replica is missing. Selective replication should not break name transparency. A lightweight, user-level mechanism similar to that implemented in the UFO file system [Alexandrov *et al.* 1997] should be included. Roam's functionality would be greatly improved.

11.4 Real-time Update Propagation

Another feature of Ficus that should be implemented in Roam is a best-effort, real-time propagation of updates. Reconciliation should still be the only process that reliably enforces consistency, because such a constraint simplifies the design of update propagation and allows us to ignore complicated issues like queuing and retransmitting messages. Nevertheless, update propagation would dramatically increase consistency in well-connected environments.

Update propagation is difficult to add in Rumor because of its architecture. However, Roam's new server architecture makes the implementation of update propagation easy and straightforward. A site that generates an update and wants to inform others can simply use its local `in-out server` to send *GetData* messages to the other interested replicas. The other replicas are identified using local information indicating where physical replicas are stored. The remote replicas, upon receipt of the *GetData* message, can choose to either ignore it or fetch data by replying with a *SendData* message. The initial message is therefore small and cheap to send; only if the remote site is accessible and willing does it request file data. The ward master could even relay the update propagation notices to other wards. Update propagation has the potential to greatly increase consistency in connected environments.

11.5 Interaction Between Wards and Mobile-IP

Recall Figure 2.2 in Chapter 2 where we argued that a peer model was required, especially in the case of portables traveling together across the country. Designing a replication system capable of supporting portable workgroups and enabling direct, any-to-any communication is certainly an integral part of the complete solution for mobile computing. However, it is not the entire solution. The portables that traveled across the country are presumably communicating via some mobile-IP protocol. Figure 11.1 illustrates what happens when we include the communication channels required by mobile-IP; the figure is almost identical to the original client-server paradigm in Figure 2.1. Clearly, for the Ward Model to truly support direct, efficient, any-to-any communication, the interaction with mobile-IP solutions must be analyzed and dealt with.

11.6 Integration With Truffles

The Ward Model could be better integrated with a security model like Truffles [Reiher *et al.* 1993a]. Since the `wardd` and the `in-out server` control all accesses into and off of the machine, we have the ability

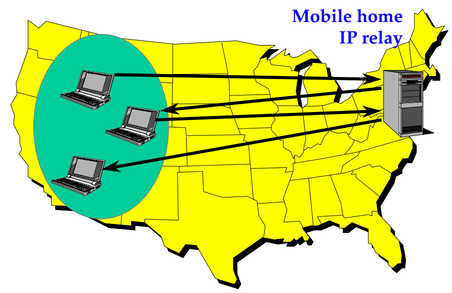

Figure 11.1: The poor interaction between a replication system supporting any-to-any communication and the underlying mobile-IP communication protocol. Even though the ward allows direct communication between machines, the mobile-IP protocol still forces communication through the distant server. Compare this picture to that of Figure 2.1 and Figure 2.2.

to institute strong security policies in only one place. We could control what replicas are allowed to move into specific wards, and could use the ward as an internal firewall for security purposes.

11.7 Prioritized Reconciliation

Mobile users have limited bandwidth, and often pay a per-byte or per-minute connection cost. Mobile users therefore want to be careful how much information they upload and download when poorly connected. For instance, someone performing code development requires the updates to the source files, but most likely propagation of the new version of PowerPoint can wait until the user is better connected—perhaps until he or she returns to the office in a few days and connects via Ethernet.

Roam does not take into account the connection quality when reconciling, unlike Coda which has implemented various optimizations for weak-connectivity environments [Mummert *et al.* 1995]. Reconciliation could, for example, prioritize the user's files, and only transmit a subset of all available updates depending on the connectivity. Automated tools such as Seer [Kuenning *et al.* 1997], which have knowledge of the files in active use, could automate the assignment of priorities.

11.8 Reconciliation Performance Improvements

Chapter 5 mentioned a list of techniques that would increase the performance of reconciliation. These techniques included building the filelists in parallel and building the filelists entirely "off-line."

11.9 Reconciliation Topologies

Concurrently with Roam's development, work is ongoing with respect to the effects of different reconciliation topologies on processor utilization, overall performance, message complexity, conflict rates, and other parameters [Wang *et al.* 1997]. When complete, those results should be incorporated, possibly altering Roam's reconciliation topologies.

11.10 Improved Selective Replication Controls

As mentioned in Chapter 6, the most desirable user interface to the selective replication controls would be a general-purpose system that allowed users to specify where replicas should be placed, either by file type or by pattern matching on the file name. The replication masks and dynamic controls currently implemented work well, but are not the optimal solution.

Chapter 12

Conclusions

This dissertation has designed and implemented solutions for a number of significant problems in large scale replication, and produced a scalable replication system for mobile environments. In this chapter we discuss review the highlights and important contributions of the project.

12.1 Summary of the Problem

With the arrival of machines capable of supporting truly mobile computing came users wanting to access and update their data while mobile. If the user has more than one machine, for instance a laptop and a desktop, or if the data must be shared between multiple users, then the data must be replicated.

Unfortunately, the existing replication systems are not mobile-compliant. Designed for stationary environments, they do not provide users with the abilities they require when mobile. Mobile users have some very basic requirements:

- Support for direct, inexpensive any-to-any communication
- Support for large replication factors
- Fine-grain (file-level) replication control
- Mobile "anywhere, anytime" functionality
- A "pay as you go" framework, where the up-front cost of temporary motion is very small
- A mobility solution that can guarantee equivalent synchronization performance regardless of physical location

Mobility represents a fundamental paradigm shift in the way people use and interact with their computers and each other. As such, it is not surprising that a new replication system must be designed to provide the new set of requirements.

12.2 Roam's Solution

ROAM's solution is based on the Ward Model, a new replication model specifically designed for mobility. As a hybrid of traditional peer and client-server solutions, the Ward Model provides the advantages of both, while minimizing the disadvantages.

Although the Ward Model clusters replicas for scalability and efficiency, it allows direct any-to-any communication between any set of replicas in the system through the use of two different mobility algorithms.

The use of different algorithms for different scenarios provides users with a "pay as you go" framework, meaning that the cost of mobility is never greater than the benefits gained.

Additionally, ROAM uses a series of "mobile-friendly" distributed algorithms and mechanisms, such as selective replication control, optimal garbage collection of user data, and dynamic version vector management, to minimize the costs associated with replication and achieve good scalability.

The ROAM solution is based on the locality premise that the dissemination and propagation of information is generally cheaper, faster, and easier between local partners rather than remote ones. Additionally, we argue that the locality premise matches the user's expected and desired behavior, from both an economic and a functionality perspective.

ROAM is also based on an optimistic assumption. First, we recognize that in a mobile environment we *cannot* guarantee continual, good quality network connections, so conservative algorithms and protocols are simply not viable. Second, we argue that a lazy approach to replication, where updates to user data and replication meta-data are lazily propagated throughout the system, works well. Studies of file systems indicate that concurrent updates are rare, and many can be automatically healed without user involvement. Furthermore, we build into ROAM's algorithms the ability to automatically handle and update outdated meta-data as part of the normal synchronization of user data. ROAM makes extensive use of the optimistic approach in its management of the system.

12.3 Contributions of the Dissertation

The contribution of the dissertation is fivefold. First and foremost, we designed the Ward Model, a replication architecture capable of supporting widespread mobile computing. It is a new hybrid model of traditional client-server and peer systems, and not only scales well but enables direct any-to-any communication between any two participants. Second, we implemented the Ward Model in a real working system called ROAM and used it in real environments. It is available for others to use in mobile scenarios and as a base for future mobility research Third, we provided algorithms for the selective control of replicated data, allowing the files to be individually replicated independently of the volume that contains them. Fourth, we designed new garbage collection semantics and a new garbage collection algorithm that reclaims user data in an optimal fashion and uses a proven-correct, fully distributed, coordinator-free algorithm to resolve the create/delete ambiguity. Finally, we devised new methods of managing the version vector, the main data structure behind all optimistic replication, making the version vector a much more scalable construct.

12.4 Final Comments

Replication is not a new problem, and has many well-known and well-understood solutions. Unfortunately, the formulation of these solutions incorporated a stationary mindset. True mobile computing has only recently become affordable and feasible; previous solutions were designed without considering the ramifications and possible affects physical motion would place on the replication system. ROAM builds upon the previous work by defining a new replication model and designing a replication system capable of supporting real mobile users.

We believe the concepts and ideas behind ROAM to be more widely applicable than just for file replication. The Ward Model may apply equally well in other large-scale environments that share common characteristics, such as in the mobile IP problem, network routing protocols, and Internet security. The algorithms behind version vector compression also apply to more general distributed consensus-forming problems. The selective-replication protocols and algorithms apply not just to the replication of files, but to more general problems concerned with managing individual objects within a large container.

ROAM is a successful piece of research, based not just on the performance analysis but also on real-world experience and use. It is our hope that ROAM paves the way for real mobile use and future mobile computing research; simultaneously, we would like the underlying ideas and concepts to bear fruit and become used in other areas of computer science.

Trademarks

UNIX is a trademark of X/Open Company, Ltd. NFS is a trademark of Sun Microsystems. WINDOWS 95 is a registered trademark of Microsoft Corporation. POWERPOINT is a trademark of Microsoft Corporation. WINDOWSNT is a trademark of Microsoft Corporation. LOTUS NOTES is a trademark of Lotus Development Corporation. WaveLAN is a trademark of AT&T Global Information Solutions Company.

References

[Aguilera *et al.* 1997] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. "Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication." Technical Report TR97-1631, Cornell University, Computer Science, May 30, 1997.

[Alexandrov *et al.* 1997] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. "Extending the Operating System at the User-Level: the Ufo Global File System." In *USENIX Conference Proceedings*, pp. 77–90, Anaheim, California, January 1997. USENIX.

[Alonso *et al.* 1989] Rafael Alonso, Daniel Barbará, and Luis L. Cova. "A File Storage Implementation for Very Large Distributed Systems." In *Proceedings of the Second Workshop on Workstation Operating Systems*. IEEE Computer Society Press, September 1989.

[Anderson *et al.* 1995] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. "Serverless Network File Systems." In *Proceedings of the 15th Symposium on Operating Systems Principles*, pp. 109–126, Copper Mountain Resort, Colorado, December 1995. ACM.

[Andrews 1991] Gregory R. Andrews. "Paradigms for Process Interaction in Distributed Programs." *ACM Computing Surveys*, **23**(1):49–90, March 1991.

[Awerbuch 1987] Baruch Awerbuch. "Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election and Related Problems (Detailed Summary)." In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pp. 230–240, New York City, 25–27 May 1987.

[Bagrodia *et al.* 1995] Rajive Bagrodia, Wesley W. Chu, Leonard Kleinrock, and Gerald Popek. "Vision, Issues, and Architecture for Nomadic Computing." *IEEE Personal Communications Magazine*, **2**(6):14–27, December 1995.

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

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

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

[Ceri *et al.* 1992] Stefano Ceri, Maurice A. W. Houtsma, Arthur M. Keller, and Pierangela Samarati. "The Case for Independent Updates." In *Proceedings of the Second Workshop on Management of Replicated Data*, pp. 17–19. IEEE, November 1992.

[Chandra *et al.* 1991] Tushar Deepak Chandra and Sam Toueg. "Unreliable Failure Detectors for Asynchronous Systems." In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, pp. 325–340. ACM Press, August 1991.

[Cooper 1992] M.A. Cooper. "Overhauling Rdist for the '90s." In *Proceedings of the Sixth USENIX Systems Administration Conference (LISA VI)*, pp. 175–188. Berkeley, California, October 1992.

[Corporation 1989] Lotus Development Corporation. "Lotus Notes: Essential Software for Group Communications." Lotus Notes Technical Series Vol. 1, December 6 1989.

[Davčev *et al.* 1985] Dančo Davčev and Walter A. Burkhard. "Consistency and Recovery Control for Replicated Files." In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pp. 87–96. ACM, December 1985.

[Davidson 1984] Susan B. Davidson. "Optimism and Consistency in Partitioned Distributed Database Systems." *ACM Transactions on Database Systems*, **9**(3):456–481, September 1984.

[Davidson *et al.* 1985] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. "Consistency in Partitioned Networks." *ACM Computing Surveys*, **17**(3):341–370, September 1985.

[Demers *et al.* 1994] Alan Demers, Karin Petersen, Mike Spreitzer, Douglas Terry, Marvin Theimer, and Brent Welch. "The Bayou Architecture: Support for Data Sharing among Mobile Users." In *Proceedings of the Workshop on Mobile Computing Systems and Applications1994*, Santa Cruz, CA, December 1994.

[Dwork *et al.* 1988] Cynthia Dwork, Nancy Lynch, and L. Stockmeyer. "Consensus in the Presence of Partial Synchrony." *Journal of the ACM*, **35**(2), April 1988.

[Ebling 1997] Maria Ebling, 1997. Personal communication with Geoff Kuenning and Maria Ebling, 11 July.

[Fischer 1983] Michael J. Fischer. "The Consensus Problem in Unreliable Distributed Systems (a brief survey)." Technical Report YALEU/DSC/RR-273, Yale University, June 1983.

[Fischer *et al.* 1982] 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.

[Floyd 1986a] Rick Floyd. "Directory Reference Patterns in a UNIX Environment." Technical Report TR-179, University of Rochester, August 1986.

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

[Gafni 1985] Eli Gafni. "Improvements in the time complexity of two message-optimal election algorithms." In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, August 1985.

[Garcia-Molina 1982] H. Garcia-Molina. "Elections in Distributed Computer Systems." *IEEE Transactions on Computers*, **C-31**:48–59, 1982.

[George *et al.* 1981] A. George and W.H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981. Prentice-Hall Series in Computational Mathematics.

[Goel 1996] Ashvin Goel. *"View Consistency for Optimistic Replication."* Master's thesis, University of California, Los Angeles, February 1996. Available as UCLA technical report CSD-960011.

[Gray 1978] J. N. Gray. "Notes on Database Operating Systems." In *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[Gray *et al.* 1996] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. "The Dangers of Replication and a Solution." In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pp. 173–182. ACM, June 1996.

[Guerraoui *et al.* 1996] Rachid Guerraoui, Rui Oliveira, and André Schiper. "Atomic Updates of Replicated Data." *Lecture Notes in Computer Science*, **1150**:365–381, October 1996.

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

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

[Guy *et al.* 1990a] 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.

[Guy *et al.* 1990b] Richard G. Guy, Thomas W. Page, Jr., John S. Heidemann, and Gerald J. Popek. "Name Transparency in Very Large Scale Distributed File Systems." In *Second IEEE Workshop on Experimental Distributed Systems*, pp. 20–25. University of California, Los Angeles, IEEE, October 1990.

[Guy *et al.* 1990c] Richard G. Guy and Gerald J. Popek. "Reconciling Partially Replicated Name Spaces." Technical Report CSD-900010, University of California, Los Angeles, April 1990.

[Guy *et al.* 1993] 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.

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

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

[Heidemann *et al.* 1992] 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. University of California, Los Angeles, IEEE, November 1992.

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

[Herlihy 1986] Maurice Herlihy. "A Quorum-Consensus Replication Method for Abstract Data Types." *ACM Transactions on Computer Systems*, **4**(1):32–53, February 1986.

[Herring 1996] Thomas A. Herring. "The Global Positioning System." *Scientific American*, **274**(2):44–50, February 1996.

[Hisgen *et al.* 1989] Andy Hisgen, Andrew Birrell, Timothy Mann, Michael Schroeder, and Garret Swart. "Availability and Consistency Tradeoffs in the Echo Distributed File System." In *Proceedings of the Second Workshop on Workstation Operating Systems*. IEEE Computer Society Press, September 1989.

[Hisgen *et al.* 1990] Andy Hisgen, Andrew Birrell, Chuck Jerian, Timothy Mann, Michael Schroeder, and Garret Swart. "Granularity and Semantic Level of Replication in the Echo Distributed File System." In *Proceedings of the Workshop on Management of Replicated Data*, pp. 2–4. IEEE, November 1990.

[Honeyman *et al.* 1992] Peter Honeyman, Larry Huston, Jim Rees, and Dave Bachmann. "The Little Work Project." In *Proceedings of the Third Workshop on Workstation Operating Systems*, pp. 11–14. IEEE, April 1992.

[Howard *et al.* 1988] 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.

[Itai *et al.* 1990] Alon Itai, Shay Kutten, Yaron Wolfstahl, and Shmuel Zaks. "Optimal Distributed t-Resilient Election in Complete Networks." *IEEE Transactions on Software Engineering*, **16**(4):415–420, April 1990.

[Jain 1991] Raj Jain. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, Inc., 1991.

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

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

[Kleinrock 1997a] Leonard Kleinrock. "Nomadicity." Presentation at the GloMo PI Meeting (February 4) at the University of California, Los Angeles, 1997.

[Kleinrock 1997b] Leonard Kleinrock. "Progress in Nomadic Computing." Presentation at the UCLA Computer Science Department Research Review (April 10) at the University of California, Los Angeles, 1997.

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

[Kuenning 1997] Geoffrey Houston Kuenning. *Seer: Predictive File Hoarding for Disconnected Mobile Operation.* PhD thesis, University of California, Los Angeles, Los Angeles, CA, May 1997. Also available as UCLA CSD Technical Report UCLA-CSD-970015.

[Kuenning *et al.* 1994] 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.

[Kuenning *et al.* 1997] Geoffrey H. Kuenning and Gerald J. Popek. "Automated Hoarding for Mobile Computers." In *Proceedings of the 16th Symposium on Operating Systems Principles*, St. Malo, France, October 1997. ACM.

[Kumar 1994] Puneet Kumar. *Mitigating the Effects of Optimistic Replication in a Distributed File System.* Ph.D. dissertation, Carnegie Mellon University, December 1994. Available as technical report CMU-CS-94-215.

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

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

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

[Lamport 1987] Leslie Lamport, 1987. Email was sent by Leslie Lamport at 12:23:29 PDT on May 28, 1987, message ID 8705281923.AA09105@jumbo.dec.com.

[Lamport 1989] Leslie Lamport. "The Part-Time Parliament." Technical Report 49, DEC Systems Research Center, September 1989.

[Lampson *et al.* 1979] Butler W. Lampson and Howard E. Sturgis. "Crash recovery in a distributed data storage system." Technical report, XEROX Palo Alto Research Center, April 1979.

[Liskov *et al.* 1991] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. "Replication in the Harp File System." In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pp. 226–238. ACM, October 1991.

[Lynch 1996] Nancy Lynch. *Distributed Algorithms*. Morgane Kaufmann Publishers, 1996.

[Marchetti-Spaccamela 1987] A. Marchetti-Spaccamela. "New protocols for the election of a leader in a ring." *Theoretical Computer Science*, **54**(1):53–64, September 1987.

[Mummert *et al.* 1994] Lily B. Mummert and Mahadev Satyanarayanan. "Large Granularity Cache Coherence for Intermittent Connectivity." In *USENIX Conference Proceedings*, pp. 279–289. USENIX, June 1994.

[Mummert *et al.* 1995] Lily B. Mummert, Maria R. Ebling, and Mahadev Satyanarayanan. "Exploiting Weak Connectivity for Mobile File Access." In *Proceedings of the 15th Symposium on Operating Systems Principles*, pp. 143–155, Copper Mountain Resort, Colorado, December 1995. ACM.

[Nance 1995] B. Nance. "File transfer on steroids." BYTE Magazine, February 1995.

[Oki *et al.* 1988] Brian M. Oki and Barbara Liskov. "Viewstamped Replication: A General Primary Copy Method to Support Highly-Available Distributed Systems." In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pp. 8–17, August 1988.

[Ousterhout *et al.* 1989] John Ousterhout and Fred Douglis. "Beating the I/O Bottleneck: A Case for Log-Structured File Systems." *Operating Systems Review*, **23**(1):11–28, January 1989.

[Page *et al.* 1991] Thomas W. Page, Jr., Richard G. Guy, Gerald J. Popek, John S. Heidemann, Wai Mak, and Dieter Rothmeier. "Management of Replicated Volume Location Data in the Ficus Replicated File System." In *USENIX Conference Proceedings*, pp. 17–29. University of California, Los Angeles, USENIX, June 1991.

[Pâris 1989] Jehan-François Pâris. "Voting with Bystanders." In *Proceedings of the Ninth International Conference on Distributed Computing Systems*, pp. 394–401, May 1989.

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

[Popek 1997] Gerald J. Popek, 1997. Personal communication by email, 26 February.

[Popek *et al.* 1981] 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.

[Prusker *et al.* 1990] Francis J. Prusker and Edward P. Wobber. "The Siphon: Managing Distant Replicated Repositories." In *Proceedings of the Workshop on Management of Replicated Data*, pp. 44–47. IEEE, November 1990.

[Quéinnec *et al.* 1993] Philippe Quéinnec and Gérard Padiou. "Flight Plan Management in Distributed Air Traffic Control System." In *Proceedings of the International Symposium on Autonomous Decentralized Systems*, Kawasaki, Japan, March 1993.

[Ramarao 1987] K. V. S. Ramarao. "Comments on "Detection of Mutual Inconsistency in Distributed Systems"." *IEEE Transactions on Software Engineering*, **13**(6):759–760, June 1987.

[Ratner 1995] David Howard Ratner. *"Selective Replication: Fine-Grain Control of Replicated Files."* Master's thesis, University of California, Los Angeles, March 1995. Available as UCLA technical report CSD-950007.

[Ratner 1997] David Ratner. "Selective Replication Design and Implementation in Rumor." Rumor internal design document, February 1997.

[Ratner *et al.* 1996a] David Ratner, Gerald J. Popek, and Peter Reiher. "Peer Replication with Selective Control." Technical Report CSD-960031, University of California, Los Angeles, July 1996.

[Ratner *et al.* 1996b] David Ratner, Gerald J. Popek, and Peter Reiher. "The Ward Model: A Replication Architecture for Mobile Environments." Technical Report CSD-960045, University of California, Los Angeles, December 1996.

[Ratner *et al.* 1996c] David Ratner, Gerald J. Popek, and Peter Reiher. "The Ward Model: A Scalable Replication Architecture for Mobility." In *Workshop on Object Replication and Mobile Computing*, October 1996.

[Reiher *et al.* 1993a] P. Reiher, S. Crocker, J. Cook, T. Page, and G. Popek. "Truffles—Secure File Sharing With Minimal System Administrator Intervention." In *Proceedings of the 1993 World Conference On Tools and Techniques for System Administration*, April 1993.

[Reiher *et al.* 1993b] P. Reiher, T. Page, S. Crocker, J. Cook, and G. Popek. "Truffles—A Secure Service for Widespread File Sharing." In *Proceedings of the The Privacy and Security Research Group Workshop on Network and Distributed System Security*, February 1993.

[Reiher *et al.* 1994] 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. University of California, Los Angeles, USENIX, June 1994.

[Reiher *et al.* 1996] Peter Reiher, Jerry Popek, Michial Gunter, John Salomone, and David Ratner. "Peer-to-peer Reconciliation Based Replication for Mobile Computers." In *Proceedings of the ECOOP Workshop on Mobility and Replication*, July 1996.

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

[Salomone 1998] John Raymond Salomone. *"Portable Rumor: A Platform Independent File Replication Package."* Master's thesis, University of California, Los Angeles, June 1998. To be finished. Expected completion: Spring 1998.

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

[Satyanarayanan 1992] Mahadev Satyanarayanan. "The Influence of Scale on Distributed File System Design." *IEEE Transactions on Software Engineering*, **SE-18**(1):1–8, January 1992.

[Satyanarayanan *et al.* 1985] Mahadev Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector, and Michael J. West. "The ITC Distributed File System: Principles and Design." In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pp. 35–50. ACM, December 1985.

[Satyanarayanan *et al.* 1990] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. "Coda: A Highly Available File System for a Distributed Workstation Environment." *IEEE Transactions on Computers*, **39**(4):447–459, April 1990.

[Satyanarayanan *et al.* 1993] Mahadev Satyanarayanan, James J. Kistler, Lily B. Mummert, Maria R. Ebling, Puneet Kumar, and Qi Lu. "Experience with Disconnected Operation in a Mobile Computing Environment." In *Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing*, pp. 11–28, Cambridge, MA, August 1993. USENIX.

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

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

[Singh 1994] Gurdip Singh. "Leader Election in the Presence of Link Failures." In *Symposium on Principles of Distributed Computing (PODC '94)*, pp. 375–375, New York, USA, August 1994. ACM Press.

[Skeen 1981] Dale Skeen. "Nonblocking Commit Protocols." In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 133–142, May 1981.

[Smith 1981] Alan J. Smith. "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms." *IEEE Transactions on Software Engineering*, **7**(4), July 1981.

[Stonebraker 1979] Michael Stonebraker. "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES." *IEEE Transactions on Software Engineering*, **5**(3), May 1979.

[Tait *et al.* 1991] Carl D. Tait and Dan Duchamp. "Service Interface and Replica Consistency Algorithm for Mobile File System Clients." In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pp. 190–197, December 1991.

[Terry *et al.* 1994] 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 Distributed Information Systems*, pp. 140–149, September 1994.

[Terry *et al.* 1995] 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 15th Symposium on Operating Systems Principles*, pp. 172–183, Copper Mountain Resort, Colorado, December 1995. ACM.

[Thomas 1979] 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.

[Walker *et al.* 1983] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. "The LOCUS Distributed Operating System." In *Proceedings of the Ninth Symposium on Operating Systems Principles*, pp. 49–70. ACM, October 1983.

[Wang *et al.* 1997] An-I A. Wang, Peter L. Reiher, and Rajive Bagrodia. "A Simulation Framework for Evaluating Replicated Filing Environments." Technical Report CSD-970018, University of California, Los Angeles, June 1997.

[Want *et al.* 1995] Roy Want, Bill N. Schilit, Norman I. Adams, Rich Gold, Karin Petersen, David Goldberg, John R. Ellis, and Mark Weiser. "An Overview of the ParcTab Ubiquitous Computing Experiment." *IEEE Personal Communications Magazine*, **2**(6):28–43, December 1995.

[Welch *et al.* 1986] Brent Welch and John Ousterhout. "Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System." *Sixth International Conference on Distributed Computing Systems*, pp. 184–189, May 19-23, 1986.

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