FRIENDSHARE: A DECENTRALIZED, CONSISTENT STORAGE REPOSITORY FOR COLLABORATIVE FILE SHARING

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University

Frank Chiang

May 2008

(Jinyang Li) Principal Adviser

(Lakshminarayanan Subramanian)

Acknowledgements

First and foremost, I would like to thank my advisor, Jinyang Li, for her knowledge, advice, and unwavering support. I would also like to thank Nguyen Tran, whom I worked with on both the Friendstore and Friendshare projects. This thesis was made possible largely due to their efforts and guidance. I would also like to thank Lakshminarayanan Subramanian for being my second reader.

This thesis was also made possible by many fantastic peers, including Sujay Lele and Charles Reich, who helped in designing and coding the base platform for Friendshare. I want to thank all my friends who participated in the painful task of beta testing, including Vivek Bhattacharyya, Ned Campion, Alex Chik, William Holloway, Warren Koo Tze Mew, Gordon Kwan, Jill Lee, Chris Li, Rebecca Quan, and Paul Schetinin.

Finally, I want to thank my parents for their dedication, support, and love. I cannot imagine where I would be without their presence.

Contents

List of Figures

Chapter 1

Introduction

Users have long relied on the Internet to publish and share data with each other. Traditionally, a shared data repository is written and organized by only a single author. There are many examples for single-writer repositories. Blogs are generally managed by one user rather than a group of users. Flickr [11] allows individuals to upload, organize, and share their own photos.

With the rise of Web 2.0, data sharing has become more and more collaborative such that multiple writers can jointly write and organize the content in the repositories. This trend is evident in the emergence of online collaborations, such as Wikipedia [40] and Google Groups [13]. Wikipedia is a popular online encyclopedia that allows users to share their knowledge by modifying the encyclopedic entries. Google Groups is a repository where users can post messages and share files. As a concrete example, consider a group of friends wanting to share the photos that they take at gatherings. At each party, multiple friends take pictures with their own cameras, which yields multiple sets of photos for the same party. It would be useful if all the friends were able to collaboratively organize the photos in a shared photo repository. The state of the art approach to build a consistent and reliable shared repository is to rely on a central site managed by a third party. However, centralized solutions are undesirable sometimes because of privacy concerns and censorship, which are problems that can be alleviated by switching to decentralized solutions. This thesis contributes the design and implementation of a decentralized multiple-writer data repository that can support a variety of collaborative data sharing applications, such as repositories for sharing music and movies, bulletin boards, etc.

1.1 Why a decentralized data repository?

The current practice of data sharing is for users to upload data onto a central site, such as Wikipedia and Google. Centralized solutions have many benefits. First, a central site can replicate the data on a set of tightly-coupled machines in well-managed data centers to ensure that the data will be highly available. Second, it is relatively straightforward to maintain the consistency of replicated data among highly-available machines with low-latency network connections within a data center. Google's GFS [12] and Amazon's S3 [2] are well-known examples of centralized data storage services.

However, centralized storage also has many drawbacks. First, by storing data using a service offered by commercial companies, user data is subject to the company's privacy policies. For example, Google's Adsense program in Gmail reads a user's emails to display targeted ads. Facebook [10] has already been criticized regarding data privacy issues in its short existence, such as for broadcasting purchase information on news feeds [3, 39]. Companies may also engage in censorship. Normally, this is to eliminate offensive content but there have been some reports of inappropriate censorship. For example, Facebook has been accused by various blogs of censoring their competitors' names [36] and tampering search results for a presidential candidate [35]. Second, companies only provide sharing services at little or no cost if they can eventually profit from it. This means that specialized applications that are useful to a small group of people may remain undeveloped due to its lack of popularity. It would be beneficial to allow these users to maintain their own storage without paying for expensive data storage services, such as the Amazon S3.

In a decentralized system, the data repository is distributed on a set of machines belonging to users who want to share data with each other. In contrast to centralized data repository solutions, decentralized peer-to-peer solutions have several advantages. First, users can choose to store data only on nodes they manage and trust as opposed to a third party's machines. Therefore, there is less concern for privacy and censorship. Second, decentralized solutions can have a grass-root deployment to support a wide variety of different applications, even unprofitable ones. Because of these potential advantages, this thesis explores a decentralized design for a multiple-writer data repository, which spreads the storage over peer nodes owned by individual users.

1.2 Challenges

There are three requirements for a decentralized data repository supporting multiple writers: durability, availability, and consistency. A storage system is durable if the data can be recovered in the event of permanent failures, such as disk crashes. Availability refers to how likely a piece of data can be retrieved in the face of temporarily unavailable nodes. The main strategy to achieve high durability and availability is to replicate data onto multiple nodes. In this way, even if a node fails or goes offline, the data that was stored on the failed node can be retrieved or recovered from another replica [42]. Since writers can potentially modify replicas stored at different nodes and many data operations involve many different objects, maintaining consistency among multiple replicas becomes a challenge. The consistency requirements of a system dictate that the same sequence of data modifications are applied to all replicas at different nodes. Consider the example where replica nodes A and B both perform modifications to the repository. If a system does not provide consistency, it is possible that one replica applies writeA then writeB whereas another replica applies writeB then writeA, resulting in different replica states. If a system is consistent, the final ordering of the operations will be identical for all nodes.

It is difficult to achieve the durability, availability, and consistency requirements in a decentralized fashion among peer nodes for the following reasons:

1. *Peer nodes have limited bandwidth and storage space.* This prevents us from simply replicating the entire repository on every node because transferring all replicas to all nodes would consume too much bandwidth and the nodes might not have enough disk space to store all the data in the repository. Therefore, a decentralized solution must employ more flexible replication techniques to efficiently ensure high data durability and availability.

2. *Peer nodes have low availability.* Since peer nodes may go offline for extended periods of time, a decentralized solution must use a replication factor larger than that required in a centrally managed system to ensure that the overall system is highly available. More importantly, low node availability poses challenges in maintaining data consistency. Strong consistency, where all nodes see an identical total ordering of writes at all times, is ideal for usability. However, ensuring strong consistency where participating nodes are not highly available has been shown to cause limited data availability [14, 23, 5]. In order to achieve a reasonable level of data availability, we must relax the consistency requirements.

3. *Over the lifetime of its operation, the system might incur membership churn where new nodes join and existing ones leave occasionally.*. The system must be able to reconfigure itself to reflect the membership changes without compromising its availability and consistency goals.

1.3 Friendshare's solutions

This thesis presents Friendshare, a decentralized multiple-writer data repository where the repository is distributed across the peer nodes. Any user running Friendshare can create a repository named by a globally unique ID. Other users can then request to become members of the repository. The repository can be written, managed, and organized by the members of the group and can be potentially read by all users in the system.

There are three classes of repository users: members, admins, and the primary. Any user that has joined the repository is a member. Admins are a set of privileged members that help maintain the consistency of the repository. One of the admins is elected to be the primary and is responsible for administrative tasks, such as approving new members and promoting members to become admins. Since admins are hand-picked by the primary, we expect them to remain in the system for an extended period of time and to be trustworthy. This provides several benefits to the system. Because admins are expected to eventually come back online, the system does not need to aggressively generate new replicas whenever an admin goes offline. In addition, the system does not need to maintain a high replication factor. Finally, users can trust that the admins will not act maliciously, such as by refusing service or changing repository content.

Friendshare replicates the repository content on the admins to maintain high durability and data availability. In order to support efficient replication, Friendshare separates the management of metadata from the data. Metadata is a description of the data, including information such as the name and location of the data and the organization of different data objects within the repository. For example, in a photo repository, the data objects are the photos and the metadata describes the virtual file system hierarchy in which the photos are organized. The metadata is replicated on all admins while the data is distributed across the members. Since the metadata is much smaller than the actual data, metadata replication is inexpensive. Only the metadata is mutable whereas all data objects remain unchanged after creation. Because data is immutable, Friendshare only needs to perform expensive replica synchronization required for consistent access on the small metadata. As a side advantage, because all admins hold a complete replica of the metadata, it is possible to browse the organization of the repository quickly by contacting any one admin node. The approach of separating the metadata from the data is similar to PRACTI's separation of invalidation messages and update messages [9]

Since providing strong consistency causes limited data availability, Friendshare strives to achieve eventual consistency. Eventual consistency ensures that new writes will eventually propagate throughout the system via gossiping techniques and the ordering of all write operations form a total order, eventually leading to identical metadata states at all admins. Friendshare's consistency system is similar to the Bayou data replication scheme [29, 37]. The admins periodically gossip with each other to exchange any new writes and the primary determines the total ordering of the writes to ensure that all admins have a consistent view of the repository.

Throughout the lifetime of the system, new users may request to join the repository and existing members may leave either gracefully or unexpectedly. When such events occur, Friendshare repositories can automatically reconfigure itself. Each member in the repository will update its membership list to reflect the joins and leaves. If an admin leaves, the primary can promote another member to become an admin in order to maintain the same metadata replication factor. If the primary leaves the repository unexpectedly, the remaining admins run a consensus protocol to elect a new primary.

1.4 Contributions

We contribute the design and implementation of a decentralized community-based data repository system that allows multiple users to create, delete, and organize data in a repository. Our design is suitable in a deployment environment where nodes have limited resources and low node availability. We explore the usage of consensus protocols in a decentralized peer-to-peer environment and offer a number of optimizations to improve its running time. As proof of concept, we built a media repository application on top of our system that allows multiple users to share and organize photos, videos, and other files into a common repository.

1.5 Organization of thesis

In Chapter 2, we discuss the challenges of decentralized storage in further detail and present an overview of Friendshare's design. This leads to an in-depth discussion of the Friendshare architecture in Chapter 3, including details on the metadata replication scheme. Chapter 4 describes the reconfiguration process of Friendshare after node joins and departures. In Chapter 5, we will discuss our implementation of Friendshare. Chapter 6 will contain simulations and evaluations that we observed with Friendshare. Finally, in Chapter 7, we will talk about the related work on the topics of sharing and data replication.

Chapter 2

Design Overview

In this chapter, we will first discuss Friendshare's design challenges in detail and then describe the basic approach of Friendshare to solve the challenges.

2.1 Design challenges

The goal of Friendshare's design is to provide data durability, availability, and consistency even though peer nodes have limited resources, low node availability, and may occasionally leave the system unexpectedly.

A strawman design for a read/write data repository is to store all data on a single peer node. Such a design is flawed for many reasons. First, since the network has limited bandwidth and storage space, it would take too long to upload the entire repository's data onto the single node and the single node's hard disk might not have enough storage space to contain everything. To rectify this problem, we want a solution that spreads data across many nodes. Second, the repository maintained by a single peer node has low data availability and durability because if the single node goes offline or crashes, all data would be unavailable or lost. Thus, the repository data should be replicated on multiple peer nodes in order to improve data availability and durability.

The main advantage of the strawman design is that it achieves data consistency trivially because the single peer node handles all read and write requests for all objects. On the other hand, employing distributed data replication creates consistency challenges. When data is replicated across multiple nodes, any sequence of write operations that are performed by one replica should be performed in the same order on the other replicas. Since node failures are common in peer-to-peer networks, a service must accept write operations at any node in order to remain available. This, in turn, may cause writes to be accepted concurrently of each other at different nodes. Therefore, there must be a consistent total ordering of these concurrent writes in order to retain eventually-identical images across all the replicas.

Maintaining strong consistency across all the replicas is the ideal choice from an application standpoint but it can be costly to maintain in peer-to-peer environments. In general, strong consistency schemes need to employ consensus algorithms or lock implementations to cope with potential failures, both of which exhibit limited data availability when operating in low-availability networks [4, 14, 23, 5]. Weakening the consistency requirements to be eventually consistent increases data availability and scalability.

Adding to the difficulties is that, once in a while, nodes might leave the system forever unexpectedly. For example, the user may lose interest in the system or their hard disk may crash. When this happens, the data stored on that node can never be retrieved. In this case, we must ensure that a new replica is created elsewhere. Abrupt departures can also cause problems for consistent data replication schemes so the system must be able to reconfigure itself carefully when such events arise. We will first present the details of Friendshare's data replication scheme in Chapter 3 and then discuss the reconfiguration process in Chapter 4.

2.2 Basic idea of Friendshare

Friendshare is a decentralized multiple-writer data repository where the data is distributed and replicated across multiple nodes in the network. Friendshare allows members to organize data in the repository into a file system-like hierarchical namespace by making directories and adding files into those directories. Each repository has a set of admin nodes, which are privileged members that help maintain node membership and consistency of the repository's hierarchical namespace. Since admins are manually promoted, we assume that they are not malicious, have reasonable availability, and will stay in the system for a long time. The admins elect one of themselves as the primary, who is responsible for handling various administrative tasks such as approving of new member joins, promoting members to become admins, and evicting malicious members.

Friendshare treats metadata differently from data. Metadata contains a description of the data object, including the physical node location where the data is stored and how the object is organized into the hierarchical namespace. Metadata is mutable and is replicated by all admins using a protocol that ensures eventual consistency. Data is immutable and each data object is replicated on a small set of member nodes. The metadata is much smaller than the data and therefore the expensive eventual consistency protocol is run on a much smaller size. As an additional advantage, since a complete replica of the metadata exists at each admin, users can browse and search the entire collection of metadata from one of the admins.

Data objects are created from local files and are named with a *fileID*, which is generated by running a hash algorithm on the file content. There are several ramifications of using the file contents to generate the data object's *fileID*. First, we can easily distinguish if two files are the identical by comparing their *fileID*s. This is desirable because each file has its own unique global name, regardless of the original filename. For example, a picture of a dog can be named jip.jpg, dog.jpg, or DSC1523.jpg, but the data object generated from each file will have the same *fileID*. Second, the *fileID* acts as an implicit security authentication to check if the file that the user downloaded corresponds to what the user expected from its *fileID*. After downloading, the user can simply run the hash algorithm on the file to check if it indeed generates the *fileID* expected. Third, data objects are immutable and cannot be changed. If a file is modified, it will correspond to a new data object with a different *fileID*. If a file is modified after it was shared to a repository, the member should issue a write to delete the old data object and then issue another write to add the new data object.

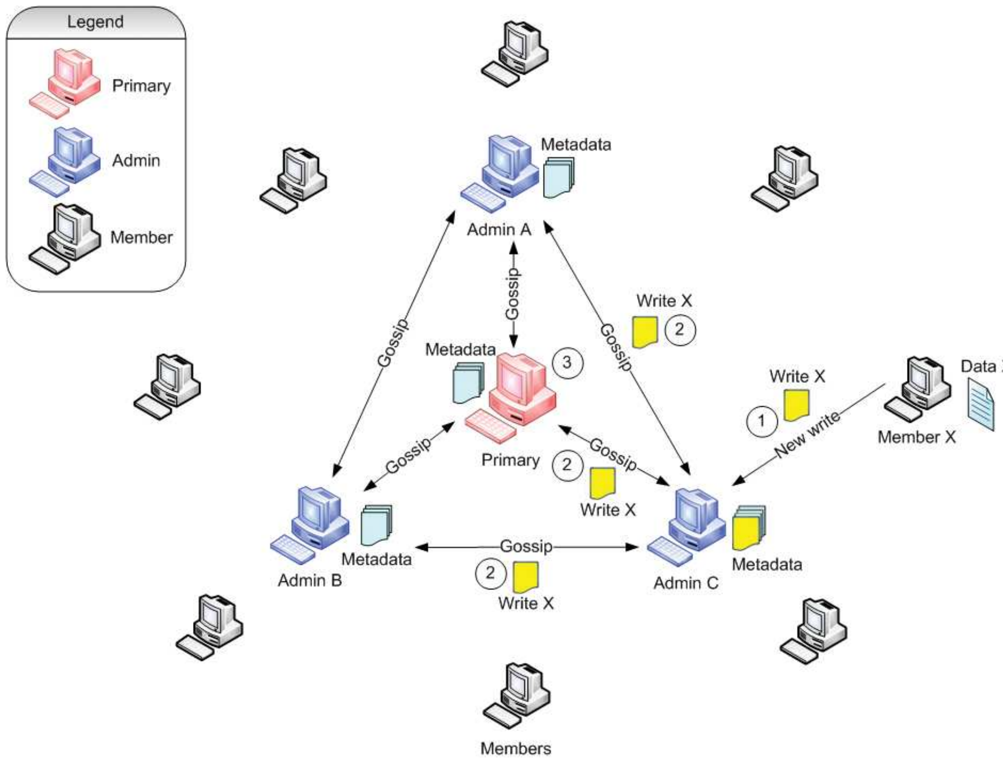


Figure 2.1: *Repository architecture writing.* Member X writes/modifies data in the repository by sending a new write request Write X to Admin C. The write is then propagated to the other admins via gossiping. The primary eventually receives and commits the write.

When a member wants to write a file to the repository, it first creates a data object that points to the file. The member then creates a write request corresponding to the data object that specifies the nodes where the data file is physically stored and the virtual directory in which the file should be placed (e.g. /photos/2008summer/). Finally, the member sends the write request to one of the admins in the repository. Upon receipt of the write request, the admin accepts and stores the write in a local write log and gossips with other admins to propagate the new write in the system, providing eventual consistency. Newly accepted writes are marked as *tentative* since it can be preceded by other writes that the admin has not yet received. The primary is responsible for committing the tentative writes to establish a total-ordering of the write operations. Figure 2.1 illustrates the following write process:

1. To share Data X in the repository, Member X sends a write request for Data X (Write X) to Admin C in the repository.

2. Admin C accepts the write and eventually gossips the new write to the other admins A, B, and the primary.

3. The primary receives Write X and commits it. The commit action will eventually be propagated to all the other admins in the repository.

In order to view data in the repository, a user queries a random admin to download the repository state. The admin constructs the repository state locally from its write log and sends it back to the user. After the state has been received, users can choose to download files described in the state. Embedded inside the state are the member locations where each file is stored so the user can download the file by connecting directly to the members. Figure 2.2 illustrates the following read process:

1. Member X wants to view the data in the repository so it gets the state from Admin C.

2. After viewing the state, Member X decides to download Data Y, which is located on Member Y.

3. Member X connects to Member Y to download Data Y.

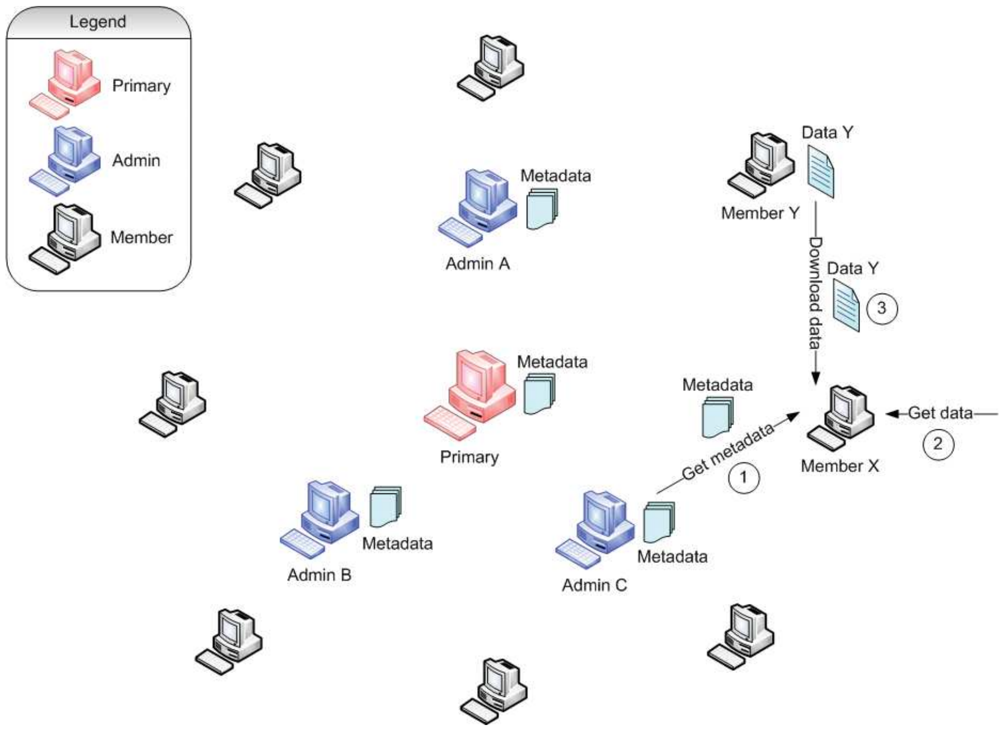


Figure 2.2: *Repository architecture reading*. Member X requests the metadata from Admin C and decides to download Data Y from Member Y.

The metadata write log can be parsed in order to form the repository's virtual file system representation, which can be displayed to the user. Figure 2.3 shows the virtual file system representation generated by parsing a sample write log.

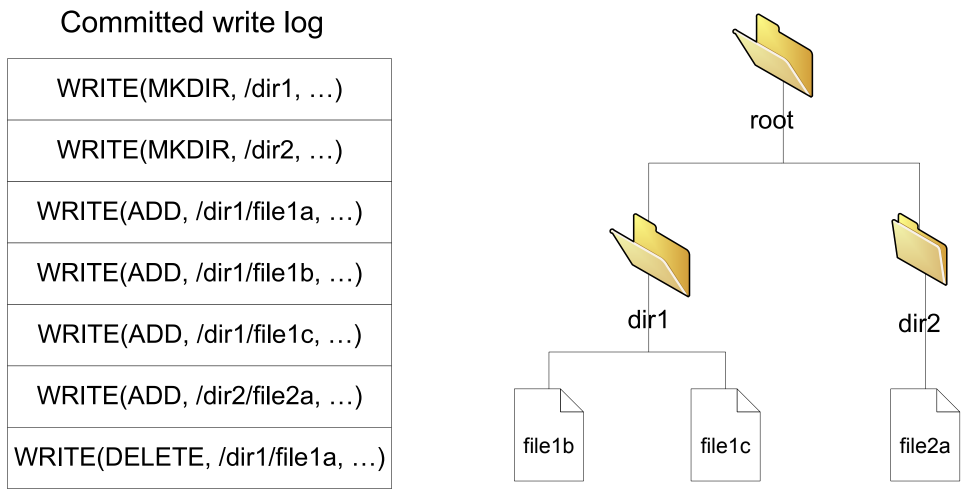

Committed write log
WRITE(MKDIR, /dir1, ...)
WRITE(MKDIR, /dir2, ...)
WRITE(ADD, /dir1/file1a, ...)
WRITE(ADD, /dir1/file1b, ...)
WRITE(ADD, /dir1/file1c, ...)
WRITE(ADD, /dir2/file2a, ...)
WRITE(DELETE, /dir1/file1a, ...)
root
dir1
dir2
file1b
file1c
file2a

Figure 2.3: Write log and its virtual representation

Eventual consistency plays a role in viewing data. In eventual consistency, tentative writes may differ from admin to admin. This can happen if concurrent writes are sent to different nodes. If tentative writes are included in the results displayed to the user, the user may see inconsistent results depending on which admin it queried. However, if only committed writes are shown, users may not see new data writes because those writes have not yet been committed. Since this decision is application specific, Friendshare allows the application to design the UI, in which the application developers can decide how to display the writes. To support this feature, Friendshare admins send both tentative and committed writes but flag the tentative results so the application can distinguish between the different types of writes.

From time to time, the membership of the repository can change as users join and leave. As will be discussed in Chapter 4, Friendshare can tolerate unexpected departures from all members, using the normal write protocol, except the primary. If the primary dies, the admins run a consensus protocol to elect a new primary. Every instance of a Friendshare repository operates under a specific *view*, identified by a view number. At the end of the election process, the new primary increments the *view*. The view number is attached to all admin messages (e.g. gossip) so that the admins can identify if a message is outdated.

Because Friendshare only provides eventual consistency, some applications may not be suitable to be built on top of Friendshare. For example, financial applications require stronger consistency to implement transactions properly. Other examples of applications that may not be suitable include distributed databases, file systems, and version control systems. Additionally, because data is immutable, modifying an existing file is costly because it requires creating a new data object and garbage collecting the old one. Therefore, Friendshare is more suitable for applications where the data does not frequently change, even though it also supports applications that do not fit this criterion. Some suitable applications for Friendshare are as follows:

1. Media (photo/video/music) repository that can be organized by a group of friends. Media files generally remain unmodified, which makes it suitable for Friendshare.

2. Broadcast authored work: blogs, vlogs, stories, poetry, composed music. Authored work are usually final after they are published.

3. Bulletin boards for people with common interests. Since bulletin board posts rarely change after posting, we can store each post as a separate file.

4. Review repository for restaurants, clothing stores. As before, reviews are unmodified, and therefore are suitable for Friendshare.

As can be seen, Friendshare can be used in a variety of different ways. In order to facilitate this, we wrote a flexible API to allow developers to easily construct their own applications with the Friendshare backend.

2.3 Friendstore data replication

Although we have described how repository metadata is replicated on the admins, we have not yet discussed how Friendshare replicates repository data. For this purpose, Friendshare is built on top of Friendstore [38], a cooperative peer-to-peer backup system where data is stored on trusted nodes. Traditional peer-to-peer backup systems replicate data on arbitrary peer nodes and have low availability. In Friendstore, each node only replicates its data on a subset of trusted peer nodes, typically belonging to a user's friends or colleagues. Because friends in real life have agreed to cooperate with each other to share their nodes' storage resources, trusted peer nodes tend to be more available for each other's requests.

The Friendstore implementation allows users to choose a list of friends with whom they entrust their data and a list of directories to backup. These directories are monitored and any changes are automatically backed up onto the user's friends, or "helpers". Periodically, owners request hashes of the replica data stored on the helpers in order to verify the integrity of the replica and recreate new replicas if existing ones are lost.

During setup, each user in Friendstore is assigned a public-private key pair that is used for authentication. The *userID* is generated by taking the SHA-1 hash of the public key and concatenating the machine name. Due to the property of hash functions, the *userID* is guaranteed to be globally unique. The *userID* is self-certifying, similar to SFS's self-certifying pathnames [25], in which users can verify the identity of a remote user by checking if its *userID* matches its public key.

Friendshare uses Friendstore to increase data durability and availability through replication. When a user wants to write a file to a repository, Friendshare sends the file to Friendstore to backup onto the user's friends. The file's metadata is then written to the repository by the process described in Chapter 3. Included in the write request is the location of the file, which includes the owner-user's computer and the owner's helpers. When another user wants to download a file, Friendshare performs load reduction by allowing the owner and its helpers to serve the file.

Chapter 3

Metadata Consistency

Friendshare's metadata replication scheme is based on Bayou [29, 37]. This chapter describes how Friendshare maintains metadata consistency when the admin membership configuration is static. Membership reconfiguration will be discussed in the next chapter.

3.1 Overview of the write process

As mentioned in Chapter 2, a member of the group writes a local file into a repository by creating a data object pointing to the file and then creating a metadata WRITE request for the data object. WRITE requests include the write type, virtual repository location, accept-stamp, commit-stamp, owner (which is the member that created the write), and the data object. The write type can be actions such as ADD, DELETE, MKDIR, RMDIR, JOIN_GROUP, LEAVE_GROUP, etc, which defines the write's actions on the repository. The virtual repository location is the parent directory in the virtual repository file system where this write should be performed (e.g. /photos/2008summer/). For writes that do not affect the file system, such as JOIN_GROUP and LEAVE_GROUP, this field is unused. The accept-stamp and commit-stamp are left empty at creation time and will be filled in later. Figure 3.1 shows the WRITE request fields.

A summary of the write process is shown below:

Write Type	Virtual Location	Accept-stamp	Commit-stamp	Owner	Data Object

Figure 3.1: WRITE request

1. A member wants to share a local file in the repository, so it creates a data object with a unique *fileID* based on the file content.
2. The member creates a write request for the data object. This write request could be to ADD a file, DELETE a file, MKDIR, RMDIR, etc.
3. The member sends the write request to an admin in the repository. If the member is an admin itself, this step can be skipped.
4. The admin stamps the write's accept-stamp with the current value of its clock and adds the write to its tentative write log.
5. The admin gossips the write to all other admins, including the primary.
6. The primary receives the write, stamps the write's commit-stamp, and commits the write. The admins will consequently move the write from its tentative write log to its committed write log when they learn of the commit via gossiping. The write can now be seen by all users.

We describe this process in further detail in the following sections.

3.2 Accepting writes

We use the Lamport clock [21] to determine a total ordering of operations on the metadata among all admins. Each clock consists of a counter and a unique value, which we set to be the *userID* that is assigned when the user installs Friendshare. If two clocks' counters are equal, we break the tie by comparing the unique value. The clock operates by obeying the following rules. First, the clock increments its counter on every local event. Second, when communicating with a remote admin, the clock synchronizes with the remote admin's clock by setting its clock counter to be greater than the maximum of the two clocks.

Admins have two local write logs: a tentative write log and a committed write log. When an admin receives a write from any member, the admin will stamp the write's *accept-stamp* with its clock value and add the write to its tentative write log. Initially, all new writes are marked to be *tentative* since the admin may later receive a write from another admin through gossiping that precedes existing writes.

Version vector

My clock (Admin A)	6 : aID
Admin B	3 : bID
Admin C	2 : cID

Committed write log

	Commit-stamp	**Accept-stamp**
WRITE(MKDIR, /dir/, …)	0:View0	1 : cID
WRITE(ADD, /dir/file1, …)	1:View0	2 : cID

Tentative write log

	Commit-stamp	**Accept-stamp**
WRITE(ADD, /dir/file2, …)	∞	2 : bID
WRITE(ADD, /dir/file3, …)	∞	5 : aID
WRITE(DELETE, /dir/file2, …)	∞	3 : bID

Figure 3.2: Admin A's write logs and version vector

Each admin keeps a version vector VV, which holds the last accept-stamp known to be accepted by all admins in the system. For instance, admin X's version vector entry for admin A, VV_A, would contain the largest accept-stamp that X has received from A. The version vector is used both during gossiping and committing.

Figure 3.2 shows Admin A's tentative and committed write logs as well as its version vector. There are 2 other admins in this example repository where the last accept-stamp that Admin A has seen from Admin B is 3 and from Admin C is 2. The version vector also stores A's current clock value (6 : aID). Admin A's *userID* is aID, B's *userID* is bID, and C's *userID* is cID. These *userID*s are also used as the unique value of the Lamport clock, as can be seen in the accept-stamps. Two writes (MKDIR and ADD(file1)) accepted by Admin C have already been committed and so the *dir* directory containing *file1* is visible to users. Both commits happened in the same *view*. Several writes remain tentative, as evidenced by the infinite commit-stamp value.

3.3 Propagating writes

The admins gossip with each other to propagate new writes and to synchronize their clocks. Admins periodically choose a random admin with whom to perform gossiping. First, the admins, which we will refer to as A and B, exchange clock values in order to synchronize their clocks. Second, if admin A is gossiping to admin B, A will request for B's version vector. Using B's version vector, A can determine the writes (both tentative and committed) that B has not yet seen. Therefore, instead of sending all the writes in its write log, A only needs to send the incremental write changes. Figure 3.3 illustrates the gossip protocol where admin A gossips to admin B.

3.4 Committing writes

Tentative writes must eventually be committed in a consistent order across all the admins. Friendshare uses a *primary commit* scheme [37] that designates one admin as the primary and allows it to commit writes that it sees. As long as the primary

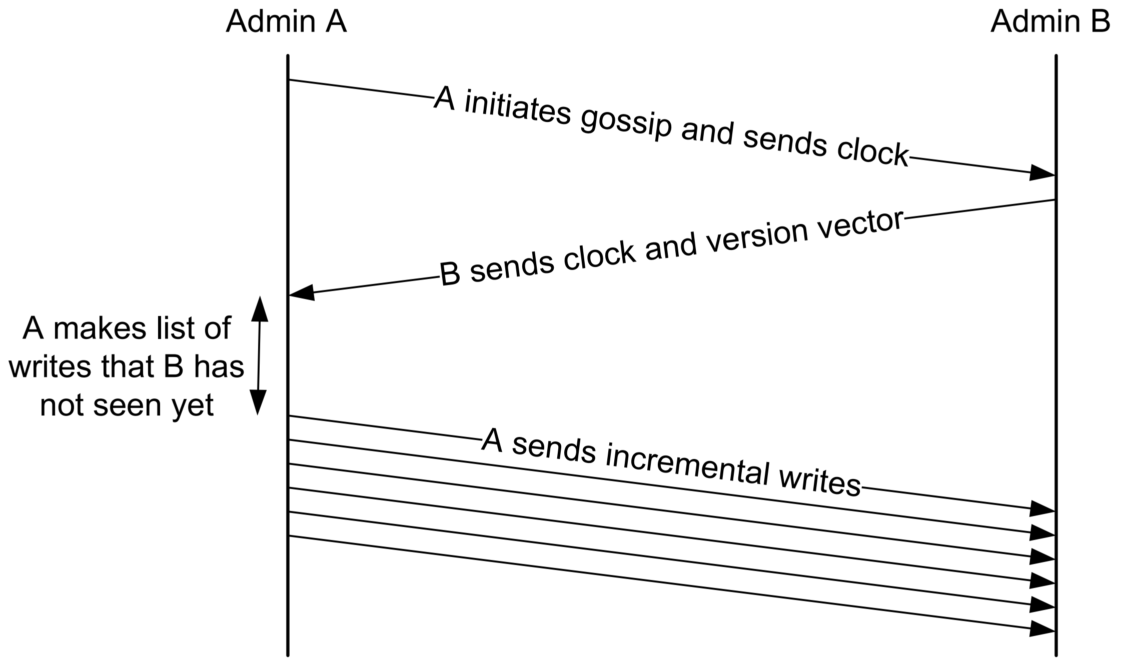

Admin A
Admin B
A initiates gossip and sends clock
B sends clock and version vector
A makes list of writes that B has not seen yet
A sends incremental writes

Figure 3.3: Gossip protocol

node is online, it can commit writes despite the unavailability of other admin nodes. Thus, this approach is better suited for environments where nodes can be offline for extended periods of time. If the primary is offline temporarily, admins can continue to accept tentative writes, which can be committed when the primary comes back online. When the primary commits a tentative write, the write is removed from the primary's tentative write log, stamped with a commit-stamp number, and added to the primary's committed write log. The commit-stamp consists of the current view number and a monotonically increasing counter. Commit-stamps are ordered first by view number and then the counter so that committed writes from earlier views are ordered before later views. The news of the commit is then gossiped to the other admins so that all other admins will eventually learn about the committed writes and move the writes from the tentative write log to the committed write log. As time progresses, the write logs can become longer and longer. To prevent infinitely long write logs, the primary can command all the admins to checkpoint the committed writes and truncate the logs.

Chapter 4

Membership reconfiguration

From time to time, new users may join and existing members may leave a repository. When the repository membership changes as nodes join and leave, the repository must be able to reconfigure itself to maintain high data availability, durability, and consistency. This chapter will describe the Friendshare reconfiguration process.

4.1 Handling joins and non-primary departures

Users can request to join an existing repository by sending a REQUEST_JOIN to one of the admins in the repository. Upon receiving the request, the admin will write the REQUEST_JOIN into its tentative write log. Eventually, the primary will receive the REQUEST_JOIN through gossiping and decide whether to approve the join request. If it approves of the user, the primary will add and commit a new JOIN_APPROVED write and add the member to its membership list. If it rejects the user, the primary writes a JOIN_REJECTED write. The other admins will eventually learn of the primary's decision via gossiping and they will subsequently update their membership lists as well.

In order to discover whether it has successfully joined the repository, joining users periodically query a random admin of the repository to inquire if the joining user now exists in the admin's membership list. After the user successfully joins, it can issue writes into the repository.

Existing members may occasionally leave a repository. Our definition of a node "leaving" the repository group is when either the node has explicitly requested to leave the group, which is the graceful departure scenario, or the node has been offline for a prolonged period of time and is deemed unlikely to ever come back online. Each admin keeps the *last-online-time* of every other member in the repository and exchanges this information to other admins during gossiping. If the other member is an admin, the *last-online-time* is updated after every successful gossip session. For non-admin members, admins must periodically ping the member to update its *last-online-time*. If a member has been offline for more than a pre-specified *timeout* value, the member is deemed to have left the system. This *timeout* should be set to be a reasonable value depending on the environment in which the system is run. For example, a typical user's desktop machine may go offline for several days so the *timeout* should not be set too short.

When the primary detects that a member has left the repository, the primary will issue a LEAVE_GROUP write specifying the departed member, commit the write, and remove the member from its membership list. The LEAVE_GROUP write will be propagated to the other admins via gossiping. When each admin receives the committed LEAVE_GROUP write, it removes the departed member from its membership list.

Graceful departures follow a similar procedure, except that the departing member issues the LEAVE_GROUP write instead of the primary. This allows the member to be removed faster since the primary does not need to wait for the pre-specified *timeout* before detecting the departure. Through gossiping, the primary will eventually receive the LEAVE_GROUP write, commit it, and remove the member from its membership list. The committed LEAVE_GROUP write will then be gossiped to the other admins.

If the departing member was an admin, the metadata replication factor of the repository will have decreased by 1 so the primary must determine whether it should promote another member to become an admin in order to maintain metadata availability and durability.

4.2 Primary election and view change

If an admin detects that the primary has left the repository, the admin initiates a variation of Paxos [22, 26], a consensus protocol, so all the admins can agree on a new configuration that replaces the departed primary node with a new one. The Paxos protocol allows a number of distributed nodes to eventually reach a consensus on something, such as a number, as long as a majority of participating nodes are available. The Paxos protocol is used in many distributed applications for reconfiguration including Google's Chubby lock service [7] and Microsoft's Autopilot cluster management service [18].

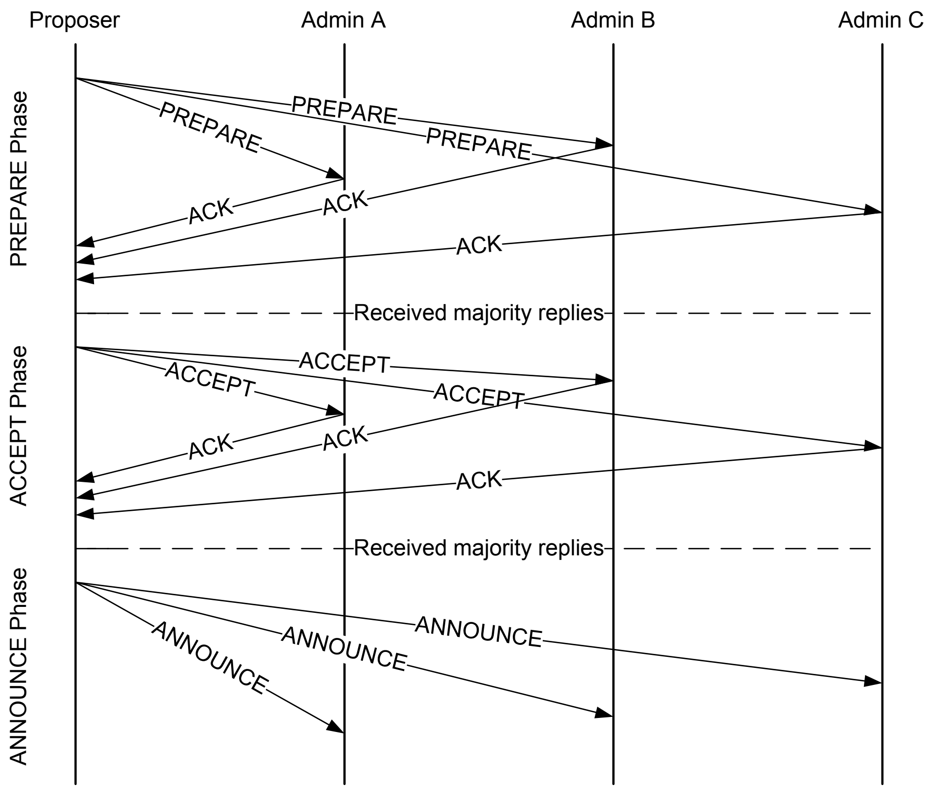


Figure 4.1: Paxos protocol with 4 admins

The basic protocol consists of three phases: PREPARE, ACCEPT, and ANNOUNCE. Figure 4.1 shows the flow of the protocol.

1. PREPARE phase: A proposer chooses a unique proposal number n, which is generated by concatenating a monotonically-increasing counter with the proposer's *userID*. The proposer then sends a PREPARE request with this proposal number n to all other admins. Each admin responds to PREPARE messages as follows:

   (a) If the admin has not seen a PREPARE request with proposal number greater than n, it will send back the highest proposal number and its corresponding value that it has accepted from an ACCEPT request (or NULL if none have been accepted yet) and promises to never accept another PREPARE request with a proposal number less than n.

   (b) If the admin has already seen a PREPARE request with a proposal number greater than n, then it sends back a decline reply.

   If the proposer receives promises from a majority of admins, it can move onto the ACCEPT phase. If the proposer did not receive a majority of promises, it will wait for currently offline admins to come back online.

2. ACCEPT phase: The proposer takes the value v proposed by the highest accepted proposal that it received in the replies from the PREPARE phase. If no admin replied with a proposal number (this can happen if this proposal was the first proposal ever), then the proposer can choose any new value v. In the context of Friendstore, the value v is the next membership configuration, which specifies the new primary and the new view number. It then sends an ACCEPT request with proposal number n and value v to the other admins. Each admin responds to ACCEPT messages as follows:

   (a) If the admin has not replied to a PREPARE request with proposal number greater than n, it will accept the proposal. The admin needs to write its accepted proposal number and value to disk so that it can recover in the event of a crash.

   (b) If the admin has already seen a PREPARE request with a proposal number greater than n, then it sends back a decline reply.

   If the proposer receives accepts from a majority of admins, a consensus has been reached with value v. However, the other admins do not know about the consensus so we need to perform the final phase.

3. ANNOUNCE phase: The proposer sends an ANNOUNCE message to the admins, informing them that a new value v has been agreed upon. No reply is necessary. At the end of Paxos, the admins have agreed upon value v, which identifies the new primary and the new view number.

The ANNOUNCE phase of Paxos may not succeed in notifying all admins in the repository since there may have been some offline admins during the phase. To ensure that all admins eventually discover the view change, admins check their view numbers at the beginning of the gossip protocol. If the view numbers are not equal, the admin with the lower view number will request a dump of the repository metadata from the higher admin.

At the start of the new *view*, the new primary immediately commits any tentative writes in its write log. It is possible that there were writes committed in previous *views* that the new primary did not see yet, which can cause inconsistencies. As an example, consider a repository with admins A, B, C, and D where A is the primary in *view* 2. A has committed writes up to commit-stamp counter 10, B has seen all committed writes up to 10, C up to 9, and D up to 8. A dies and the other admins run Paxos, eventually electing one of them as the new primary in the new *view* 3. There can be inconsistencies in the admins' commit write logs depending on who is elected as the new primary. If B becomes the new primary, there are no inconsistencies since B has seen all the writes that A has committed. If C becomes the new primary, it can commit one of its tentative writes with commit-stamp counter 10, which is inconsistent with the original write with commit-stamp 10 that was committed by A in the previous *view*. Similarly, D can commit two writes with commit-stamps 9 and 10, resulting in an inconsistent write log. For this reason, Friendshare orders all committed writes from previous views before any committed writes from later views regardless of the commit-stamp's counter value. Using this ordering scheme, if D becomes the new primary, it can commit tentative writes *w(9, 3)* (with commit-stamp counter 9 and view number 3) and *w(10, 3)* and eventually, all admins' committed write logs will be consistent: ..., *w(9, 2)*, *w(10, 2)*, *w(9, 3)*, *w(10, 3)*.

4.3 Paxos performance problems

Paxos does not guarantee liveness [22] and thus it is possible for the proposal to take an infinite amount of time before reaching agreement. The following example illustrates why a proposal might not succeed even though all nodes are available. During reconfiguration, an admin proposes with proposal number 1. While the first admin is in the PREPARE phase, another admin proposes with proposal number 2, so that when the first admin proceeds to the ACCEPT phase, many admins will reject its request since they have seen a higher number (proposal 2). So, the first admin proposes with proposal number 3, which blocks proposal 2, and so on.

In order to rectify this problem, Lamport suggests choosing a "distinguished proposer" to be the only node that starts proposals [22]. However, if nodes are commonly offline, which is precisely our situation, this would not work well because the "distinguished proposer" may be offline, so a new proposal would not be initiated until that node comes back online.

One possible way to alleviate this problem is to use an exponential backoff time. If a proposal fails, the proposer will double its backoff time before starting a new proposal. This will decrease the probability of proposal conflicts. This works pretty well in normal situations when a majority of admins are online. However, because Friendshare runs on ordinary users' desktop machines, we expect admins to be offline more often than they are online. The effect of this "offline-heavy" tendency is that proposals take longer to complete because proposers will spend most of their time waiting for admins to come back online. As the proposal duration lengthens, the probability of proposal conflicts increases. Simulation results in Chapter 6 show that as admins become more offline-heavy, the basic Paxos protocol would increasingly fail to reach a consensus.

In Paxos, a proposer must wait until it has received a majority of promises before it can move onto the next phase. By the time the proposer can collect a majority of replies, a multitude of events may have happened:

1. The proposer may have gone offline. Since proposals are not continued after coming back online, this effectively terminates the proposal.
2. Another proposal may have overwritten the old proposal. The old proposal's proposer will discover this when it sends requests in the next phase or when it receives requests from the newer proposer.
3. Some of the previously-online admins that accepted the PREPARE request may have gone offline. Therefore, when the proposer moves onto the ACCEPT phase, there may not be a majority online. In this case, the proposer once again must wait for a majority to come online. If another proposal starts at this point, all progress is lost. An example of this is described below (which we will refer to as the "Overwrite" case):
    (a) Admin1 sends PREPARE requests to admins 2, 3, 4, 5 (10 admins in all), but is waiting for one more admin to come online in order to form a majority.
    (b) Admin3 goes offline then admin6 comes online.
    (c) Admin1 sends a PREPARE request to admin6. Since it now has a majority, it moves onto the ACCEPT phase.
    (d) Admin1 sends ACCEPT requests to admins 2, 3, 4, 5, 6 but finds that admin3 has gone offline. Therefore, it must wait for another admin to come online to form a majority.
    (e) Admin2 starts its own PREPARE phase, and broadcasts PREPARE requests that overwrite admin1's proposal. All progress is lost and we need to start from the PREPARE phase again.

If a majority of admins are online at any point, proposals generally succeed. This is because proposals finish faster when proposers do not need to wait for admins to come online and therefore, there are less proposal conflicts. On the flip side, if a majority of admins do not come online, the basic Paxos protocol almost never succeeds. This is because while proposers are waiting for a majority to come online, another proposal may have overwritten it (see the "Overwrite" case above). Unfortunately, in offline-heavy situations, it is quite rare for a majority of admins to be online at one time. Therefore, the "Overwrite" case happens very frequently.

4.4 Paxos optimizations

We propose two techniques to optimize the basic Paxos protocol so that it will work well in an offline-heavy environment. In the Eliminate-Duplicates optimization, we reduce proposal conflicts by preventing new proposals when there is an existing proposal. In the Virtual-Token optimization, offline admins allow other admins to act on their behalf to virtually increase the number of admins online.

4.4.1 Eliminate-Duplicates optimization

One problem with the basic protocol is that long-running proposals can be overwritten by another proposal, which eliminates any progress and forces the consensus to start from the beginning. One way to reduce conflicts is to reduce the number of concurrent proposals. We propose an optimization, which we will refer to as the Eliminate-Duplicates optimization, to reduce the number of proposals by preventing the creation of new proposals if there is already an existing proposal. It is important to note that we do not care about which admin succeeds in creating a proposal so long as a proposal succeeds. We describe the optimization in detail using the "Overwrite" case:

1. Admin1 successfully sends PREPARE requests to admins 2, 3, 4, 5, but is waiting for one more admin to come online in order to form a majority.

2. Admin3 goes offline then admin6 comes online.

3. Admin1 sends a PREPARE request to admin6. Since it now has a majority, it moves onto the ACCEPT phase.

4. Admin1 sends ACCEPT requests to admins 2, 3, 4, 5, 6 but finds that admin3 has gone offline. Therefore, it must wait for another admin to come online to form a majority.

5. Admin2 wants to start its own proposal but it knows that admin1 has already started a proposal. It must contact admin1 and determine the proposal's status. At this point, several things may have happened:

   (a) Admin1 is still proposing. In this case, admin2 cancels its own proposal and initiates backoff. After its backoff, it will repeat the above procedure.

   (b) Admin1 has gone offline. Since admin 1's proposal cannot succeed now that it is offline, admin2 can start its proposal.

   (c) Admin1 has stopped proposing. This can happen if admin1 has received a higher proposal request from another admin (which has not reached admin2 yet) and has terminated its own proposal. In this case, admin2 will repeat the above procedure with the new proposer to discover whether it is still proposing.

Our simulations show that this optimization improves performance as compared to the basic Paxos protocol, especially in offline-heavy situations. It effectively eliminates most proposal conflicts and enables one proposer to continue as long as possible.

The Eliminate-Duplicates optimization requires an admin to check the status of existing proposals before proposing. To simplify the implementation, we piggyback the status check with the gossip protocol. During gossiping, admins exchange their proposer status in addition to their version vectors. If an admin is currently proposing, it includes `proposerStatus=TRUE` in its gossip message. Each admin will keep the proposer status for every other admin. When an admin wants to propose, it can simply check the last status of the highest proposer. If the proposer was online during the last gossip attempt and its `proposerStatus=TRUE`, then the admin will not start a proposal. If the proposer was offline during the last gossip attempt or if the proposer's `proposerStatus=FALSE`, then the admin can start a new proposal.

4.4.2 Virtual-Token optimization

The Eliminate-Duplicates optimization does not address the problem that a proposal can be prematurely terminated when the proposer goes offline. In extremely offline-heavy situations, it is very likely that the proposer will go offline before finishing its proposal simply because it takes much longer to see a majority of admins come online. This forces proposals to start over repeatedly, thereby making no progress. In order to further improve performance, we need to increase the number of online admins. It is impossible to increase the number of admins physically online at a certain time but it is possible to increase the number of admins virtually online.

In our Virtual-Token optimization, admins that are going offline try to gracefully hand-off their virtual token to another admin (which we refer to as a token-holder). The virtual token allows the token-holder to act on the offline admin's behalf to accept PREPARE and ACCEPT requests. If we add up all the online admins as well as the virtual tokens that they are holding, the number of admins virtually online can increase dramatically. We give an example of the optimization:

1. Admin1 gracefully hands-off its virtual token to admin2 and goes offline.
2. Admin3 starts a proposal and sends a PREPARE request to admin2.
3. Admin2 replies for itself as well as for admin1 (as admin1's token-holder). Admin3 now has 2 replies instead of 1.

Obviously, the number of virtually online admins depends on the probability that an admin is able to gracefully hand-off its virtual token before going offline. If we have perfect hand-off probability, then all admins are always virtually online if at least one admin is physically online. That one physically online admin would act as the token-holder for all the other offline admins. In a perfect hand-off situation with at least one physically online admin, there is always a majority online and therefore, Paxos would most likely succeed. However, we do not expect to achieve anywhere close to perfect hand-off since admins can crash suddenly and be unable to hand-off its virtual token. We will show in simulations that this optimization offers significant improvements even when hand-offs are not perfect.

Each virtual token is identified by a globally unique *tokenID*, which is assigned by the token-owner. After receiving an owner's token, token-holders log the tokens that it is holding onto disk so that it can recover them in the event of a failure. All virtual tokens are only valid for the current *view*. When a new primary is elected, all tokens of the previous *view*s are considered void and not accepted during future Paxos proposals. We will discuss the reasoning behind this rule below.

When an admin has handed-off its virtual token, it cannot participate in the Paxos protocol until it has reclaimed its token because otherwise it would be able to vote more than once in Paxos. For this reason, token-holders should return the tokens to their owners as soon as possible by periodically checking whether the owners of the held tokens have come back online. The reason that the token-owners can not directly ask for the return of their tokens when they come online is that they do not know which node is holding its token since its token may have been handed-off multiple times. When returning a token to its owner, the token-holder specifies the *tokenID* that it is returning to prevent a token-owner from confusing a delayed return message of a prior token for the return message of the current token.

When a token-holder goes offline, it tries to gracefully hand-off its token and the tokens that it is holding to another admin. If a token-holder goes offline abruptly, then this situation is the equivalent of multiple admins going offline at once. It is important to load balance the number of tokens held by each admin to reduce the effect of it going offline. Therefore, when an admin is handing-off its virtual token, it should always hand-off to the admin carrying the least number of tokens. To load balance tokens, admins exchange a list of tokens that they are holding during gossip.

If a token-holder crashes, any admin whose token it is holding will be unable to participate in Paxos until that token-holder comes back online so that the admin can reclaim its token. If the token-holder never comes back online, it becomes impossible for the token-owners to reclaim their tokens. If Paxos succeeds in creating a new *view*, the admins that lost their tokens in the previous *view* can once again participate in Paxos in the new *view*. However, if a majority of tokens are lost, Paxos cannot succeed because there will never be a majority of admins virtually online. In the event that Paxos becomes impossible, we allow a manual restart of the system. Essentially, an admin initiates a manual restart by creating a new repository and copying the data of the old repository. The admin, who becomes the new primary, then invites all the members from the old repository to join the new repository.

With these changes, running Paxos with the Eliminate-Duplicates and Virtual-Token optimizations out-performs basic Paxos in almost all situations. We will be employing the Paxos protocol with both optimizations in our metadata replication scheme whenever the primary is offline for a duration exceeding the *timeout*.

Chapter 5

Implementation

This chapter describes the implementation details behind Friendshare. We discuss installation details, repository creation, and repository management.

5.1 Implementation overview

Friendshare is implemented in Java and requires Java 1.5 or higher. It can be run as a daemon and is compatible with various operating systems. Friendshare is easily installed and setup with a built-in installer that was developed using IzPack, a custom Java installer toolkit [19]. The installer allows users to choose the installation directory and setup configuration parameters used in Friendshare and Friendstore. Users can create, join, and manage repositories through the Friendshare GUI, which is implemented using Java Swing.

The Virtual-Token optimization requires admins to hand-off their virtual token before going offline. Friendshare uses the Java Service Wrapper [20] to implement shutdown hooks for graceful virtual token hand-off. The wrapper attempts to call the shutdown method whenever the user performs shutdown or if the Friendshare application is closed.

When peer nodes are behind NATs, connection attempts may be blocked by the NAT. In order to get around this issue, Friendshare uses NUTSS, a Java-based NAT traversal technique, to punch a hole through the NAT [15]. The basic idea of NAT traversal is that when a connection-request packet is sent out through the NAT, the NAT opens a hole for the expected reply from the recipient. Unfortunately, a recipient behind a NAT will not receive the initial request packet. However, an exposed third-party can be used to coordinate the connection protocol. First, the connector notifies the third-party and sends the connection-request packet to the recipient, which will be blocked by the recipient's NAT. The third-party will then notify the recipient and instruct it to send a connection-request to the connector. Since the first step will have opened a hole in the connector's NAT, the recipient's connection-request will be successfully received by the connector.

5.2 Repository management

This section deals with details in repository management. In particular, we discuss how the repository information and metadata is stored on a user's hard disk, as well as the details behind creating a repository, joining a repository, approving new members, promoting members to admins, and evicting uncooperative members.

5.2.1 Storing repository on hard disk

The list of a user's repositories and the repositories' metadata (if the user is an admin) must be stored on the user's hard disk so that it can be reloaded when the user restarts its computer. The list of repositories is stored in a single file `Repositories.data`, which includes the repository ID, last-known membership list of the repository, and the last-known role of each member. The membership list is used to find an admin when the user wants to read or write in the repository. Since the membership list changes periodically, the user may need to update the list from time to time. Each repository of which the user is an admin has its own separate metadata file, which includes the current view of the repository, tentative and committed write logs, and version vector.

5.2.2 Creation

Users can create new repositories by using the GUI as shown in Figure 5.1. The GUI will prompt the user to enter a repository name, which will be concatenated with the user's *userID* to generate the *repositoryID*. A listing for the new repository is then added to `Repositories.data`. Since the creator user is the sole member of the repository at creation time, it is designated as the primary. The repository *view* is initialized to view number 0.

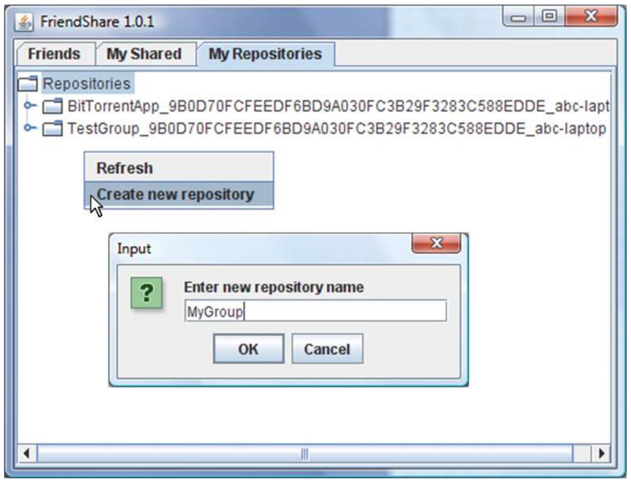
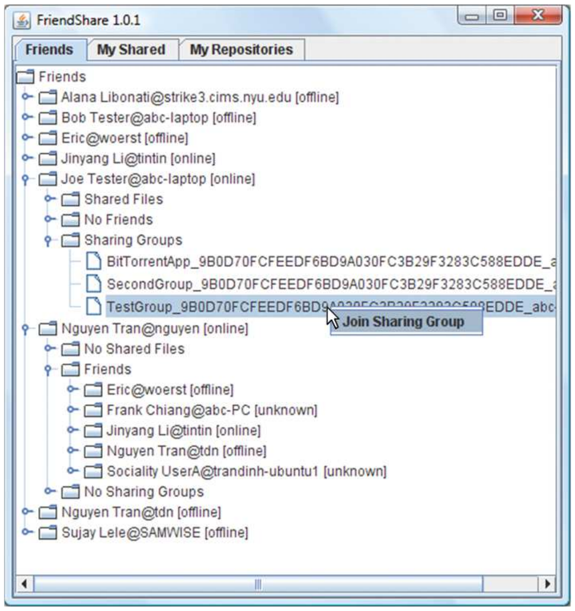

FriendShare 1.0.1
Friends
My Shared
My Repositories
Repositories
BitTorrentApp_9B0D70FCFEEDF6BD9A030FC3B29F3283C588EDDE_abc-lapt
TestGroup_9B0D70FCFEEDF6BD9A030FC3B29F3283C588EDDE_abc-laptop
Refresh
Create new repository
Input
Enter new repository name
MyGroup
OK
Cancel

Figure 5.1: Creating a new repository

5.2.3 Joining

After a repository has been created, other users can request to join it. Figure 5.2 shows a user trying to join the TestGroup repository. As discussed in Chapter 4, a user requests to join another repository by sending a REQUEST_JOIN request to its friend who will then forward the request to one of the admins in the repository.

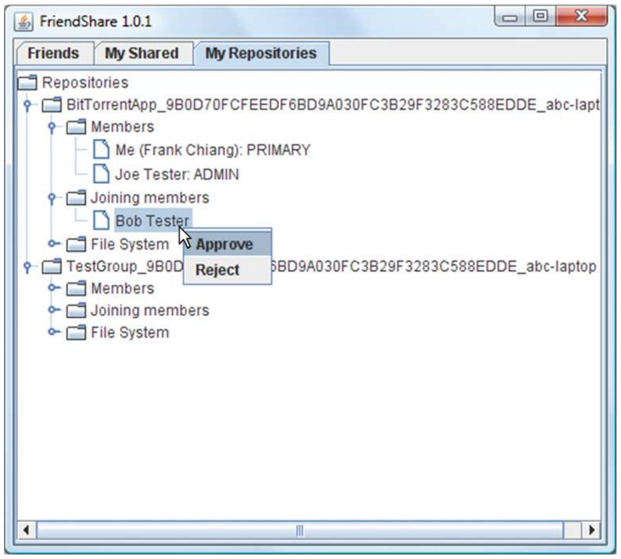
The join request will eventually reach the primary through gossiping and is displayed in the GUI. The primary can then choose whether to approve or reject the join request as shown in Figure 5.3.


FriendShare 1.0.1

Friends | My Shared | My Repositories

Friends
- Alana Libonati@strike3.cims.nyu.edu [offline]
- Bob Tester@abc-laptop [offline]
- Eric@woerst [offline]
- Jinyang Li@tintin [online]
- Joe Tester@abc-laptop [online]
  - Shared Files
  - No Friends
  - Sharing Groups
    - BitTorrentApp_9B0D70FCFEEDF6BD9A030FC3B29F3283C588EDDE_a
    - SecondGroup_9B0D70FCFEEDF6BD9A030FC3B29F3283C588EDDE_a
    - TestGroup_9B0D70FCFEEDF6BD9A030FC3B29F3283C588EDDE_abc
- Nguyen Tran@nguyen [online]
  - No Shared Files
  - Friends
    - Eric@woerst [offline]
    - Frank Chiang@abc-PC [unknown]
    - Jinyang Li@tintin [online]
    - Nguyen Tran@tdn [offline]
    - Sociality UserA@trandinh-ubuntu1 [unknown]
  - No Sharing Groups
- Nguyen Tran@tdn [offline]
- Sujay Lele@SAMWISE [offline]

Figure 5.2: Joining a repository


FriendShare 1.0.1

Friends | My Shared | My Repositories

Repositories
- BitTorrentApp_9B0D70FCFEEDF6BD9A030FC3B29F3283C588EDDE_abc-lapt
  - Members
    - Me (Frank Chiang): PRIMARY
    - Joe Tester: ADMIN
  - Joining members
    - Bob Tester
  - File System
- TestGroup_9B0D 6BD9A030FC3B29F3283C588EDDE_abc-laptop
  - Members
  - Joining members
  - File System

Figure 5.3: Primary approving new members

5.2.4 Managing

In the Repositories tab of the GUI, users can perform management actions on repositories of which they are members, such as downloading files, sharing new files, deleting files, making new directories, and removing directories, as shown in Figures 5.4 and 5.5. When a user downloads a file, Friendshare connects to one of the replicas storing the file to download. If a user wants to share a new file into a specific directory in the repository, a file browser window is displayed, allowing the user to select a file to share (see Figure 5.5). Friendshare then creates an ADD write request that specifies that the selected file should be shared into the selected directory, and the request is sent to an admin. Similarly, deleting files, making new directories, and removing directories cause DELETE, MKDIR, and RMDIR write requests to be sent to an admin.

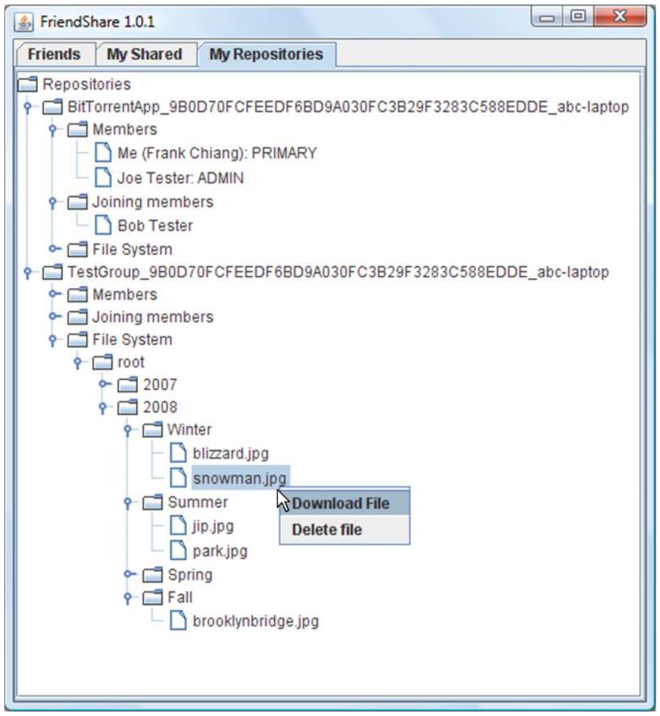
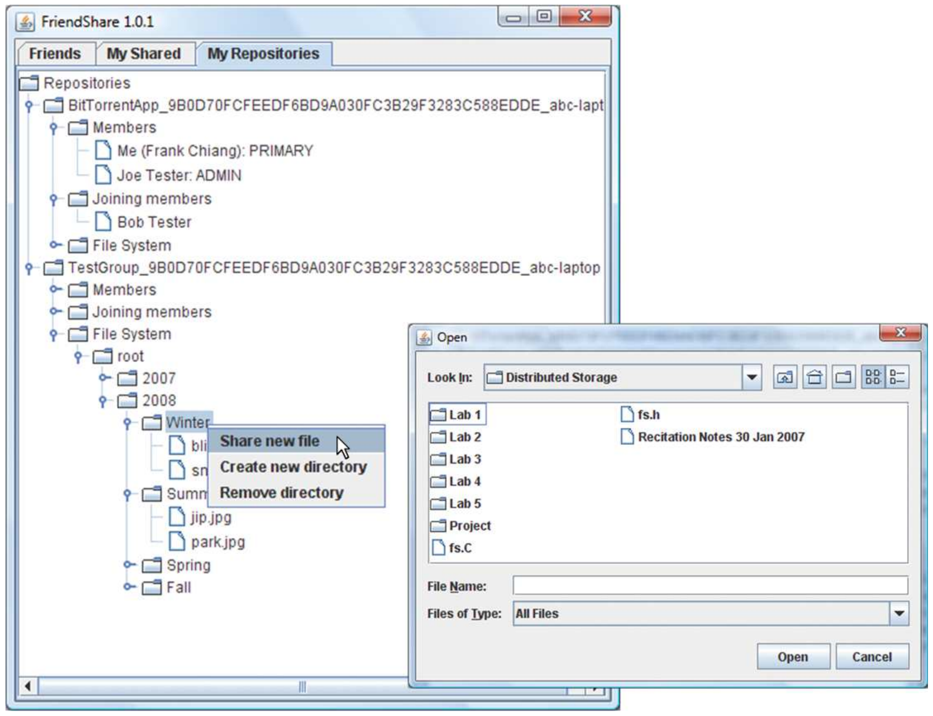


Figure 5.4: Downloading and deleting files

To increase the replication factor of the repository metadata, the primary can promote a member to become an admin. In order to accomplish this, the primary



Look In: Distributed Storage

Files of Type: All Files

Figure 5.5: Directory actions

writes a PROMOTE_MEMBER entry in its write log and commits it. This write will be passed to all the admins in the repository through gossiping and the member's status will be changed from MEMBER to ADMIN. The member will discover its promotion when one of the admins gossips with it.

If a member is malicious or disruptive in some way, the primary can evict the member from the repository group. Similar to admin promotions, the primary writes a KICK_MEMBER entry in its write log and commits it, which will be gossiped to the other admins. The evicted member is thus removed from the membership list and therefore unable to write to the repository anymore.

Chapter 6

Evaluation

This chapter presents our simulations of Friendshare's performance in metadata replication and consensus protocols, focusing on the effects of a deployment environment where nodes have low resources and low availability. The evaluation focuses on two important performance measures. First, how quickly can a write be committed and become visible to the nodes in the system? Second, if the primary leaves the system, how quickly can a new primary be elected to form a new *view*?

6.1 Simulation setup

To conduct the simulations, we built an event-driven simulator in Java. Our simulation experiments consist of 20 admins, where each admin would randomly go online and offline. The online and offline durations are controlled by 4 variables: ONLINE_MIN, ONLINE_MAX, OFFLINE_MIN, and OFFLINE_MAX. For each admin, the simulator randomly chooses a value between ONLINE_MIN and ONLINE_MAX to determine how long the admin will stay online before going offline. Similarly, each admin stays offline for a random duration chosen between OFFLINE_MIN and OFFLINE_MAX. Setting different values for these 4 variables allows us to control the average online percentage of the admin. When online, each admin picks a random admin to gossip with every 60 seconds to exchange version vectors and writes.

6.2 Basic write performance

After a write is accepted by an admin in a repository, it is propagated to the other admins through gossiping and eventually committed by the primary. After some time, all admins in the repository will learn that a write is committed and at this point the write is "consistent" because the committed write logs of all admins will be identical up to and including the consistent write.

From our discussion above, we are interested in the rate that writes are promoted:

1. How quickly is a write committed?

2. How quickly will a committed write become consistent? This happens when all admins know that a write is committed.

To measure write performance, the simulator periodically generates new writes at random intervals, with the average of one write every 15 minutes, and sends it to a random online admin in the repository. One of the admins is designated as the primary and is responsible for committing the writes in the repository. Reconfiguration is not included in this simulation as we are primarily interested in the rate at which writes become committed and consistent.

Figure 6.1 shows a comparison of the time elapsed before a write is committed and then becomes consistent as a function of the admins' online percentages. When admins have low online availability, write promotion takes longer. This is expected because an admin needs to wait until offline admins come online before it can gossip new writes to them. From the simulations, we find that writes are committed in several minutes when the primary is online but in a few hours when the primary is offline. Low availability also plays an important role in the rate that a write becomes consistent, since it must be seen by all of the admins. Therefore, if there are not enough admins online, a write must wait until more admins come online before it can be promoted.

In order for a new write to be accepted into the repository, at least one admin must be online. Figure 6.2 shows that the repository service is almost always available to accept new writes at reasonable node availabilities. However, when admins are online

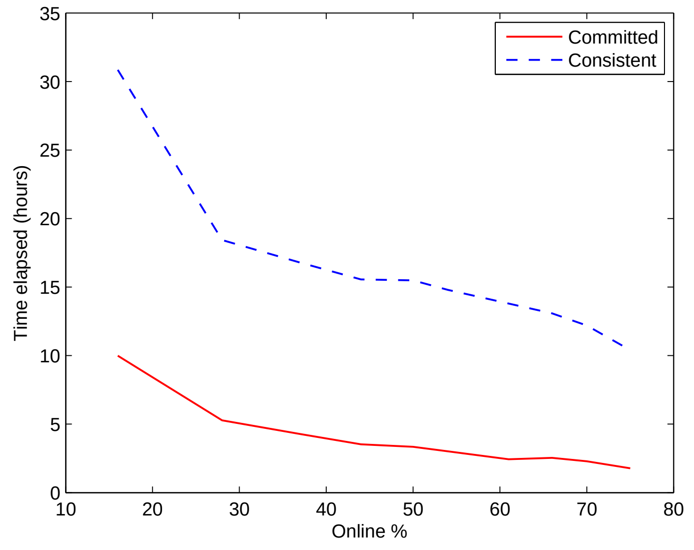


Figure 6.1: *Time elapsed before a write is committed and consistent at various online percentages.* The online percentages are varied by setting OFFLINE_MIN to be 2 hours and OFFLINE_MAX to be 8 hours while varying ONLINE_MIN and ONLINE_MAX.

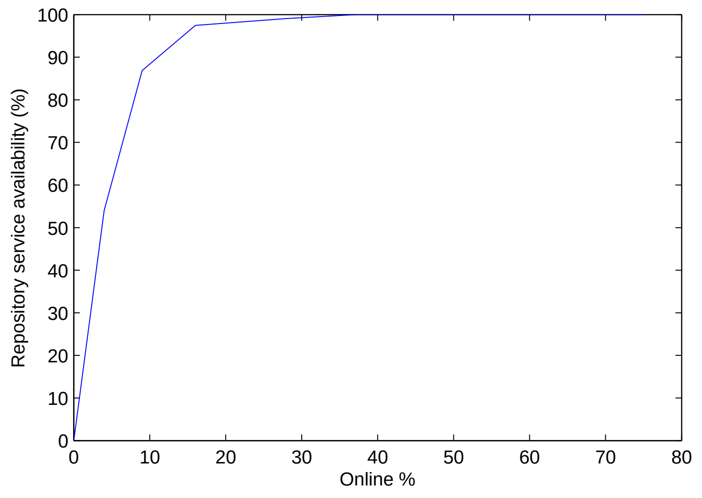


Figure 6.2: *Probability that at least one admin is online when a new write is created (i.e. Repository service availability) at various online percentages.* The online percentages are varied by setting OFFLINE_MIN to be 2 hours and OFFLINE_MAX to be 8 hours while varying ONLINE_MIN and ONLINE_MAX.

less than 20% of the time, the repository service is at a greater risk of being unavailable (no admins online to accept a new write). To make the overall repository service more available, we should increase the number of admins in the system. However, the tradeoff is that increasing the number of admins also increases the time taken for committed writes to be promoted to the consistent state.

6.3 Reconfiguration performance

Friendshare runs Paxos to elect a new primary when the old one has timed out. As mentioned in Chapter 4, Paxos requires high node availability. In this section, we discuss our experiences in running Paxos with low node availability. We also present performance statistics of our optimizations that were discussed in Chapter 4.

As in 6.2, the experiments include 20 admins that randomly go online and offline. Each admin keeps track of the last-online-time of the primary. If an admin detects that the primary has been offline for more than 30 minutes (*timeout*), it starts a new Paxos proposal to elect a new primary. We chose a short *timeout* in order to allow the simulations to finish faster. In real world deployment, the *timeout* should be set to a much larger value (e.g. 1 day). A longer *timeout* would follow the same trend but would have a different absolute value. After running the simulation for a while, the simulator manually kills the primary. We stop the simulation when a new primary has been elected. In the event that the simulation has exceeded the maximum threshold time of 3 days without reaching an agreement, the simulation is recorded as a failure.

Figure 6.3 shows the time that it takes for the basic Paxos protocol to succeed in reaching an agreement at various online percentages. We find that as the admin availability drops below 50%, Paxos' duration begins to increase dramatically. This is because basic Paxos requires at least a majority of admins to be online simultaneously for a proposal to succeed. As the admin availability drops below 50%, it becomes less likely that a majority of admins will be online at one time. If a majority of admins is not online, the proposer must wait for admins to come online. Since this increases the proposal duration, there is a higher risk of proposal conflict. Figure 6.4 shows that simulations begin to fail (i.e. exceed maximum threshold time) as the online

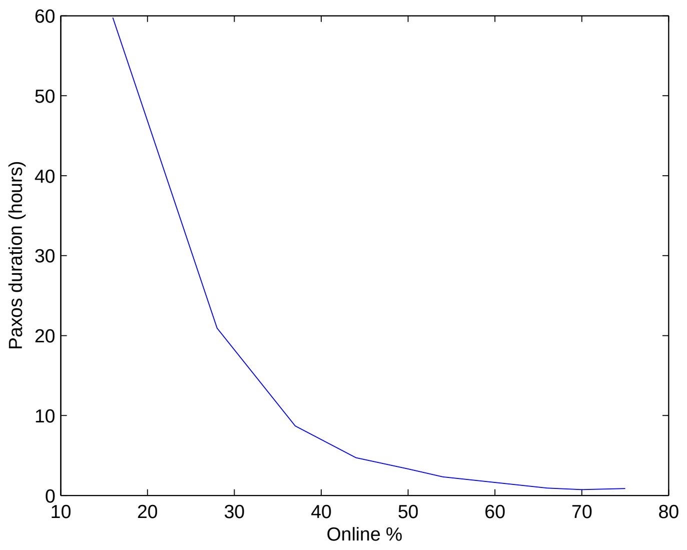


Figure 6.3: *Time elapsed before consensus is reached by basic Paxos at various online percentages.* The online percentages are varied by setting OFFLINE_MIN to be 2 hours and OFFLINE_MAX to be 8 hours while varying ONLINE_MIN and ONLINE_MAX.

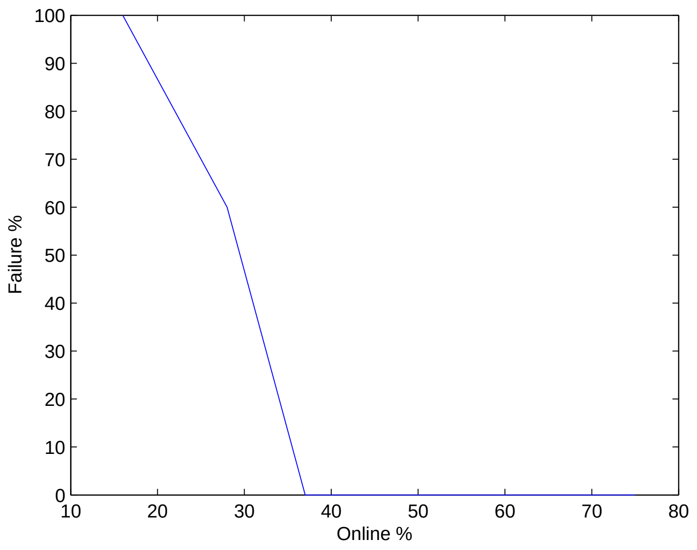


Figure 6.4: *Probability that a simulation fails in basic Paxos at various online percentages.* The online percentages are varied by setting OFFLINE_MIN to be 2 hours and OFFLINE_MAX to be 8 hours while varying ONLINE_MIN and ONLINE_MAX.

percentage drops below 40%.

Next, we compare the performance of basic Paxos with our optimizations. We run four simulations side-by-side: basic Paxos, Paxos with the Eliminate-Duplicates optimization, Paxos with the Virtual-Token optimization, and Paxos with both Eliminate-Duplicates and Virtual-Token optimizations. As discussed in Chapter 4, Eliminate-Duplicates tries to prevent new proposals if there is already an existing proposal and Virtual-Token allows admins to hand-off their virtual token before going offline, allowing other admins to operate on their behalf. We do not expect that admins will always be able to hand-off their virtual tokens in real-life so we include a variable TOKEN_HANDOFF_PROBABILITY which specifies the probability that an admin will be able to hand-off its token before going offline.

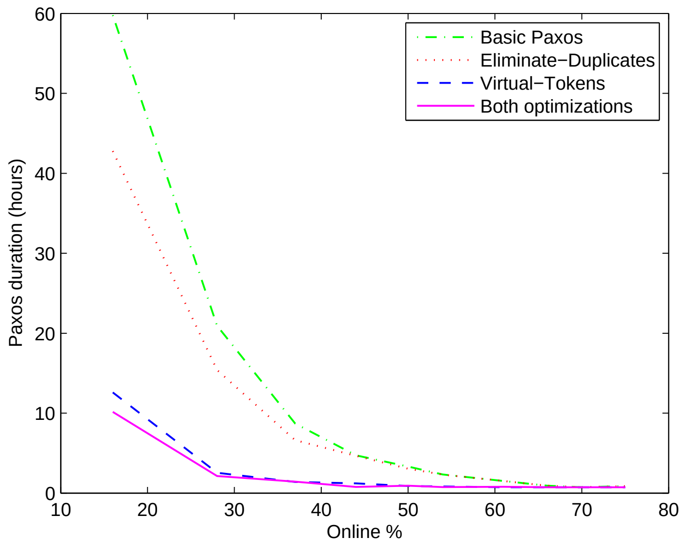


Figure 6.5: *Optimizations comparison of Paxos duration at various online percentages.* The online percentages are varied by setting OFFLINE_MIN to be 2 hours and OFFLINE_MAX to be 8 hours while varying ONLINE_MIN and ONLINE_MAX. For the Virtual-Token optimization, admins hand-off their tokens 50% of the time.

Figure 6.5 shows a performance comparison of basic Paxos and the optimizations by comparing the time that it takes for a consensus to be reached. We set TOKEN_HANDOFF_PROBABILITY to be 50%. Our simulations show that the

Eliminate-Duplicates optimization performs slightly better than basic Paxos while the Virtual-Token optimization offers substantial performance improvements. Using both Eliminate-Duplicates and Virtual-Token optimizations together gives the best performance across the entire online percentage spectrum.

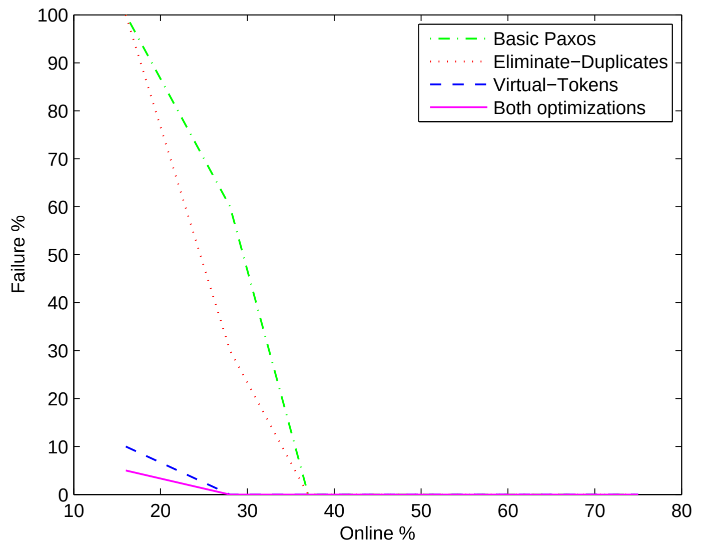


Figure 6.6: *Optimizations comparison of failure percentage.* The online percentages are varied by setting OFFLINE_MIN to be 2 hours and OFFLINE_MAX to be 8 hours while varying ONLINE_MIN and ONLINE_MAX. For the Virtual-Token optimization, admins hand-off their tokens 50% of the time.

When the admin availability drops to a certain level, Paxos has trouble reaching a consensus. If a consensus has not been reached after 3 days of running Paxos, that run is recorded as a failure. Figure 6.6 shows the probability that Paxos fails to reach a consensus at various online probabilities. Both basic Paxos and the Eliminate-Duplicates optimization begin to fail when admins are online less than 40% of the time. The Virtual-Token optimization maintains reasonable success probabilities even at low admin availabilities.

We are also interested in the effect that the admin's hand-off probability (i.e. TOKEN_HANDOFF_PROBABILITY) has on the performance. In Figure 6.7, we compare the time taken to elect a new primary using the Virtual-Token optimization

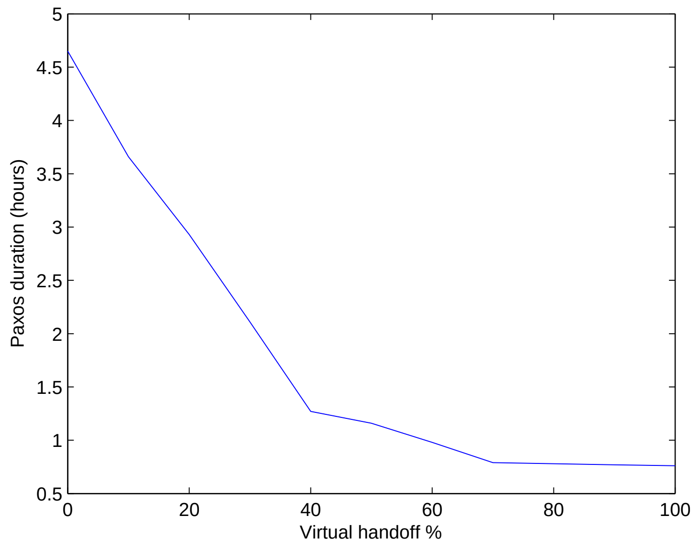


Figure 6.7: Time elapsed before consensus is reached by Virtual-Token optimized Paxos at various token hand-off percentages

at various TOKEN_HANDOFF_PROBABILITY values. The online percentage is set to 50% by setting both ONLINE_MIN and OFFLINE_MIN to be 2 hours and both ONLINE_MAX and OFFLINE_MAX to be 8 hours. From the simulations, we find that the Virtual-Token optimization provides reasonable performance even if admins only hand-off their virtual tokens 40% of the time. None of the simulation runs failed, which is expected since Figure 6.6 shows that an online percentage of 50% does not cause any failures.

Chapter 7

Related Work

Shared repositories are currently implemented on centrally managed sites, such as Facebook [10], Wikipedia [40], and Flickr [11]. However, centralized repositories have privacy issues, censorship restrictions, and a limited variety of features based on profitability. Friendshare offers a decentralized solution that allows multiple users to write and modify a common repository.

Network data storage has been visited by many previous systems. Grapevine [6] was one of the first systems to offer weak consistency in message delivery. NFS [33] allows a user to transparently access files on remote machines. AFS [17] later improved NFS's performance by employing leases and caching. Coda [34] replicates data to improve availability and allows all servers to accept writes. Harp [24] uses the primary copy scheme, in which a single primary node is responsible to serialize the writes to the system. These systems are generally "two-point" where there are dedicated clients and servers.

Ivy [28] was the first read/write file system that allows multiple writers. Each Ivy user has its own log, which is stored in a DHash distributed hash table. In this way, the logs can be accessed even when the user is offline. Other examples of multiple writer systems include Bayou [29, 37] and Farsite [1]. Bayou is a distributed replication system that ensures eventual consistency by using the *primary commit* scheme, in which the primary specifies the ordering of writes. As long as the primary is online, the system is capable of accepting and committing writes. If the primary is offline, the nodes can still accept writes although they must wait for the primary to come back online before writes can be committed. Bayou, as well as Ivy and TACT [41], employ log propagation schemes to synchronize the replica data.

When exchanging updates, it is more efficient for a node to send only the incremental changes to another node. To achieve this, systems such as LOCUS [30], Coda, and FICUS [32] use version vectors to specify the writes that they have seen so far. In addition, version vectors are used to detect and resolve any concurrent conflicts that may occur during writing.

LOCUS, Coda, FICUS, Harp, and Bayou replicate data by storing a complete copy on every node. However, this involves performing expensive replica synchronization on large data sets. Both PRACTI [9] and Farsite optimize replica synchronization by separating the metadata from the data. This separation allows the synchronization to only be done on the smaller metadata.

If data objects are mutable, the system must either consume large amounts of bandwidth to aggressively synchronize replicas or lazily propagate data modifications. To eliminate the difficulties of updating data, systems such as Farsite, SWALLOW [31], and Amoeba [27] restrict data objects to be immutable. Immutable data objects prevent the system from continuously synchronizing every time the data is modified. Instead, the old data object is deleted and a new data object is created when updating modified data.

One aspect of distributed data storage that we do not discuss in great detail in this thesis is data coding for efficient storage. Friendshare relies on Friendstore to handle the data object replication. The common options of data storage are to either generate complete replicas or perform data coding, which consists of either striping or erasure coding. When striping, data is split up into chunks and distributed across the nodes in the system. All of the chunks are required to reconstruct the original data. Erasure coding overlaps the data during chunking, which allows reconstruction even when some chunks are not available. Zebra [16] and Myriad [8] are examples of systems that use erasure coding for data storage.

Friendshare's metadata replication is based on Bayou because its *primary commit* scheme allows the system to operate in low availability environments. One problem with Bayou is that it cannot tolerate the primary leaving the system forever unexpectedly. Unfortunately, this could be a situation that happens quite frequently in peer-to-peer systems. Generally, nodes do not stay online for very long and once they go offline, they may never come back online (e.g. they decided to uninstall the application or their hardware died). Even if an admin does return online, a long time may have elapsed since it was last online. Many things may have happened: the user may have gone on vacation, their computer may have been in repair, or perhaps they simply did not run the application. It is not reasonable to expect the repository to wait and hope for the primary to come back online before committing any new writes. In contrast to Bayou, Friendshare is capable of reconfiguring repositories if the primary leaves the system by running a consensus protocol to elect a new primary.

Friendshare's consensus protocol is an optimized version of the Paxos protocol [22, 26]. The Paxos protocol consists of 3 phases, PREPARE, ACCEPT, and ANNOUNCE, where the PREPARE and ACCEPT phases require acceptances from a majority of nodes. Paxos does not guarantee liveness and our simulations show that as nodes become more heavily offline, the protocol's success rate decreases. To increase the success rate of Paxos, we contribute optimizations and show the effects of the optimizations with simulations.

Chapter 8

Conclusions

This thesis presents Friendshare, a decentralized data repository that can be created and organized by multiple users. Friendshare provides high data availability, durability, and consistency in peer-to-peer environments even when peer nodes have limited bandwidth and storage space, low node availability, and may occasionally leave the system unexpectedly.

Separating the metadata from the data allows Friendshare to provide efficient metadata replication across the privileged admin nodes. Members write in the repository by creating data objects from files and issuing write requests for those data objects to the admins in the repository. Friendshare ensures eventual consistency by employing the *primary commit* scheme. Admins tentatively accept the writes and propagate it to the other admins through gossiping. The primary eventually commits the write to stabilize the total-ordering of the repository.

If the primary leaves the system unexpectedly, the remaining admins reconfigure the repository by running Paxos, a consensus protocol, to elect a new primary. We contribute two optimizations, Eliminate-Duplicates and Virtual-Token, for the Paxos protocol to improve performance in low availability networks. In the Eliminate-Duplicates optimization, we reduce the number of proposals by preventing new proposals when there is an existing proposal. In the Virtual-Token optimization, we virtually increase the number of online admins. To accomplish this, admins pass their virtual token to another admin before going offline, which allows the other admin to act on its behalf in Paxos.

There are many possible future steps that can be taken. First, caching repository metadata can improve performance and availability so that the repository can be browsed without connecting to an admin. Second, the behavior of Friendshare should be studied when there are malicious members or admins. Finally, real world evaluations are needed to fully quantify the results in this thesis.

Friendshare is written in Java and can be run as a daemon on multiple operating systems. The bundled installer allows for easy deployment. Friendshare offers a flexible API that allows a wide variety of different applications to be built on top of the Friendshare system.

Bibliography

[1] Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. Farsite: federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev. 36*, SI (2002), 1–14.

[2] Amazon, 2008. `http://aws.amazon.com/s`. [Online; accessed 03-April-2008].

[3] Aspan, M. How sticky is membership on Facebook? Just try breaking free. *The New York Times* (Feb. 2008).

[4] Baliga, A. Data replication: weak consistency is a strong paradigm! *Rutgers University Technical Report* (2006). `http://www.research.rutgers.edu/~aratib/presentations/weakconsistency.pdf`.

[5] Bernstein, P. A., and Goodman, N. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. Database Syst. 9*, 4 (1984), 596–615.

[6] Birrell, A. D., Levin, R., Schroeder, M. D., and Needham, R. M. Grapevine: an exercise in distributed computing. *Commun. ACM 25*, 4 (1982), 260–274.

[7] Burrows, M. The Chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 335–350.

[8] Chang, F., Ji, M., Leung, S.-T., MacCormick, J., Perl, S., and Zhang, L. Myriad: Cost-effective disaster tolerance. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2002), USENIX Association, p. 8.

[9] Dahlin, M., Gao, L., Nayate, A., Venkataramani, A., Yalagandula, P., and Zheng, J. Practi replication for large-scale systems, 2004.

[10] Facebook, 2008. `http://www.facebook.com`.

[11] Flickr, 2008. `http://www.flickr.com`.

[12] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. *SIGOPS Oper. Syst. Rev. 37*, 5 (2003), 29–43.

[13] Google groups, 2008. `http://groups.google.com`.

[14] Gray, J., Helland, P., O'Neil, P., and Shasha, D. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (1996), pp. 173–182.

[15] Guha, S., Takeda, Y., and Francis, P. Nutss: a sip-based approach to udp and tcp network connectivity. In *FDNA '04: Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture* (New York, NY, USA, 2004), ACM, pp. 43–48.

[16] Hartman, J. H., and Ousterhout, J. K. The zebra striped network file system. *ACM Trans. Comput. Syst. 13*, 3 (1995), 274–310.

[17] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R. N., and West, M. Scale and performance in a distributed file system. *SIGOPS Oper. Syst. Rev. 21*, 5 (1987), 1–2.

[18] Isard, M. Autopilot: automatic data center management. *SIGOPS Oper. Syst. Rev. 41*, 2 (2007), 60–67.

[19] Izpack, 2008. `http://izpack.org/`.

[20] Java service wrapper, 2008. `http://wrapper.tanukisoftware.org/`.

[21] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[22] Lamport, L. Paxos made simple. *SIGACT News 32*, 4 (December 2001), 51–58.

[23] Lindsay, B. G., Selinger, P. G., Galtieri, C., Gray, J. N., Lorie, R. A., Price, T. G., Potzulo, F., and Wade, B. W. Notes on distributed databases. Tech. Rep. RJ2571(33471), IBM, San Jose Research Laboratory, 1979.

[24] Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., and Shrira, L. Replication in the Harp file system. *SIGOPS Oper. Syst. Rev. 25*, 5 (1991), 226–238.

[25] Mazieres, D. *Self-certifying file system.* PhD thesis, 2000. Supervisor-M. Frans Kaashoek.

[26] Mazières, D. Paxos made practical. January 2007.

[27] Mullender, S. J., and Tanenbaum, A. S. A distributed file service based on optimistic concurrency control. In *SOSP '85: Proceedings of the tenth ACM symposium on Operating systems principles* (New York, NY, USA, 1985), ACM, pp. 51–62.

[28] Muthitacharoen, A., Morris, R., Gil, T. M., and Chen, B. Ivy: a read/write peer-to-peer file system. *SIGOPS Oper. Syst. Rev. 36*, SI (2002), 31–44.

[29] Petersen, K., Spreitzer, M., Terry, D., and Theimer, M. Bayou: replicated database services for world-wide applications. In *EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop* (New York, NY, USA, 1996), ACM, pp. 275–280.

[30] Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G., and Thiel, G. Locus a network transparent, high reliability distributed system. *SIGOPS Oper. Syst. Rev. 15*, 5 (1981), 169–177.

[31] Reed, D. P., and Svobodova, L. Swallow: a distributed data storage system for a local network. *International Workshop on Local Networks* (August 1980).

[32] Richard George Guy, I. *FICUS: a very large scale reliable distributed file system.* PhD thesis, Los Angeles, CA, USA, 1992.

[33] Sandberg, R., Golgberg, D., Kleiman, S., Walsh, D., and Lyon, B. Design and implementation of the sun network filesystem. 379–390.

[34] Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H., and Steere, D. C. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comput. 39*, 4 (1990), 447–459.

[35] TechCrunch. Facebook censors Ron Paul?, 2007. `http://www.techcrunch.com/2007/11/07/facebook-censors-ron-paul/`.

[36] TechCrunch. Is Facebook really censoring search when it suits them?, 2007. `http://www.techcrunch.com/2007/11/22/is-facebook-really-censoring-search-when-it-suits-them/`.

[37] Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. Managing update conflicts in Bayou, a weakly connected replicated storage system. 322–334.

[38] Tran, N., Chiang, F., and Li, J. Friendstore: Cooperative online backup using trusted nodes. *EuroSys Affiliated Workshop on Social Network Systems* (2008).

[39] Wikipedia. Criticism of Facebook, 2008. `http://en.wikipedia.org/wiki/Criticism_of_Facebook` [Online; accessed 04-April-2008].

[40] Wikipedia. Wikipedia, 2008. `http://en.wikipedia.org/wiki/Wikipedia` [Online; accessed 02-April-2008].

[41] Yu, H. Tact: tunable availability and consistency tradeoffs for replicated internet services. *SIGOPS Oper. Syst. Rev. 34*, 2 (2000), 33.

[42] Yu, H., and Vahdat, A. The costs and limits of availability for replicated services. *ACM Trans. Comput. Syst. 24*, 1 (2006), 70–113.