KUNGL
TEKNISKA
HÖGSKOLAN

Royal Institute of Technology
Dept. of Teleinformatics

Read-Write Replication in Arla

by
Noora Peura

IT (Institutionen för Teleinformatik)
Electrum 204
164 40   Kista

Deptartment of Teleinformatics
Electrum 204
Royal Institute of Technology
SE-164 40 Kista, SWEDEN

Read-Write Replication in Arla

by
Noora Peura

Master´s Thesis in Computer Science (20 credits)
at the School of Computer Science and Engineering,
Royal Institute of Technology year 2000
Supervisor at IT was Fredrik Lundevall
Examiner was Karl-Filip Faxén

Abstract

Arla is a free implementation of the distributed file system AFS. AFS supports read-only replication of files. To improve the functionality Arla could be equiped with read-write replication. The goal of this degree project is to study the possibilities of implementing read-write replication in Arla while keeping Arla compatible with AFS. Several replication algorithms and other file systems with read-write replication are studied for ideas. Options for implementing read-write replication in Arla are listed and two concrete protocols are recommended.

Sammanfattning

Arla är en fri implementation av det distribuerade filsystemet AFS. AFS stödjer läsreplikering av filer. För att föbättra funktionaliteten kan Arla utrustas med möjligheter till skrivreplikering av filer. Syftet med detta examensarbete är att studera möjligheterna till att implementera skrivreplikering i Arla med bibehållen AFS-kompatibilitet. Flera replikeringsalgoritmer och filsystem med skrivreplikering studeras. Flera alternativ för implementering av skrivreplikering i Arla beskrivs och två konkreta protokoll rekommenderas.

Contents

Chapter 1

Introduction

The file system is an important part of the operating system. It handles the storage and access of data on physical disk. It can also contain mechanisms for access control and other services for the user. To increase the performance and scalability of a file system it can be distributed. In distributed file systems one or several file servers store the files and clients can access the same files from any place in the network without knowing exactly which server or disk the files are stored on.

When a distributed file system grows the load on the file servers increases. There is also always a risk that a server fails and the data on that server becomes temporarily unavailable. A remedy for both high load and server failures is replication, which means that each file is stored on several servers simultaneously. Then users can get the same files from another file server if one server fails or gets too high load.

Many distributed file systems have support for replication of read-only files, since read-only files are seldom updated but frequently read. Some file systems have support for replication of all files, which takes a bit more effort. The contents of the files should be kept consistent, which isn't trivial. Also the propagation of updates to the other replicas can get costly if updates are frequent.

Arla is a clone of the distributed file system AFS. AFS implements read-only replication of files. To improve the functionality of Arla it could be equipped with read-write replication. The implementation would have to be transparent to the client, since AFS clients must be compatible with Arla servers and vice versa.

The goal of this degree project is to study the possibilities of implementing read-write replication in Arla. It includes studies of different replication algorithms, case studies of other file systems with read-write replication and a study of AFS and Arla. The reasonable options for implementing read-write replication in Arla will be presented together with recommendations. The final choice of implementation will be left to the Arla implementors.

1.1 Terminology

Since different authors use different terms for the same things it has sometimes been difficult to choose which terms to use. In the cases when a new algorithm is described I will often use the same terms as the authors used in their paper. For some more common terms I have tried to be consequent.

A server can often either mean a physical machine or a server process. Here a server is a server process, and a server machine is a physical machine that runs one or several server processes. In this paper a “node” or a “replica” is a server storing a replica of a certain file. Some may prefer to call it a “site”, but since that can nowadays have so many meanings I will not use it as a synonym to a replicating server.

A word which can often cause confusion is “partition”, since it can both mean a part of something or the event when something is split in several parts (or rather partitions). Also, in a computer environment, a partition can either be a part of a disk, a network being split up in two or a part of a split-up network. In this paper the partitions are mostly about networks, but there are also a couple of references to disk partitions. When a part of a network is meant, the word “partition” is used. The partition event is mostly referred to as a “network partition”. A partition of a hard disk is a “disk partition”. Hopefully also the context can help deciding which one is meant.

Chapter 2

Plan

This report starts with an introduction including a section about terminology. Chapter two, that you are currently reading, is a plan of the report. Chapter three is a brief introduction to distributed file systems for those readers who wish to refresh their memories. Chapters four and five are introductions to AFS and Arla respectively. Thereafter follows the study of read-write replication in general in chapter six and replication algorithms in chapter seven. Chapter eight briefly describes some efforts to evaluate, compare and optimize algorithms. Case studies of other replicating file systems can be found in chapter nine. In chapter ten the possibilities for implementing replication in Arla are discussed. My recommendations can be found in chapter eleven. Finally there is an appendix with some pseudo-code for the recommended replication algorithms.

Chapter 3

Introduction to Distributed File Systems

3.1 Overview

The file system is a part of the operating system and handles file access. The user often sees the files represented by file names and ordered into hierarchical directories. When a user opens a file the file system finds the file in the storage system and returns a file handle to the user so that the user can access the data. The file system takes care of all the technicalities that the user doesn't want to know about. Many file systems also handle things like security and access control.

Most modern operating systems have a built-in file system. There are also many file systems which can be used in many different operating systems. These can work above the built-in operating system or parallel to it, more or less replacing it.

A distributed file system is basically a file system where the files can be transparently stored on several servers instead of one. The user doesn't have to have any idea of which server physically stores the files he or she currently is accessing. The user still sees a collection of file names ordered in hierarchal directories while the files actually can be stored on a collection of servers all across the world.

The advantages of distributed file systems are many. First of all a distributed file system is more scalable. One single server cannot handle as many access requests as a group of servers. Second, availability increases. If one server goes down only a part of the file system becomes unavailable. With mechanisms like replication the availability can be increased even more. If one server goes down the data can be accessed from other replicas. Advantages can also be gained when a site is geographically widely spread by distributing the servers to different geographical locations. The users see advantages like being able to access their files from any workstation at their site and (hopefully) better availability.

There are also many difficulties in the design of distributed file systems. To meet the increased demand for avaiability and scalability the file system needs to be more complex. Network load and network failures affect the access times more than if all users had their data stored only on their workstations. Caching and replication schemes can create consistency problems depending on how updates are propagated and how changes are detected. Often choices have to be made between consistency, availability and access times.

3.2 Files and Directories

The data in a file system is stored in files. The file usually has a readable file name that the user can use to refer to the file. Most file systems also keep internal file ID:s for the files. Also most file systems store bits of additional information - metadata - like when the file was last changed, a version number, the owner of the file etc.

Most file systems organise the files into directories. Directories are special objects which can contain files or other directories, forming a fully connected directory tree. The directory tree has a single starting point, called the root directory.

Insertion of a file into a directory is usually called *linking*. In some systems linking and unlinking is automatic and a file can only be linked into one directory. In other systems a file can be linked to several directories, which gives the impression that the file resides in several directories at the same time. If a file is unlinked from all directories it resides in it is removed.

Distributed file systems and also some non-distributed file systems use *mounting* (see figure 3.1 and figure 3.2) to build the complete directory tree from subtrees. Administrative units like disk partitions or volumes contain each a directory tree, which can be mounted into the file system to form a complete tree. The first disk partition to be mounted contains the root directory and is often called the root partition. Some file systems have a whole set of disjoint trees, each with separate roots. The position in the directory tree where a subtree is mounted is called a mount point. Many file systems allow a subtree to be mounted at several mount points at the same time.

In a distributed file system the directory tree can contain subtrees that physically reside on different servers. In a correctly designed distributed file system the user does not have to know the difference between a mount point and a "ordinary" directory or between mounted subtrees residing on different disks. All the user has to know is the file's name and position in the directory tree. This is what makes a collection of servers running a distributed file system different from a collection of servers with non-distributed file systems

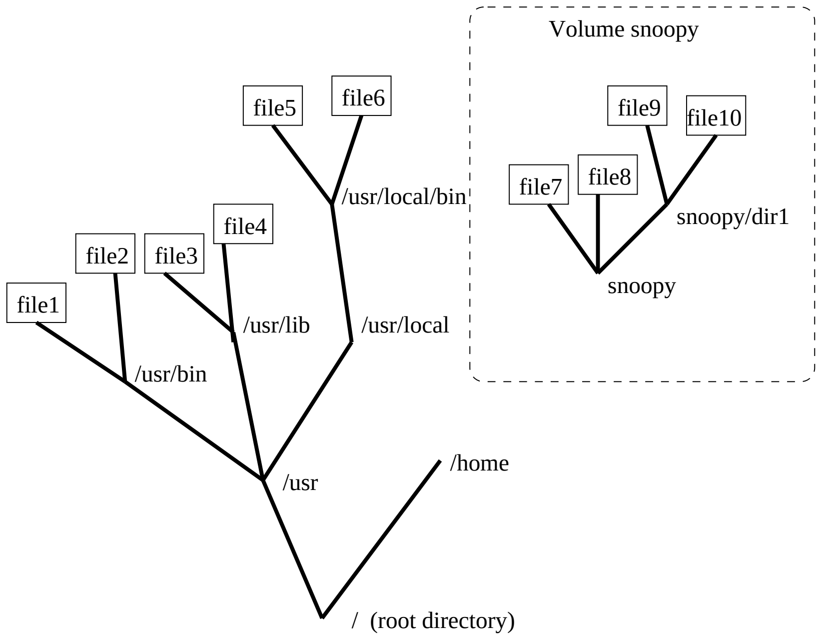

Figure 3.1. Example file system before the mounting of the directory `snoopy`.

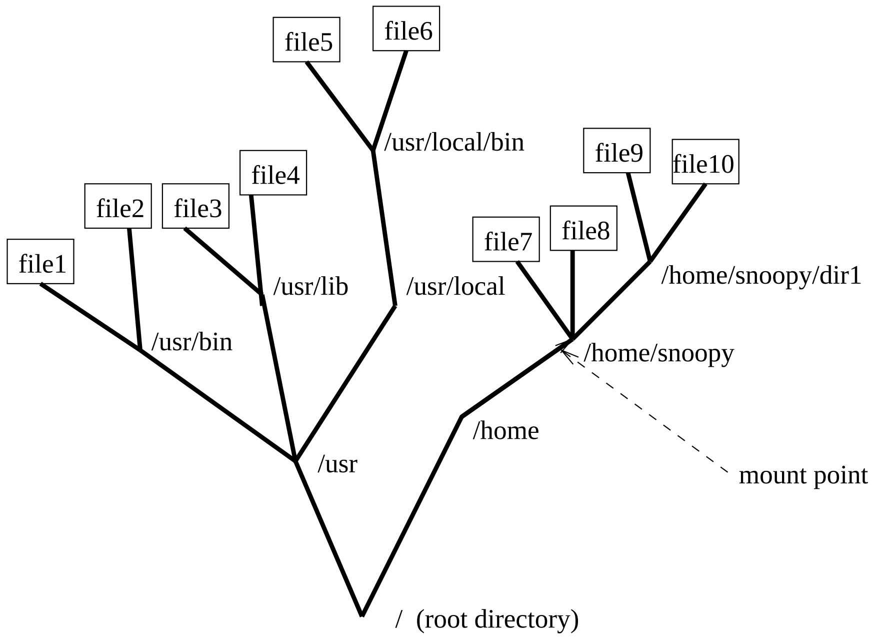

Figure 3.2. Example file system with the directory `snoopy` mounted into `/home`.

- in a non-distributed file system the user would have to indicate which server to access in addition to the file name and position in the directory tree. This makes it possible to move a subtree in a distributed file system transparently from one server to another.

3.3 Clients and Servers

Distributed file systems are usually based on clients and servers. Central servers store the files on disks and communicate with clients residing on the workstations. The clients receive the file system calls from the users or applications and forward the requests to a server. When the server has replied the client forwards the reply to the user or application that issued the file system call.

Some clients use caching to increase performance. When a file is first accessed it is kept in the cache on the client, so that subsequent accesses to the file can be served without calling the server. When caching is used there must be a way of notifying the clients when a cached file has been changed on the server. This is done in different ways in different file systems. There are also many different policies for when to send file updates to a server. Some file systems send updates at certain intervals while others send them right away. Some only send changes when the file is closed.

3.4 Security

In multi-user file systems security is often achieved by giving each user a user name and associating all files with their owners' user names. The files can have different levels of access for different users or user groups. The users can set access rights for their files to allow other users to read or modify them.

Many modern operating systems use some kind of authentication protocol and encryption for security. Some protocols like Kerberos are used both for login and for identification when files are accessed.

3.5 Reading

More about file systems can be found in [29]. [24] is a survey of the distributed file systems in 1989. Much has changed since then, but the survey is still interesting as a history lesson. A fresher survey can be found in [26].

Chapter 4

Overview of AFS

4.1 History of the Andrew File System

The construction of the Andrew File System started in 1982 at the Information Technology Center (ITC), a collaboration between Carnegie-Mellon University (CMU) and IBM. The goal of ITC was to develop an inexpensive distributed computing environment for CMU. The project was called Andrew after the two benefactors of CMU, Andrew Carnegie and Andrew Mellon. The Andrew File System was one part of the extensive project.

The Andrew system was designed as a crossing between a time-sharing system with a common file system and communication between users, and personal computing with constant and high performance. Users should be able to have the same working environment wherever they are, but still use the local workstation for doing the computations, thus combining the benefits of personal computing and time-sharing systems.

The demands on the file system were high. It had to have good security and authentication and the Andrew system had to be scalable to a very high number of nodes. Since no existing file system was found good enough the Andrew File System was created. When the Andrew project's progress was presented in [15], the file system was an important part of the project.

The Andrew File System was eventually renamed to its short form, AFS. There has been three versions of AFS: AFS-1, AFS-2 and AFS-3. When the development of AFS-3 was started the project was taken over by a spin-off company called Transarc. AFS is now a commercial product and there are over 150 AFS cells all around the world connected to the public AFS tree.

4.2 AFS Architecture

The AFS system is basically a client-server system with clients running at the user workstations and a group of servers with different tasks running on server machines. It is a transparent distributed file system where the files are stored on a set of file servers and can be accessed by the user without knowledge of where the file is physically stored.

Traditionally AFS is run on UNIX based systems, but has also been ported to other systems like Windows NT.

The following sections describe briefly the basic architecture of AFS and the functions of the many servers in AFS. A more detailed description can be found in [30].

4.2.1 Cells

The name space of AFS is divided into cells. Each independent administrative organization that runs AFS has its own cell. A cell contains a set of client and server machines which belong exclusively to that cell.

A cell can be connected to the common world-wide name space (public cells), in which case any user from any public cell can access files (according to access permissions, of course) in that cell. The cell can also be kept locally (private cells), in which case only users within the cell can access it.

Many administrative operations can only be done within a cell. For example replication of a read-only volume (see section 4.2.3) can only be done within a cell. There are no physical limits to the cell's lay-out. It is possible to have the servers of a cell spread out across the world or packed together in a small server-room. Thus it is possible, for example, to have a replica of a volume both in Los Angeles and in Stockholm, as long as there is a file server from the same cell in both cities.

4.2.2 Overview of the AFS Servers

Servers in AFS are user-level processes run on one or several machines. The servers are: the File Server, the Volume Location Server, the Volume Server, the Protection Server, the Authentication Server, the Basic OverSeer Server, the Update Server and the Backup Server. Sometimes also the Salvager is counted as a server, although it runs only when invoked. The servers are described below.

Some of the servers maintain databases for keeping information needed for the service. These databases can be replicated, so that the server can be run on several machines at the same time. The replication is done with the *Ubik* replication package.

4.2.3 Volumes and the Volume Server

The administrative unit of data in AFS is called a volume. A volume contains connected subtrees of the file system, for example a home directory or a shared work directory. A volume is smaller than or equal to the size of a disk partition, so that a disk partition can contain one or several volumes.

The volume can dynamically vary in size, but has an associated quota, which is the maximum size of the volume.

There are three types of volumes in AFS: *read-write volumes*, *read-only volumes* and *backup volumes*. User volumes and other volumes that are written to often are usually read-write volumes. Read-only volumes can be replicated and are used for data that seldom changes, such as binaries. Backup volumes are used for backups.

Volumes are attached to the file system with AFS mount points. The mounting is transparent and a user can't directly see the difference between a directory within the same volume or a directory holding another volume, except by using special commands. Mounting can also be done between cells.

Since the user-visible names are location-independent the volumes can be moved from one disk partition to another. This helps in load balancing and makes it possible to move volumes when they grow and fill up the disk. Due to a special relocation algorithm the interruption to the file service is only a few seconds.

A volume can be replicated to other partitions or servers. The replicas are read-only and any of them can serve a read request for a file in the volume. This is mostly useful for volumes containing system binaries or other seldom changed but often accessed data. The replication is initiated by a system administrator. Updates to the replicas are made in a master volume and released to the replicas with a special command.

Manipulation of volumes is done through the Volume Server. The system administrator or the user sends commands to the Volume Server to move or replicate volumes, ask about the volume's attributes (like the physical location or size of the volume) etc..

4.2.4 The Volume Location Server

The Volume Location Server (VL Server) keeps track of which File Server or File Servers each volume is stored at. For this it maintains a Volume Location Database (VLDB). Mainly the VL Server answers queries about a volume's location and status when a client (Cache Manager) tries to open a file. The VLDB can be replicated, so that there can be several VL Servers in one cell.

4.2.5 The File Server

The File Server is responsible for the physical storage of a set of volumes. It answers calls from the Cache Manager and cooperates with the Protection Server to check the access permissions for a user. There are usually several File Servers running in a cell, in which case every File Server has its own set of volumes to handle. Unless a volume is replicated it is only stored at one File Server.

4.2.6 Authentication and the Authentication Server

Authentication in AFS is achieved with the Kerberos authentication system. Kerberos provides mutual authentication through a third-part authentication server. The client uses a password to get a ticket from the authentication server, and then uses the ticket during file system operations. The ticket is used as an id card to prove that the user is really who he or she claims to be.

The authentication is managed by a dedicated server, called the Authentication Server. The Authentication Server maintains an Authentication Database (ADB) in which it stores encrypted user passwords and the encryption key. There can be several Authentication Servers replicating the ADB in one cell.

4.2.7 Authorization and the Protection Server

In addition to the operating system's mode bits each directory in AFS has its own Access Control List (ACL). The ACL provides information about the access rights for different users and groups. The permission flags that can be set are Read, Lookup, Write, Insert, Delete, Lock and Administer. The AFS ACL permissions usually override the operating system's access control bits.

An AFS group is a list of users that will be treated according to the ACL rights for that group. A group can be created by any user, and is identified with the creator's user name and a group name. For example the user `charlie` can create a group named `charlie:friends` containing the users `snoopy`, `lucy` and `linus`. This group can then be given permissions in directories, which means that all members of the group get those permissions.

There are also some predefined special groups with special characteristics, for example `system:anyuser` and `system:administrators`. Permissions set for `system:anyuser` are valid for anyone trying to access the directory. The members of `system:administrators` have automatically the Administer rights in all directories.

The groups and user id:s are stored in the Protection Database (PDB). The PDB is maintained by a Protection Server, which is used to create, modify and delete users and groups and determine which groups a user belongs to. The main task of the Protection Server is to help the File Server to determine if a user has the rights to access a certain file.

4.2.8 The Basic OverSeer Server and the Salvager

The Basic OverSeer Server (BOS Server) is run at each server machine and is responsible for keeping the servers alive on that machine. It automatically restarts a server that goes down and can also be used by the administrator to take down and restart servers. At system boot it starts up the servers in correct order.

The BOS Server invokes the Salvager when a File Server or a Volume Server fails. It can also be invoked manually by the system administrator. The Salvager tries to repair any disk corruption caused by the failure.

4.2.9 Backup and the Backup Server

The volume is the backup unit in AFS. Each volume can have its own backup volume containing pointers to files which have not been changed since the previous backup, and the contents of files which have been modified. The backup volumes can be transferred to tape at any time before the next backup.

Backup is handled by the Backup Server. The Backup Server also maintains a Backup Database to keep track of the backups. The system administrator invokes the Backup Server to backup and restore volumes etc.

4.2.10 The Update Server

The Update Server is an administrative tool that can be used to distribute system upgrades and system files. The Update Server is run at one machine of each type, and the other machines run an Update Client. The Update Clients poll the Update Server for news and updates.

4.2.11 The AFS Client

The AFS client which runs on each client machine is called the Cache Manager. It creates an illusion that referenced files from AFS reside on the local disk. At a request for a file the Cache Manager first contacts the VL Server to find out at which File Server the file is stored. Then it contacts the File Server and asks for the file showing the user's authentication token. When a file is received it is cached on the local disk before it is passed on to the application.

A file doesn't have to be fetched completely. It is possible to fetch only a chunk of a certain size. The chunk size can be set individually for each client machine. The default chunk size is 64 kilobytes. There is also support for prefetching of chunks that are expected to be needed soon.

For cache consistency the Cache Manager uses a callback system. When a chunk is delivered from a File Server a callback is included. When the file changes on the server the callback is broken and the Cache Manager knows that the file has changed on the disk.

Dirty chunks are flushed back to the File Server at a close system call or with a fsync() system call. Directory data is write-through, which means that changes are written to disk immediately.

4.3 Ubik Replication

The databases in AFS use the Ubik replicating database system, described in [12]. Ubik uses a pessimistic quorum consensus protocol called quorum completion. The data in Ubik is stored in files, but the whole database is replicated. The database has an incremental version number.

One node is elected as the synchronizing site. Election is done when a node recovers from crash or regains contact with nodes it hasn't had contact with before. The election is done with a simple election protocol, where votes are given to the lowest numbered node. The node can change its vote whenever it likes, but it must wait a certain safety time interval, T, before voting for the other node.

When a node has been elected it knows that during the T interval it is guaranteed to be the synchronizing site. It will also send messages to the other nodes every time interval T, to check that they still are alive and consider it to be the synchronizing site. As long as the synchronizing site has heard from the other nodes at most the time T ago it knows that it is still the synchronizing site. The other nodes promise to wait the time T before starting a new election or voting for another site.

The synchronizing site must have contact with a majority of the sites, a write quorum, to continue operation. If it fails to contact the write quorum it must stop service. Also the members of the write quorum must be contacted periodically by the synchronizer to be allowed to continue service. If a node in the write quorum haven't heard from the synchronizer in the time T it will start an election.

All writes are directed to the synchronizing site. The synchronizing site will try to commit the write to the write quorum with a two-phase commit protocol. If it fails the write will be aborted. Reads can be serviced by any up-to-date member of the write quorum.

When a new synchronizing site is elected it checks the other nodes to find the latest version of the database. It then copies the latest version and gives it a new version number. Then it updates all the nodes in the write quorum. Since the write quorum must contain a majority of the servers, every pair of write quora intersect. Thus the new write quorum is guaranteed to have at least one node that is up-to-date.

Chapter 5

Arla

Arla is a free AFS implementation consisting of a client called Arla and a server implementation called Milko. Both are still under construction. The development started as a non-profit project at the computer club Stacken at the Royal Institute of Technology (KTH) in Stockholm. Nowadays it involves a large group of people coding, testing and suggesting updates or corrections. The project is supported by several institutions at KTH and the Stockholm University.

The first goal of the Arla project was to develop AFS-compatible clients for platforms not supported by Transarc's AFS. There are now beta versions of clients for a wide range of platforms. The client is stable and used in production. Building Milko is a much larger project and has not come as far yet. The essential servers have been implemented, but they lack many features and are unstable.

Chapter 6

Replication in General

6.1 Why Replicate?

The goals of a distributed file system are usually to provide reliability, availability and load balancing in an effective way. In small systems and with stable servers these goals are relatively simple to achieve, but when the system grows and since servers aren't stable you need to find means to improve quality.

One common problem is too high load on the servers. You can, of course, buy a faster file server when your old one can't serve enough people, but there is also another way. By replicating frequently read files to several servers the load can be spread out. This also helps users in a wide-spread network, since users in Los Angeles don't have to wait for the files to come from Stockholm if they have a local replica. Replication also increases availability, since you can read the file from another replica if one replica goes down.

Reliability and availability can be increased even more by allowing writes to the replicas. With some schemes for read-write replication you can continue your work as long as there is one replica of your files available.

With read-write replication the system administrator can take any file server down for maintenance without disturbing the users. Some forms of read-write replication can make disconnected use easier to implement, and replication can be used for making backups and easy restores.

6.2 Problems in Read-Write Replication

There are many issues to consider in the design of a read-write replicating file system. Ultimately most problems lead back to the consistency problem. How do we ensure that the accessed file is always the latest version? Do we even want to ensure that?

There are several approaches to read-write replication, and a lot of work has been done in the area. Some file systems have been implemented using read-write replication, but none of them has become widely used yet.

The first choice that is usually made is whether to guarantee consistency or not. This divides the camp into optimistic replication, where consistency is sacrificed to achieve highest possible availability, and pessimistic replication, where consistency is guaranteed at the cost of lower availability.

There are many other small and large decisions to make. Some of them depend on each other. Choosing a certain solution to one problem automatically solves some other problems, while some solutions create new problems. Here is a list of the most common problems and some suggested solutions.

- How is a write to a file distributed to all replicas? For example the client can distribute the update to all servers, in which case it must know which replicas there are. Alternatively the client only contacts one server and the update is distributed by the server.

- What do we do when the network is partitioned? In some (pessimistic) systems only one partition can continue functioning, while other (optimistic) systems allow both to continue. If the pessimistic approach is chosen it leads to the following question:

- Which partition is allowed to continue operation? Usually the partition which has the majority of the servers will be allowed to operate, though it is often not as trivial as that.

- What do we do when a server crashes? This question is interesting in some systems, while it is trivial in others. If the server, for example, held the master copy of a file, a new master copy must be elected. Some systems only need to note that the server is down and continue as usual.

- What happens when a server comes up again? This follows on the previous question, as does the answer. For example, in a master copy system a new election might be held, to see if the "new" server could be master. In some systems the server only needs to be updated with the latest news and can continue its work.

- What happens when a partitioned network gets connected again? This is related to the previous question, but also depends on how a network partition is handled. If only one partition could continue, the servers in the inactive partition can be treated as if they are recovering from a crash, but if both continued operation the system must be checked for inconsistencies.

In the following section the different approaches and algorithms for the different solutions will be discussed more thoroughly.

Chapter 7

Replication Algorithms

7.1 Overview

There has been a lot of work done in the area of replication algorithms. Most of it is done for distributed database systems, which can be very large both in numbers of nodes and geographically. Thus many algorithms are unnecessarily complicated for file systems, where the number of nodes is reasonably small (most sites have less than ten file servers) and the servers often are located close to each other and are seldom partitioned from each other due to network failures. Most of the algorithms described in this section are chosen because they can be or even have been implemented for a file system. A few algorithms and topics have been added to give a better overall view of replication algorithms.

A replication protocol can be either optimistic or pessimistic. Optimistic protocols usually use some form of available copy algorithms. Pessimistic protocols mostly use primary copy algorithms or voting algorithms. Voting algorithms, including quorum consensus algorithms, are also used in primary copy algorithms to elect the primary copy. There are many variations of voting algorithms, giving better availability or performance than the original voting algorithm. There are also lazy replication protocols using epidemic algorithms to distribute updates. Update propagation is discussed in one of the sub-sections.

Another protocol described below is the *two-phase commit* protocol, which is commonly used for atomic transactions when distributing updates. Also the concept of *view consistency* will be discussed.

Many of the algorithms are quite old and well-known. Those can most easily be found in algorithm overviews in books etc. since the original papers can be tricky to find, and it is sometimes even difficult to determine which paper is the "original". For this study I have used the overviews and descriptions in [6].

Some more advanced algorithms and concepts are described further in appendix B. Related information can also be found in appendix C, where some work on evaluating and comparing replication algorithms is studied.

7.2 Optimistic or Pessimistic Replication?

Replication algorithms are often categorized into *optimistic algorithms* and *pessimistic algorithms*. The difference is whether consistency is guaranteed or not. Optimistic algorithms allow writes to the same data item in several disjoint partitions, hoping that conflicts are rare and easily resolved. Pessimistic algorithms guarantee consistency by allowing updates to a certain data item in only one partition if the network is partitioned. Most pessimistic algorithms also hold on to *one-copy serializability*, which means that the semantics of a replicated system remain the same as if there was only one copy of the data.

Optimistic replication usually means that a file is writable as long as there is some copy available. This gives good availability, in fact about as good as it can get. On the other hand it gives many opportunities for conflicts in the file system. Solving these conflicts is in fact the major challenge in optimistic replication.

When using a pessimistic replication protocol a file is only writable if consistency can be guaranteed. Pessimistic replication can be achieved with some sort of voting algorithm or a primary copy algorithm. If the network is partitioned, this means that at most one partition can continue writing to a file or sometimes even reading a file.

7.3 Optimistic Algorithms

The most commonly used optimistic replication algorithm is the *optimistic available copy* algorithm. In simple terms it means that updates are sent to all available replicas and reads can be done from any available replica.

Problems arise when the network is partitioned. If conflicting updates are done in the two partitions the file system becomes temporarily inconsistent. When the partitions regain connection the inconsistencies must be resolved by special resolver processes. Some inconsistencies can not be resolved automatically and require manual resolution.

Epidemic and lazy replication schemes are also often optimistic, since updates are propagated slowly. When a node tries to update another node it can find inconsistencies which need to be resolved.

Optimistic protocols will be described more in appendix D, where some optimistically replicated file systems are described.

7.4 Primary Copy Protocols

Primary copy protocols are pessimistic and use a master replica for each file or volume. All writes are done to the master replica, which then forwards the updates to the other replicas. This is illustrated in figure 7.1. In some primary copy protocols reads can be serviced by any replica, while in others all file access is done via the master. The master replica can be static or elected dynamically.

A static master server is simple to implement, but is less fault-tolerant, since the write availability is the same as the availability of the master replica. Static primary copy methods are not very useful for read-write replication, and will not be further discussed here.

For dynamic election of the master replica there are several algorithms, usually based on some kind of voting. The algorithm must choose a server which is up to date, and there must be a way to distinguish the case when the master replica is down from when the master replica is in another partition. If the master replica is down a new master must be elected, but if it is in another partition it is possible that the other partition is operating, in which case the present partition must discontinue operation. This is often determined by having the master replica check regularly that it has contact with a majority of the nodes, and to yield if it has not. Thus any partition which finds that is has a majority of the nodes but no master replica can elect a new master.

The major benefit of primary copy algorithms is easily guaranteed semantics. Since all updates are sent to the master replica they are automatically serialized. Mutually exclusive write access is as trivial to implement as in a single-copy system. One-copy serializability is automatically guaranteed, since all writes are done to the same replica. The only time when serializability might suffer is when a new master is elected. This is mended by ensuring that the new master is up to date.

The major drawback with primary copy is that load balancing cannot be achieved for individual files, since all write accesses to a certain file or volume must go through the master replica. Some primary copy systems allow reads from any copy, while others demand that both reads and writes are directed to the master replica.

Since the primary copy method is quite well known and easy to understand it will not be described here at length. Instead there are some examples of file systems using primary copy algorithm in appendix D.

7.5 Voting Algorithms

Voting algorithms are based on the requirement that a certain numer of nodes must be contacted to allow a read or write operation. Most voting algorithms use quorum consensus in a simple or more complex form. The simplest voting algorithms allow operation as long as a majority of the nodes are up and available. More complex protocols allow almost all nodes to go down or be partitioned while providing some degree of service in some part of the network and guaranteeing one-copy serializability. Voting is mostly used in pessimistic systems.

Voting can be used "as it is", so that there is no primary copy for a file, which means that a client can access a file from any replica. Usually the client accesses the "best" server, choosing the server according to some criteria like load or proximity. Often a "favourite" server is chosen for each session. A form of voting can also be used in a primary copy system to elect the primary copy.

7.5.1 Quorum Consensus

The principle for quorum consensus protocols is that a *read quorum* and a *write quorum* must be contacted to perform a read operation or a write operation respectively. Every read quorum (Q_r) and write quorum (Q_w) must intersect, as must every pair of write quora.

In the simplest case it is sufficient if, in a system with N nodes, the following is true:

$$Q_r + Q_w > N$$

$$2 * Q_w > N$$

Updates can be done when at least Q_w votes are present. The update is done atomically to all present nodes, so that either all nodes in the write quorum are updated or none are. A read can be done by contacting Q_r nodes and reading from the replica with the highest version number. If these requirements are met it can be guaranteed that there is at least one up to date replica in a read quorum and that the data is read from the up to date replica.

The simplest case of quorum consensus is called *majority consensus voting*. There the read quorum and write quorum are set to $N/2 + 1$, which means that a majority of the nodes must be contacted to perform an operation. The client can either be coordinator itself or it can choose a node as a coordinator. If nodes are used as coordinators the client can choose any node as a coordinator. Majority consensus voting with a node as a coordinator is illustrated in figure 7.2. Notice the difference between this and the primary copy protocol in figure 7.1. Majority consensus voting works best as long as there are only a few replicas.

In the case when $Q_r = 1$ and $Q_w = N$ the protocol is called *read-one/write-all*, which means that a file can be read from any node and updates must be written to all nodes before operation can proceed. This is often used

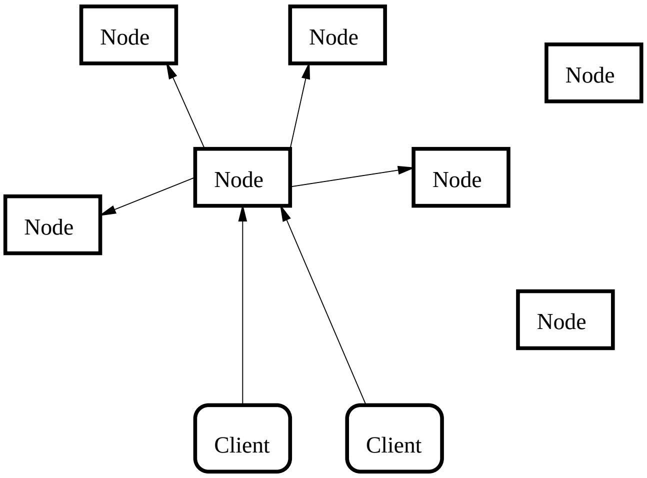

Figure 7.1. Use of a primary copy for writing. All clients send updates to the same node, which then distributes the updates to (a majority of) the other nodes.

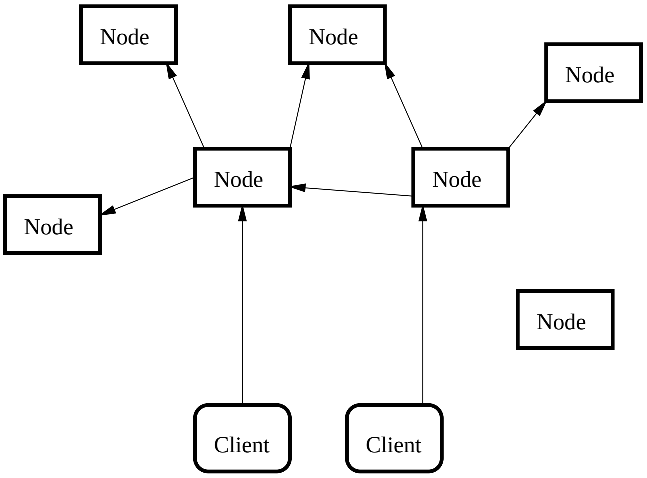

Figure 7.2. Majority consensus voting with nodes as coordinators. Clients can choose any node as coordinator. The coordinator distributes the updates to (a majority of) the other nodes.

in systems where reads are frequent and writes rare. It is not recommended for systems where writes and network failures are frequent, since it does not allow writes if some nodes are unavailable.

The *available copy protocol* is similar to read-one/write-all. The difference is that writes are sent only to available nodes, and thus allows nodes to go down. The original (pessimistic) available copy protocol is designed for systems where nodes can crash but the network is stable. Crashed nodes will miss some updates but can get updated when they recover. Network partitions could lead to inconsistencies in the file system, since conflicting updates could be done in separate partitions. Optimistic available copy protocols (see section 7.3) allow writes even if the network is partitioned and can deal with inconsistencies. The available copy protocol is illustrated in figure 7.3.

7.5.2 Weighted Voting

Weighted voting is an extension of the quorum consensus protocol. Each server which holds a copy of a replicated object can have several votes instead of only one, as in "simple" quorum consensus protocols. Thus more important servers can be assigned a higher weight by giving it many votes. Servers which are likely to go down or get disconnected can be assigned fewer votes, so that their absence will affect less. Thus weighted voting is more flexible than simple quorum consensus or voting and increases availability by allowing some nodes to go down more frequently.

In other aspects weighted voting differs very little from normal quorum consensus. Read and write quora are needed for read and write operations, and the read and write quora must overlap. Quora are counted as numbers of votes instead of numbers of nodes, which doesn't affect the quorum protocol notably.

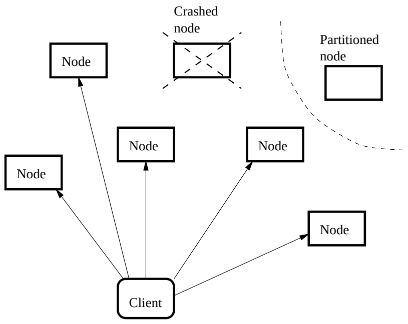

Figure 7.3. The optimistic available copy protocol. The client sends updates to all available nodes.

7.5.3 Witnesses

To improve availability *witnesses* can be added to the set of replicating nodes. Witnesses store only update information, like a version vector and update logs. They do not store the data itself. Witnesses are allowed to participate in voting and can sometimes be promoted to "normal" nodes when a "normal" node becomes unavailable, which means that they start storing the replicated data.

Witnesses increase availability by increasing the number of nodes without requiring more storage. In some cases they can even use volatile memory for storing the update logs and versions, which makes them much faster. A basic group of nodes could for example consist of two ordinary replicas and one witness. There is normally no point in having more witnesses than ordinary replicas, since a write quorum always should include at least one ordinary node. Also there is no point in having witnesses if the protocol requires that all nodes participate in either reading or writing. Witnesses can be used for most voting protocols where less than all nodes are required to participate in read and write operations.

An example of a system with volatile witnesses is described in section B.3 in appendix B.

7.6 Advanced Voting Protocols

There are many replication protocols designed for databases and other distributed systems with a larger number of nodes. There are variations of the common voting protocols, like the *dynamic voting protocol*, and protocols where the read and write quora are designed in such a way that for example the write quorum doesn't have to contain a majority of the nodes. Most of these protocols give better availability of the number of nodes is large and crashes are frequent. They are of little use in a file system like Arla where the number of replicas is going to be small. Some advanced protocols, including the dynamic voting protocol, are described in appendix B but will not be further discussed here.

7.7 Two-Phase Commit

The two-phase commit protocol is used for atomic transactions from one node (coordinator) to several other nodes (participants). An atomic commit satisfies the following conditions:

- All nodes must agree to commit or none will commit
- Either all nodes commit or none commit
- Once a node has voted it cannot reverse its decision

- If all nodes agree to commit and there are no failures, the transaction is committed
- If all nodes can agree to commit they will eventually agree to commit if all failures are repaired.

This means that if no failures occur and all nodes can commit, all nodes will commit. As soon as any node decides that it cannot commit the transaction will not be committed. Also when a transaction has committed its effects will be permanent.

The first phase of the two-phase commit protocol is called precommit. The coordinator sends out a precommit message, telling the participants what it wants to be done, for example an update to a file. The coordinator also logs the precommit on stable storage. The participants log the precommit to stable storage and decide whether or not they can commit. The participants write their votes to the log and send them to the coordinator. The coordinator waits for a certain amount of time for replies to come in.

The second phase is the commit phase. If the coordinator has received a positive reply from all participants it writes this to the log and sends out a commit message to all participants. When a participant receives a commit message it commits the precommitted transaction previously written to the log. If the coordinator receives a negative reply or doesn't receive enough replies before it times out it logs an abort and sends abort messages to all participants. When a participant receives an abort message it notes this in the log.

The coordinator has no way of knowing if a participant goes down before it gets the commit or abort message in the second phase. Instead the participant will notice this when it recovers, because it has a precommitted transaction in the log but no information about the outcome. The recovering participant must then contact the other nodes to find out whether the transaction was committed or aborted.

If a participant crashes before it has cast its vote it can at recovery time assume that the transaction was aborted, since it hasn't voted and all nodes must vote yes to commit the transaction. Also if it has precommitted a negative vote it knows that the transaction hasn't committed.

7.8 View Consistency

In some replicated systems there is a risk that the user accesses old data, since some nodes might have missed an update. Especially this is the case in optimistically replicated systems, but is also a problem in some pessimistic protocols. In the worst case the user gets data older than they have previously accessed due to unfortunate network or server failures.

View consistency guarantees that you only access the same version or later of data you have previously accessed. The view is personal for each user. Different users can access different versions of the data and in an optimistic system even make conflicting updates, depending on their views.

View consistency can be implemented in the client by letting the client remember which versions of the files that have previously been accessed. The client can deny access to a file if the versions on the available servers are lower than the last accessed version.

The problem can also be avoided by using a majority consensus protocol and denying access to a file if a majority of the nodes cannot be contacted.

An example of an implementation of view consistency guarantees is described in appendix B.

7.9 Update Propagation

Update propagation in replicated systems can be done in several ways. The most common method is to immediately spread the update to all available replicas or to all replicas in the write quorum. This gives, of course, next to instant availability to all updates at the other nodes. The update can be sent synchronously or asynchronously, which give different guarantees for update availability and write performance.

The most common alternative update propagation method is *epidemic propagation*. Epidemic algorithms let each node spread an update to a random node at certain intervals until they can assume that the update has been spread enough. Since this gives slower updates than instant distribution and thus can jeopardize the semantics and correctness of most replication protocols they will be described here only briefly.

7.9.1 Synchronous Updates

Updates can be sent synchronously to all alive nodes or to the members of a write quorum. The client itself can send updates to the nodes, but then it must know which replicas there are. With synchronous updates the client doesn't return from the update call before enough nodes have stored the update. This gives some form of voting protocol where the client is the coordinator. The client can also send the update to only one node that returns the call when it has sent the update to enough of the other nodes as illustrated in figure 7.4. This is in practice a primary copy protocol.

The positive side of a synchronous update protocol is that it guarantees that the write has reached out to other users as soon as it returns. A drawback is that it will cause delays if too many nodes fail. Also it either causes a lot of traffic at once during the update or requires a good multicast protocol.

7.9.2 Asynchronous Updates

An asynchronous update is usually done via a primary node. The client sends the update to the node, and the operation returns as soon as the node has recorded the update in permanent storage as seen in figure 7.5. Then the primary node sends out the update to the other nodes asynchronously. Primary copy protocols often work like this. If the choice of the primary node is up to the client, either the node must acquire mutual exclusion for the file before propagating the update or the protocol will be optimistic.

The asynchronous update can also be done by the client, by sending updates to all nodes that it knows of. This is an optimistic protocol, since the client does not care if enough nodes receive the update. It only needs to see that some nodes have received it, and then rely on other mechanisms to send the update around and solve conflicts. For this some sort of epidemic algorithm (described below) can be used.

Asynchronous updates have the advantage that writes take less time to execute for the client. They allow the network load to be spread out more. At the same time total ordering can suffer, except in primary copy systems. In the extreme case one-copy serializability will suffer, since the protocol becomes optimistic.

7.9.3 Epidemic Update Propagation

Epidemic algorithms rely on mathematical models which guarantee that all nodes will eventually get the update. Two common epidemic algorithms are *anti-entropy* and *rumor mongering*, and are described in [7].

Both algorithms are based on some common principles. When a node receives an update it is considered *infective*, a node which hasn't yet received the update is called *susceptible*, and a node which has decided not to distribute the update any more is called *removed*.

In simple anti-entropy all nodes are infective. At certain intervals every node chooses another node at random and resolves any differences between their contents by pushing and pulling updates between the nodes. This is a quite expensive algorithm, since the whole contents or version vectors for the contents must be compared at each interval.

Rumor spreading is a more advanced and efficient algorithm. When a node receives an update ("hot news") it starts spreading ("gossiping") it to random neighbours at certain intervals. In a while it will find that most of the neighbours it contacts have already received the update and will stop spreading it. Usually the nodes keep a counter for counting how many other nodes they have tried to contact that have already got the update. When the counter reaches a threshold value the node will stop spreading the rumor. The threshold for stopping the rumor spreading will decide the probability that some nodes miss an update. While it is good to stop spreading the

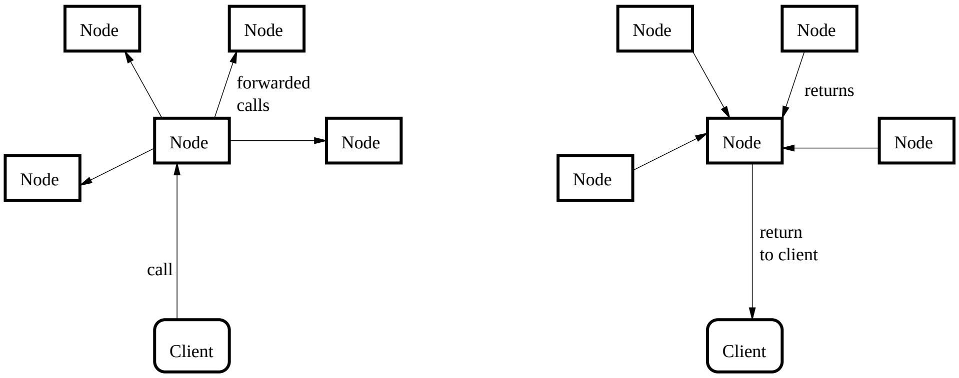

Figure 7.4. Synchronous update propagation. The update is distributed to the other nodes before the call returns.

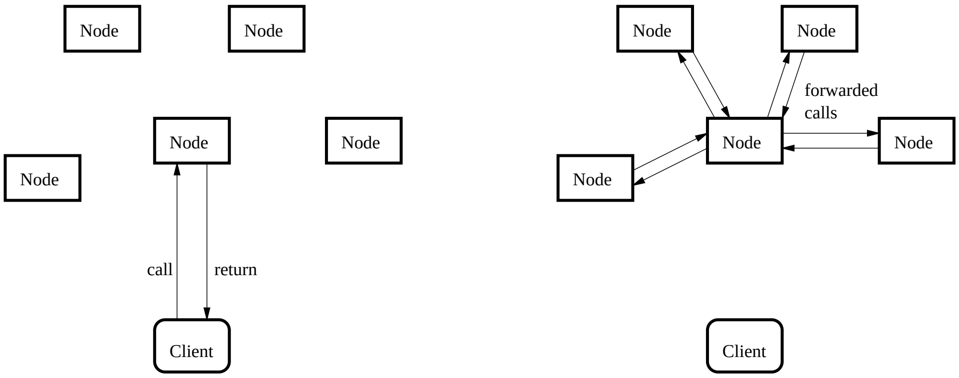

Figure 7.5. Asynchronous update propagation. The call returns before the update is distributed to the other nodes.

update so that there will be less traffic, it is not good if some nodes miss the update.

The threshold can also be implemented as a probability for whether to stop spreading the update or not. This requires less storage since no counter is needed. At the same time it is more uncertain, since it can happen that a node stops spreading the rumor almost at once or keeps sending it for too long. Also it can be decided that nodes which haven't heard from other nodes in a while will try to pull information from a random node at certain intervals, to be sure that they haven't missed any updates.

Deletion of an item requires that a special item type, a "death certificate", is spread. Otherwise the absence of an item at one node will be interpreted as that the node hasn't received the item yet and the item will be restored. The death certificate circulates in the same way as normal updates. When a node tries to "update" another node with a file that has been deleted at the other node the other node will see that the death certificate is later than the file and thus refuse to receive the file.

Chapter 8

Replication in Other File Systems

Read-write replication of files has been previously implemented in several other file systems. One of the more successful is Coda, developed at Carnegie-Mellon University and partly based on the same concept as AFS. Some other attempts never came further than primary testing due to lack of interest and funding or because the platform it was developed for became obsolete. It is in the replicating file systems where the most complete solutions often can be found, even though the solutions aren't directly adaptable to Arla. The implementations show what protocols have previously been considered worthwhile, and which of them really worked. In this section some of these existing implementations of read-write replication will be discussed and compared. More detailed descriptions of each file system separately can be found in appendix D.

8.1 Optimistic Replication

Optimistic replication has been implemented successfully in Coda ([27], [25], [5]) and Ficus ([17], [22]). They are similar in some aspects but have many differences. Both use an optimistic available copy protocol to initially spread the updates, but in Coda the updates are sent to the available nodes by the client while in Ficus the client chooses a favourite server that distributes the updates to the other nodes. Conflicts are guaranteed to be detected. In Ficus the nodes contact each other for resolution periodically, while the nodes in Coda keep track of each other and do resolution when a node regains contact with a node it previously lost contact with. Both systems use automatic resolvers when possible and contact the users when automatic resolution cannot be done.

Ficus and Coda have both been developed a long time. The recovery protocols are complicated and seem to have required a lot of work. Measurements described in the papers show that both systems have good performance.

8.2 Primary Copy Protocols

Two file systems using a pessimistic primary copy scheme are Echo ([14], [4] and Harp ([13]). There are several minor differences between Echo and Harp, but the replication scheme is roughly the same. Both use a primary copy scheme with a primary node, a set of secondary nodes and some witnesses. The primary sends updates to the secondaries with a two-phase commit protocol. The secondaries commit the logged changes to disk asynchronously. The primary copy handles both reads and writes.

The primary node checks its contact with the secondaries periodically. When the number of alive secondaries changes an election is held. Election will also be initiated by a secondary node if it doesn't hear from the primary for a period of time. The initiator of the election (the coordinator) sends out an election message to the other nodes. A node receiving an election message gives a positive vote to the coordinator if the coordinator has higher priority than the node itself and if the coordinator has the same or later versions of the replicated files. The versions of the files are kept track of in a similar way in both Echo and Harp. In Echo an *epoch* variable is used to keep track of the current election period. In Harp this is called a *view*. The nodes keep track of which epoch or view they were last updated in. A node that can collect positive votes from the majority of the other nodes and gets no negative votes becomes the new primary and announces this by announcing a new epoch or view.

In both systems it seems like the most complicated part of the protocol is to keep track of the epochs and views respectively. Recovery from crashes becomes slower due to the election, and during election the replicated data will become unavailable for a short period of time. The performance of Echo was not measured in any of the above mentioned papers, and the development was stopped in 1993. Measurements for Harp are difficult to interpret, since Harp stores the logs in volatile memory (each server is connected to an UPS to protect it from memory loss), which makes updates faster than if the changes had to be synchronously stored to disk.

8.3 Replication with Tokens

Tokens are often used for mutually exclusive writes. For example in Echo read and write tokens are given out to the clients by the primary server. Deceit ([28]) and Huygens ([8]) use tokens to determine which node can write to a file at a given moment. In Deceit the tokens are administrated by token holders, while Huygens uses a virtual token ring to pass the tokens.

Deceit uses *replication on demand*, which means that a new replica is created at a node when a client tries to access the file at that node. A minimum replica level is upheld for each file, but the number of replicas can increase. The node holding the token is responsible for keeping up the number of replicas. If a node goes down a new replica is placed on another node. The token is passed between nodes whenever a node wants to write to the file. Before actually writing the token holder needs to send out a notification that the file is unstable and get a response from a certain number of other nodes.

Since the number of replicas in Deceit is variable it is possible to have two network partitions that have enough nodes to continue operation. This means that inconsistencies can occur. Inconsistencies will be detected when the nodes get connected again. Two versions of the file will be kept and the user will have to resolve the inconsistency.

In Huygens the nodes are ordered into a virtual ring. The token is sent around in a token-ring fashion. Also a keep-alive message is sent around to detect breaches in the ring. When the ring breaks a new ring will be formed without the missing nodes. Depending on the number of missing nodes the new ring can continue either full service, read-only service or no service.

Tokens can be used to guarantee mutual exclusion, but mutual exclusion is not always necessary. In many file systems it is sufficient that the updates are totally ordered, as they will be if a two-phase commit protocol is used.

8.4 Epidemic Update Propagation

Epidemic update propagation is an interesting concept, though perhaps not so useful for Arla. Ficus uses a kind of epidemic propagation of updates after the first update of available copies. Bayou ([21]) uses epidemic update propagation to spread all updates. The update protocol is based on one-way reconciliation, where the receiver sends its version vector to the sender and the sender sends all more recent updates to the receiver. Each node chooses periodically one or several other nodes to reconciliate with. Thus the updates are eventually spread to all nodes.

8.5 Other Solutions

There are a number of other replicating file systems. Many claim that they have replication, but most have only read-only replication. Some, like Frolic ([23]) are high-level solutions for wide-area networks. There are also hardware based solutions and hardware aid for stability. Harp requires that all file servers are connected to an UPS to ensure that the volatile logs aren't lost if the power fails. Many file systems use RAID to get reliable disks. In HA-NFS ([2] two servers are connected to the same set of SCSI disks. If one server crashes the other will take over its IP address and disks. Solutions like these can be interesting to look at, but since they often are old and built for specific hardware or software they are not worth much more attention in this work.

Chapter 9

Replication in Arla

9.1 The Choice of a Replication Protocol

Many replication protocols have been described earlier in this study. Most of them are not possible to implement in Arla. In this chapter the possibilities and limitations will be described and discussed. First there will be some discussion about the limitations, requirements and other factors that affect the choice. Due to these factors many protocols can be ruled out. When the limits are clear the remaining possibilities will be described.

The protocols that actually will be considered here are *pessimistic primary copy*, *pessimistic majority consensus*, *pessimistic read one/write all* and *optimistic available copy*. All of them have been previously described in chapter 7. Pseudo-code for some of the algorithms can be found in appendix A.

The pessimistic algorithms can be varied in several ways. The most important choices affecting the protocols are asynchronous or synchronous updates and the design of the recovery protocol. These will also be discussed in this chapter. Finally the concept of *witnesses* will be discussed.

9.2 Limits and Requirements

There are certain limitations and requirements in the choice of a read-write replication scheme for Arla. To find out what the system administrators would like I interviewed some AFS administrators via e-mail. I have also talked to some of the Arla implementors to hear their opinions.

The greatest limitation lies in the compatibility with AFS. The Milko server implementation must be compatible with the AFS client, which immediately rules out some approaches to replication. The client expects to send all write requests to one file server, which is usually pointed out to it by a Volume Location Server. Thus the replication must be handled by the servers without help from the clients.

It seems fairly clear from the interviews that the file system should be kept consistent. An optimistic scheme will be discussed, but the main focus will be on pessimistic replication.

It also seems clear from the interviews and discussions that the main reason for replication of read-write data would be to increase availability during server, disk or network failures. Also most AFS cells have quite few file servers, so it can be assumed that read-write data would be replicated to only a minimum number of nodes. There is simply no point in having more replicas than two or three, and more replicas would take up more disk space. Thus the replication algorithm should be optimal for two or three replicas. Scalability is still important, but the emphasis should be on small numbers of nodes.

Extra network load should be kept as low as possible. Network load is one of the major drawbacks of most of the replication schemes available. It is inevitable that the load increases, but some choices can be made so that the load increase is minimised. Instead, the file servers often have CPU cycles to spare, so if there is a trade-off between network and CPU load the network should be spared.

9.3 Which Parts are Affected?

The main participants in the replication scheme will, of course, be the file servers. Also the volume location servers will have to be aware of it. The natural unit of replication is a volume, since it is the administrative unit in AFS. As stated earlier the client should not need to be aware of the replication. It is possible to give the Arla client extra features which increase the performance compared to the AFS client, but the AFS client should not get lower performance than usual.

In AFS the file servers are not aware of the other file servers. In a replicated Arla system the file servers need to be aware of each other when distributing updates etc. The alternative would be to let the volume server handle the distribution, which would load the volume server unnecessarily much.

9.4 The AFS and Arla Clients and Replication

The AFS client is aware of read-only replication. When it requests volume location information for a read-only replicated volume it gets a set of file servers as a reply. It will then choose a suitable file server to read from. If the chosen file server goes down it will choose another.

The client caches volume location information and can access the volume without consulting the volume location server. It is also possible to manually access a volume on a file server if you know which file server it is located on. There needs to be a way of redirecting or delaying a call from a client if it tries to access a file server that isn't currently in service for one reason or another. There is also a need for a way to choose another server if one is completely unreachable.

There are some mechanisms which can be used to redirect access requests. The client can be told by the volume location server that there are several replicas of a volume. The AFS client could get confused if a read-write volume is said to be replicated, but the Arla client could easily be modified to accept this. This allows the client to choose another file server if one goes down. Thus the AFS client will not get lower availability than if the volume was non-replicated and the Arla client will get better availability.

If a file server finds that it isn't allowed to serve a request from a client because it has lost contact with too many other file servers it can "lie" to the client and say that the volume has moved to one of the unreachable file servers. The client will try to access the other file server and will (probably) find that it cannot be reached. At this point an AFS client will simply wait for the other server to come up. If the client is an Arla client it might know that there are even more replicas and will try to contact them all one at a time, but all replicas it can reach will tell it that the volume has moved to an unreachable file server. This can be optimized by letting the file server know that the client is an Arla client so that the server can tell the client that the volume cannot be reached right now.

Another solution is to let the client wait until the file server regains contact with enough other servers to serve the request. The volume could even be marked "busy" until it can be written to again.

9.5 The Effect of Caching

The client in Arla and AFS caches file data on the client machine's local disk. Writes are not committed to the server immediately. Instead several writes to a file are committed together in a flush operation periodically, when the file is closed or with a fsync() system call. Thus the cache can be more updated than the server, and a user reading another cached copy of a file might read a stale version. It is even possible for two users to update a file simultaneously without noticing it until either of them is flushed to the server. While both changes still are in the caches the system could even be called inconsistent.

This gives a certain amount of uncertainty. You know that you'll be notified by broken callbacks (see section 4.2.11) if the file is changed on the disk, but you cannot be sure that changes reach the disk immediately. Since this is well known, users do not expect anything but problems if, for example, two users edit the same file at the same time. Files requiring changes to be valid immediately, like databases that can be accessed by several users at the same time, aren't even kept in AFS. When mutual exclusion is needed locking is used. Locking can also be implemented for a replicated system if a pessimistic replication protocol is used.

This "relaxed" view on update propagation can be seen as an advantage when designing a replication protocol. The programmers and the users know that simultaneous changes in a file can give unpredictable results and thus avoid it. Thus the replication protocol can also be allowed a slightly relaxed update propagation. For example asynchronous updates could be allowed to some extent. Whether this is useful or not depends on the replication protocol, but it is still worth notice.

9.6 Consistency Models and Availability

There is always a trade-off between consistency and availability. Should servers be allowed to continue service if they have lost contact with the other servers? If all alive servers are allowed to write to all files they replicate the file system might become inconsistent and if all servers are allowed to read the files they replicate some might give out stale data. On the other hand, if only one partition of servers which have lost contact are allowed to serve requests the accessibility will suffer. The choice lies between optimistic and pessimistic protocols, and between different pessimistic protocols.

A pessimistic quorum consensus algorithm can be used to maintain the consistency. There are two reasonable quorum settings for this case: majority quorum and read one/write all. Majority quorum requires that a majority of the replicas is reachable for both read and write requests. This lets only one partition operate in a partitioned network. Read one/write all requires that all replicas are available when a write request is served, but lets read requests be served as long as any replica is reachable. This gives higher read availability but lower write availability in the presence of network partitions or crashes.

A pessimistic primary copy algorithm keeps the updates ordered and maintains consistency with ease. Problems can only arise when a new primary copy is to be elected.

The optimistic approach is to allow reads and writes to any replica. There is a risk that inconsistencies occur, since the same file can be updated in different network partitions, but it will always be possible to update a file as long as there is a replica available. The problem is how to handle inconsistencies. All inconsistencies need to be reliably detected and resolved. The detection must be automatic, but resolution can be done manually. Often manual resolution is in fact the only possibility.

9.7 Pessimistic Replication with a Primary Copy

In a primary copy system all reads and writes are directed to a primary copy that distributes the updates to the secondary nodes. The system can easily be kept consistent and writes are automatically ordered in the order they arrive at the primary copy. The protocol will not help with load balancing but will increase stability and availability since the data is "backed up" on the secondary servers. If the primary server goes down a secondary can fairly quickly be elected as primary and start service.

The major design issue in the primary copy protocol is the election of a new primary node. A suitable algorithm could be the bully election algorithm. Election is to be held when the number of active nodes changes. The nodes are ordered hierarchally and will vote only for a node with higher priority than itself. If a node gets an election message from a node with lower priority than itself it will also start an election. A node that has started election and receives an election message from a node with higher priority will abort its own election. Thus the active node with the highest priority will get the most votes and win the election.

The primary node sends messages periodically to the secondary nodes to check that they still are alive. If it fails to get a reply from a secondary, or gets a reply from a node that was previously down it will start election. A secondary node that doesn't get a check message from the primary after the appointed interval also starts election. Thus the primary knows that after a check it is the primary for at least the time until the next check and can thus serve reads during this time without consulting the secondaries.

The new primary node has to be up to date, which means that there needs to be a way of telling whether a node is up to date. Many systems keep an incremental view number for each election period and to let the nodes remember which view they were last updated in.

The distribution of updates from the primary to the secondaries is done with two-phase commit. Writes could be allowed as long as the primary gets a positive answer from a majority of the nodes, but even better is to require that all nodes that participated in the latest election send positive replies, similarly as described in section 9.9.1. If the primary doesn't get positive replies from all nodes that were previously available it starts a new election. This guarantees that the view number changes as soon as the number of active nodes changes. Thus it is assured that a node that is last updated in a specific view has all precommits sent in that view. If the node has missed a precommit an election has been held by the coordinator and a new view has been started, since the node naturally didn't send a reply to the precommit it missed.

The election itself is fairly simple. A node that starts election (the coordinator) for one of the reasons above sends out an election message to all other nodes stating its priority number and which view it was last changed in. A node that receives an election message compares the coordinators priority and view number with its own priority and current view. If it finds that the coordinator has higher priority and the same or higher view number it sends it a yes vote. Otherwise the node starts an election itself. The coordinator waits a certain time for votes and if it receives votes from a majority of the nodes (including itself) it becomes the primary. The new primary announces itself by sending out the new view number to the other nodes with a two-phase commit.

Whenever a primary node receives an election message that it votes for it will stop service. It will also stop service if it starts an election itself. Service is restarted only when the new primary (or re-elected old primary) is elected and has sent out the new view number to the secondary nodes.

A recovering node should first try to get itself updated according to the procedure described in 9.11.1. Thereafter it starts election to see if it is entitled to be the primary node. It is possible that it gets election messages from other nodes during the recovery phase, and in that case it will vote according to its view number and priority.

Witnesses can be used as discussed in 9.13. The minimum group of replicating nodes consists of one primary, one secondary and one witness.

Directing reads and writes to the primary copy can be done either on the volume location level or on file server level. It would probably be easiest to let the secondary nodes redirect the clients to the current primary node.

9.8 Pessimistic Replication with Majority Consensus

Majority consensus voting means that a majority of the nodes must participate in all read and write operations. This guarantees consistency, since at least a majority of the nodes will be up to date. A two-phase commit protocol is used for reads and writes to ensure that all nodes get the updates in the same order. If a node misses an update completely it will have to detect that and do a recovery from a node that is up to date. Version numbers can be used for detection of missed updates. If a node is asked to write a later version than the next one according to its own version numbers it can assume that it has lost an update.

The design of the recovery operation influences the read and write operation, as will be seen later on. The following descriptions of the read and write operations in the majority consensus voting and read one/write all schemes are suitable for both eager and lazy recovery (see section 9.11), though they could be simplified if eager recovery is used. When using lazy recovery a node that isn't fully recovered, i.e. has unresolved precommits in its log or has an older version of the file than the new version minus one, considers itself busy until it is fully recovered. A busy node will refuse read and write calls until it is free.

9.8.1 The Write Operation

When data is flushed from the cache the client sends the data to a file server. This file server acts as the coordinator in the following procedure:

- The coordinator sends out a precommit write request to the other nodes, containing the data that is to be written.
- The other nodes record the precommit in a log and send a reply to the coordinator if they can commit. A node that cannot commit because it is busy can either answer that it is busy or delay its answer until it can commit. A node that cannot commit for a more permanent reason, like a full or broken disk, sends a negative answer stating the reason for failure.
- The coordinator waits for answers from the other nodes. When it has received positive replies from a majority of the nodes it commits the data to disk and sends out a commit message to the other nodes, which then commit the data.
- The coordinator will time out if it doesn't receive replies from enough nodes. It will detect if the nodes which didn't answer are busy or down. If a majority of the nodes are unreachable the write fails and the client is told that the volume is unreachable.
- If a majority of the nodes send negative replies the write fails and the client is given an error message.
- Whenever a write fails the coordinator logs and sends out cancel messages to the other nodes so that they know that the commit was cancelled.
- If a majority of the nodes can be reached but are busy the volume will be considered busy. The coordinator delays the commit until the write can be either committed or cancelled, depending on the replies it gets when the busy volumes become active.

9.8.2 The Read Operation

At a cache miss the client will ask for data from a chosen file server. This file server acts as the coordinator in the following procedure:

- The coordinator sends out a request to the other nodes asking them for the version numbers for the requested file and volume.

- The other nodes send replies to the coordinator.
- When the coordinator has received replies from more than a half of the nodes it compares the version numbers. The node which has the highest version number (including itself) is guaranteed to have an updated copy of the file, since there is at least one node that is up to date in every majority group of nodes.
- If the coordinator isn't up to date it asks for the file from the node which has the highest version number for the file.
- The coordinator sends the data to the client.

9.9 Pessimistic Replication with Read One/Write All

Pessimistic replication with read one/write all is similar to the current read-only replication scheme in AFS. For a write operation all nodes are required while a read operation can be served by any node. Consistency is guaranteed by requiring that all nodes have identical replicas at all times. The difference between read-only replication and read-write replication with read one/write all is that the `vos release` operation which updates the read-only replicas in AFS must be done by an administrator while read-write replication can be done automatically. Also the update procedure can be made faster compared to the `vos release` procedure.

The write operation is done with a two-phase commit protocol. Error detection and recovery is simpler in the read one/write all protocol than in the majority consensus protocol, since all nodes should have the same versions of the files. From the user's point of view, read one/write all works best with synchronous updates. The drawback to that is that all nodes must answer before a write can be committed.

9.9.1 The Write Operation

When data is flushed to disk the client sends the data to a file server. This file server acts as the coordinator in the following procedure:

- The coordinator sends out a precommit write request to the other nodes, containing the data that is to be written.
- The other nodes record the precommit in a log and send a reply to the coordinator if they can commit. Nodes that cannot commit because of full or broken disks or other reasons of failure send negative replies stating the reason of failure.

- The coordinator waits for the answers from the other nodes. If it receives positive replies from all nodes it commits the data to disk and sends out a commit message to the other nodes, which then commit the data.

- If the coordinator receives any negative replies the commit fails and the client is notified of the reason.

- Nodes that are busy or down will be detected by the coordinator. A node which cannot be contacted before timeout is considered down. This will cause the write to fail with the reason that a node is unreachable.

- Whenever a write fails the coordinator logs and sends out cancel messages to the other nodes so that they know that the commit was cancelled.

- If a node is busy the volume is considered busy. The coordinator waits with the commit until the write can be either committed or cancelled, depending on the reply from the busy volume.

9.9.2 The Read Operation

The read operation works just like a read operation on a read-only replicated volume. At a cache miss the client asks for the data from a chosen file server and the file server sends the data to the client. Since all active nodes have the same version of the data it is guaranteed that the data read is up to date.

9.10 Optimistic Replication

The best availability can be achieved with optimistic replication. Optimistic replication means that a file can be written to as long as there is at least one available server holding a replica. The simplest optimistic protocol is the *optimistic available copy* protocol. Updates are simply sent to all available nodes. When nodes regain contact or recover from crash they have to reconciliate their contents with the other nodes. Implementation of a good optimistic protocol is complicated, so I will only briefly discuss the main issues from the Arla point of view.

The write operation needs to update several nodes which may have different versions of the same file. If the system is inconsistent the inconsistencies must be resolved before the write can proceed.

The read operation must get the latest possible version of the file. Preferably it should also see to it that a client never reads an older version than it has previously read for a certain user. Otherwise the user might suddenly get an older version of a file than previously.

The reconciliation process consists of two parts: detecting inconsistencies and resolving them. There must also be a strategy for reconciliating a node with a group of nodes, since it is not guaranteed that all nodes in the group has the same versions.

The detection of inconsistencies in Arla would require comparison of data rather than version numbers, since the version number is a plain integer that only tells how many times the file has been updated. Two replicas could have been updated equally many times but still have different contents. Comparison of data can be done in several ways. All updates can be logged and the update logs can be compared, or the files themselves can be compared. Another option is to let all nodes keep track of each other and log their states whenever they lose contact with another node. These are all quite complicated operations.

The resolution process is no less complicated. Many optimistic file systems use automatic resolution as much as possible. For directory data automatic resolution is feasible, but for most user files only manual resolution is possible. A way to administer this is to keep several versions of the conflicting files until the user has resolved the conflicts.

A good optimistic replication protocol is complicated. It will also require manual conflict resolution, something which most system administrators would like to avoid.

9.11 Recovery in Pessimistic Replication

When a node recovers from a crash or a link failure it must somehow get updated. This can be done either before the server is allowed to participate in voting (what I call eager recovery) or whenever it needs the latest replica (what I call lazy recovery). Eager recovery would guarantee that all replicas in contact with each other have the same version while making the recovery of the server slower. Lazy recovery means that the server is allowed to participate in voting as soon as it gets up or regains contact but that it can have an older version of a file and will be required to update the file at a later time.

Recovery is mostly needed for majority consensus protocols, but also the read one/write all protocol will need some amount of recovery, since a node might have missed a commit if it went down after agreeing to a precommit but before the precommit was resolved.

A node isn't fully recovered as long as it has an older version of a file than any of the nodes it can reach or if it has missed commit or cancel messages. A node is up to date when it has the latest version of all files in all volumes it replicates.

Recovery is best done one volume at a time until all volumes are recovered.

9.11.1 Eager Recovery

Eager recovery keeps all active nodes recovered by requiring that a node gets all updates for a volume before it starts participating in voting for that volume after a crash or failure. During the recovery process the node is considered busy regarding the particular volume, which could mean that the node remains busy a long time if it cannot contact a node that has newer information.

The first step in the recovery is to determine whether the node has missed any commits. Since the commit-log is kept in permanent storage it can compare its latest log-entry to the other nodes' log-entries.

The recovering node sends its latest log-entry to an active and fully updated node. Since eager replication guarantees that all active nodes which have contact with each other have the same versions it can choose any active node. The active node finds the position in its commit-log and sends all newer log-entries to the recovering node.

Since the commit-log contains all information needed for updating the volume the recovering node can simply go through the newly received entries in the commit-log to update the volume.

While the recovering node waits for the log entries it can receive new pre-commit, commit and cancel messages. These must be stored in a temporary log until the previous entries have been received. When all previous entries are received the new entries can be added to the top of the log.

If eager recovery is used the read operation becomes a bit simpler in the majority consensus protocol. Since the node knows that it has the same version of all files as all other nodes it can reach it only needs to determine whether it can get an answer from a majority of the nodes without having to compare versions.

The commit-log will eventually take up a lot of space. Thus it needs to be truncated, throwing away the oldest entries. The simplest way to do this is to allow truncating at will, for example when the log reaches a certain size. This could lead to that the newest entry of a node that has been out of reach for a long time is too old to be found in an active node's log. In this case the whole file or volume will be sent to the recovering node, which can then throw away its old commit log, starting a new one from the state of the copied file.

9.11.2 Lazy Recovery

Fast recovery of servers is often asked for. With eager recovery of the replicated volumes it will take some time before the server is fully functional.

Lazy recovery can improve the recovery time, but the cost must be paid later. Instead of updating all volumes at once they are updated when they are next written or if the server is chosen to be a coordinator for a read or a write. This makes the read and write operations slower if a server isn't fully recovered.

The recovery of a volume can, when it is done, be done similarly to eager recovery, with the exception that the recovering node first has to find a fully recovered node. With the majority consensus protocol it needs to contact a majority of the nodes and recover from the node that had the highest version number, similarly to a read operation. With read one/write all the recovering node has to find nodes which know the results of all pending precommits that have been committed or cancelled. A good idea is to start by trying to contact the coordinator for the precommit.

When a node is chosen by a client to act as a coordinator it needs to be recovered. At a write operation the coordinator must recover before it starts the two-phase commit sequence, since it should have the latest version of a file before it starts sending updates to the file to other nodes. During a read operation in the majority consistency protocol the coordinator can do a recovery from the node it finds has the latest version of the file. When the read one/write all protocol is used the node chosen for reading must try to resolve all unresolved precommits it has in its log.

To fulfill the consistency requirements all nodes that participate in a write operation must do a recovery for the volume being written. This can cause a lot of traffic during the write operation if many nodes are in need of recovery. This can be optimised a little by letting the coordinator recover before it starts the two-phase commit and then letting the participants recover from the coordinator instead of having to first find an up to date node to recover from. The participants are considered busy while doing the recovery.

As described previously the logs can be truncated, in which case the recovery sometimes requires copying of complete files or volumes.

9.12 Asynchronous or Synchronous Updates?

In a pessimistic update protocol for a replicated system the write operation is usually complicated and takes a considerably longer time than the corresponding operation would take in a non-replicated system. The suggested pessimistic protocols in this study use a two-phase commit protocol, which requires timeouts. In the presence of failures the write operation can be considerably prolonged while waiting for time out.

To avoid waiting for timeouts and slow links it is possible to do the write operation asynchronously. For example, the write operation can return as soon as the coordinator has initiated the two-phase commit protocol. This would give as fast writes as in a non-replicated system, since the coordinator is the only node that needs to record the update to its logs before the write operation returns.

The problem with asynchronous updates is that it cannot be guaranteed that the write operation succeeds until the two-phase commit protocol is completed. The user will think that the write is done since the write operation has returned, while the coordinator only has begun the real updating process. The user might even think that all work is saved and log out before the coordinator can report that the write failed.

Synchronous updates are safer but slower from the users point of view. When the write operation returns the update is guaranteed to be either succeeded or failed. The user will know the outcome at once. This can be extra important in a read one/write all system, where the write operation requires the participation of all nodes, which gives more points-of-failure.

Timeouts can make the update slower, but mostly in the presence of so many failures that the update cannot succeed anyway. In those cases one might even prefer to wait a while to be told that the write failed than to first hear that the write was completed and a short time later that the write actually failed.

9.13 Witnesses

A way to virtually increase the number of replicas in a pessimistic system while using a minimum amount of disk space is to use witnesses. A witness is a node replicating a volume without storing the data. Instead, it only stores update logs and version numbers for the files. The witness can participate in voting and even recovery as long as it has enough logs stored. The witness cannot serve read or write requests by itself.

There can be any amount of witnesses, but it must be remembered that at least one ordinary node must participate in every read or write operation. Witnesses are often used in systems with only a few replicas to get an odd number of replicas. A good constellation is having two ordinary nodes and one witness.

Of the replication schemes described earlier witnesses are most useful for primary copy and majority consensus replication. For read one/write all they would only increase the amount of nodes that have to participate in the write operation without increasing the amount of nodes that can be read from.

The recovery process of a witness is simpler than the recovery of a ordinary node. The easiest thing to do is to discard the old logs and get the latest version information from another witness or an ordinary node. It is also possible to do recovery in the same way as for an ordinary node, which means copying the update log from another node.

An ordinary node can use the witness for recovery as long as the witness has sufficiently old logs. If the witness is insufficient the ordinary node will have to contact another ordinary node to do recovery.

A witness can be implemented so that it stores the logs and version numbers in volatile memory. This would mean that it loses all information if it crashes, but since it can discard the logs and must get new version information anyway this won't do any harm. The advantage of "volatile" witnesses is that they are faster than witnesses storing logs on disk.

9.14 Summary

Considering the limitations and requirements for a replication protocol in Arla the following options are reasonable:

- Protocols:

  Primary copy (pessimistic), all reads and writes are done through a primary node. Requires election.

  Majority consensus voting (pessimistic), a majority of the nodes participate in both reads and writes.

  Read one/write all (pessimistic), write to all nodes and read from any node.

  Available copy (optimistic), write to any node. Can cause inconsistencies.

- Updates:

  Asynchronous, fast but will cause some difficulties in the protocols.

  Synchronous, slower but safer.

- Recovery (pessimistic protocols):

  Eager, gives longer recovery times but keeps the nodes updated.

  Lazy, givers quicker recovery but reads and writes might be delayed by recovering nodes.

Chapter 10

Recommendations

10.1 Recommended Protocols

Based on the earlier discussion I would like to recommend pessimistic replication with synchronous updates and eager recovery. Optimistic replication is complex and requires more maintenance than pessimistic replication. Synchronous updates guarantee that errors are detected before the flush operation is ended and makes the update algorithms simpler. Lazy recovery would make the servers quicker to recover, but will cause some slow updates when a replica must be recovered during a write operation. Eager recovery is easier to implement and also makes the read and write processes simpler.

The best protocol to implement is the pessimistic read one/write all protocol. The protocol is fairly simple and easy to understand, thus easier to implement correctly. Load balancing and higher availability is achieved for reads than when using the primary copy protocol. Recovery of a node is simpler and faster than for a majority consensus replicated node, since a node can only miss commit or cancel messages. Updates cannot be done if a node is down, thus a crashed node cannot miss any. Read one/write all replication can be used for many volumes which are currently read-only replicated, thus allowing ordinary users to make changes in them without assistance from an administrator. In addition, the update procedure in read one/write all should be faster than the `vos release` operation required to update the replicas of a read-only replicated volume.

It is also possible to implement a second protocol along with the read one/write all protocol. It can be chosen for each replicated volume which replication protocol to use. Majority consensus or primary copy could be useful for work directories that need increased read-write availability. For example users could have both a small replicated volume and a larger non-replicated volume to their disposal. In the replicated volume they can keep important data that they want to be able to access even when there are server or network problems. Less important data can be kept in the non-replicated volume. This would allow the administrator to keep the amount of replicated data down to a minimum and thus save disk space.

The choice between majority consensus and primary copy depends mainly on whether the period of unavailability during election can be accepted or not. The primary copy protocol is simpler, since reads and writes are automatically serialized, but also the majority consensus protocol can guarantee serializability, even if it takes a bit more effort.

If the primary copy protocol or the majority consensus protocol is implemented witnesses should also be implemented. A minimum group of nodes replicating a volume would then consist of two ordinary servers and one witness. To read or write to a file in a certain volume two of the nodes in the group need to be available. Thus the data in the volume can be accessed even if one of the nodes goes down. The amount of replicating nodes can be increased, but the number of replicas should always be odd. A majority of the nodes should be ordinary servers.

10.2 Replication or not?

The advantages of implementing replication with the read one/write all protocol are:

- It can replace read-only replication for some volumes, making them easier and faster to update.
- It can be useful for some volumes that are currently non-replicated, giving them higher read availability and better load balancing.

The advantage of replication with the majority consensus protocol and the primary copy protocol compared to a single copy is:

- It increases both read and write availability during server or network failures.

The drawbacks of read-write replication for a volume that is currently non-replicated are:

- Slower updates and sometimes slower reads.
- Slower recovery of the servers, especially when using the majority consensus or primary copy protocol.
- Increased network load during read, write and recovery operations.
- Periods of unavailability during election if the primary copy protocol is used.

In addition the read one/write all protocol has the drawback that a volume cannot be updated if any of its replicas is unreachable or down.

Considering the advantages and drawbacks I would recommend implementation of at least read one/write all replication. Replication with primary copy or majority consensus can be added later, if the implementors have sufficient time and interest.

10.3 Administrative Functions

Some administrative functions have to be slightly adjusted for replicated volumes. Since it will be fairly obvious what needs to be done I will only discuss this briefly and leave the details to the implementors.

The administration of a read-write replica resembles in many ways the administration of an ordinary read-only replica. A read-write replica in Arla can be created and deleted in the same way as a read-only replica in AFS. It can be moved in the same way as any volume.

Backups can be done for each replica separately. All replicas don't have to be backed up, as long as the backup is done from an updated replica. A destroyed replica can be recovered either from a backup or by ordinary recovery from another replica.

Chapter 11

Conclusions

There are many algorithms for read-write replication of files in a distributed file system. Most of them are not suitable for Arla due to the compatibility requirements with AFS. Since it can be assumed that the group of nodes replicating a volume in Arla would be quite small the simplest protocols should be good enough.

A read one/write all protocol can be implemented to partly replace the existing read-only replication. Better write availability can be achieved with a majority consensus voting protocol with witnesses. Both protocols can be used simultaneously, in which case the replication protocol can be chosen individualy for each replicated volume.

Both protocols use a two-phase commit protocol for distributing updates. The coordinator is a node chosen by the client. Updates should be synchronous.

Read-write replication with read one/write all has enough advantages to be considered worth implementing. The majority consensus protocol has some more drawbacks but might still be worth consideration.

Chapter 12

Acknowledgements

First of all I would like to thank my examiner Karl-Filip Faxén, my supervisors Fredrik Lundevall and Ragnar Andersson and my oponent Mikael Vidstedt. I would also like to thank Assar Westerlund, Magnus Ahltorp and Love Hörnqvist-Åstrand for their invaluable help in finding material and understanding AFS and Arla. For their comments and opinions about replication in Arla I would like to thank Björn Grönvall, Robert Watson, Karsten Thygesen, Mikael Andersson and Mårten Zimmerman. many thanks also to Harald Bart, Tomas Olson and all others who have contributed with comments and corrections to the report. Finally I would like to thank Ola Westin for all the support, comments and opinions during the project.

Appendix A

Pseudo-Code

Here is some pseudo-code for the coordinator and the participants in the two-phase commit protocols used for reads and writes, for the recovery protocol and for the election protocol. The pseudo-code is rough, mostly drawing the outlines for what the protocols should achieve. Many details need to be added when designing the final protocol.

A.1 Coordinator read one/write all, Write operation:

```
procedure getReplies():
  repeat the following until either NumberOfReplies=NumberOfParticipants,
                                    ReceivedNo=true or timeout:
    receive a reply
    NumberOfReplies := NumberOfReplies + 1
    check the reply value:
      if "yes" then NumberOfYes := NumberOfYes + 1
      if "no" then ReceivedNo := true
      if "busy" then add node to BusyNodes
  end repeat

  if a "no" is received (ReceivedNo = true) then
    log "cancel"
    send "cancel" to each participant
    if cause of the "no" message was a version error then
      recover and start over
    else
      return write and state cause to client
  else if NumberOfYes = NumberOfParticipants then
    log "commit"
    send "commit" to each participant
```

```
    commit the write
    return write successfully
end(getReplies);

procedure write():
  calculate new version etc.
  log "precommit" with data, file id, coordinator id, commit id etc.
  reset NumberOfReplies, NumberOfYes, BusyNodes, ReceivedNo etc.
  send "precommit" with parameters to each participant
  run getReplies()

  repeat until no BusyNodes left:
    if any BusyNodes then
      signal to client that volume is busy
      run getReplies() again
end(write);
```

A.2 Participant read one/write all and majority consensus, Write operation:

```
procedure precommit(parameters):
  log "precommit" with parameters

  if the suggested version is lower than own version then
    log reply "no"
    send reply "no" with the cause "version error"
    return

  if the suggested version is higher than own version + 1 then
    recover

  if busy then
    send reply "busy"
    wait until not busy

  if there is a precommit pending for the same file then
    log reply "no"
    send reply "no"
    return

  if can commit then
    log reply "yes"
```

```
    send reply "yes"
    return
  else
    log reply "no"
    send reply "no" stating cause
    return
end(precommit);

procedure commit():
  log "commit"
  commit the data
end(commit);

procedure cancel():
  log "cancel"
end(cancel);
```

A.3 Read one/write all, Read operation

```
procedure read():
  read the data and send it to the client
end(read);
```

A.4 Read one/write all, Recovery

```
procedure recover():
  for each volume do
    for each unresolved precommit with coordinator other than self in log do
      send "query" to coordinator
      wait for reply (commit, cancel, not ready)
      if timeout then send "query" to each other participant
                                until reply is received
      if result is "commit" or "cancel" then
        log result
      if result was "commit" then
        commit the data
    end for

    for each unresolved precommit with coordinator=self do
      log "cancel"
      send "cancel" to all participants
```

```
    end for
  end for
end(recover);

procedure query():
  look up the result of the precommit in the log
  if a result exists then
    send result
  if no result exists and coordinator=self then
    send "not ready"
end(query)
```

A.5 Coordinator majority consensus, Write operation

```
procedure getReplies():
  repeat the following until either NumberOfReplies=NumberOfParticipants,
                                    NumberOf{Yes,No}=NumberOfParticipants/2
                                    or timeout:
    receive a reply
    NumberOfReplies := NumberOfReplies + 1
    check the reply value:
      if "yes" then NumberOfYes := NumberOfYes + 1
      if "no" then NumberOfNo := NumberOfNo + 1
      if a "no" is received and cause of "no" was a version error then
recover and start over
      if "busy" then add node to BusyNodes
  end repeat


  if NumberOfYes = NumberOfParticipants/2 then
    log "commit"
    send "commit" to each participant
    commit the write
    return write successfully
  else if NumberOfNo = NumberOfParticipants/2 then
    log "cancel"
    send "cancel" to each participant
    else
      return write and state cause to client
end(getReplies);
```

```
procedure write():
  calculate new version etc.
  log "precommit" with data, file id, coordinator id, commit id etc.
  reset NumberOfReplies, NumberOfYes, NumberOfNo, BusyNodes, etc.
  send "precommit" with parameters to each participant
  run getReplies()

  if getReplies doesn't return the write and there are busy nodes then
    signal to client that volume is busy
    run getReplies() again
end(write);
```

A.6 Coordinator majority consensus, Read operation:

```
procedure getReplies():
  repeat the following until either NumberOfReplies=NumberOfParticipants/2
                                    or timeout:
    receive a reply
    NumberOfReplies := NumberOfReplies + 1
  end repeat
end(getReplies);

procedure read():
  send "versionQuery" with parameters to each participant
  run getReplies()

  while getReplies times out repeat the following
    signal to client that volume is busy
    run getReplies() again
  end repeat

  if a participant has a higher version of the data then
    recover

  read the data and send it to the client
end(read);
```

A.7 Participant majority consensus, Read operation

```
procedure versionQuery(parameters):
  send the requested version information to the coordinator
end(versionQuery)
```

A.8 Majority consensus, Recovery

```
procedure getReplies():
  repeat the following until NumberOfReplies=NumberOfParticipants/2:
    receive a reply
    NumberOfReplies := NumberOfReplies + 1
  end repeat
end(getReplies);

procedure recocery():
  for each volume do
    reset NumberOfReplies
    send "query" to all other nodes
    run getReplies()
    choose the node with the highest version numbers
    send "getLog" to chosen node stating the last committed/cancelled
                                             entry in log
    wait for reply
    if timeout then
      choose another node if you can and try again
    if a log is received then
      save the received log and go through it to catch up
    else
      copy the complete data

    for each unresolved precommit with coordinator=self do
      log "cancel"
      send "cancel" to all participants
    end for
  end for
end(recover);

procedure query():
  send current version numbers
end(query);
```

```
procedure getLog(Lastlog):
  look up Lastlog in the log
  if Lastlog is still in log then
    send rest of the log
  else
    send complete data
end(getLog);
```

A.9 Election, coordinator

```
procedure startElection()
  reset NumberOfVotes
  send ``voteForMe'' to all other nodes
  repeat the following until either voteForMe is received from other node or
                             NumberOfReplies > NumberOfNodes/2 or
                             timeout:
    receive vote
    NumberOfVotes := NumberOfVotes +1
 end repeat

 if a voteForMe was received from another node then
   vote
 if timeout before enough votes then
   abort own election

 else
   send ``newView'' to all other nodes with two-phase commit
   if two-phase commit fails then
     startElection
   else
     start acting as primary
end(startElection);

procedure vote()   /* run when a voteForMe is received */
  if other node has same or higher view and
     other node has higher priority then
       send vote to other node
       if own election is going on then
         abort own election
  else
    do nothing
end(vote);
```

Appendix B

Advanced Replication Algorithms

B.1 Overview

In this section I have collected the descriptions of various advanced replication algorithms and concepts that I have studied in this project but found only marginally interesting for this report. Most of these are mentioned or referred to previously in the report. This appendix is for the readers who might be interested in reading a bit more about what lies at the outskirts of this project.

B.2 Voting and Directories

The most common way to keep track of the latest version is to use incremental version numbers for all items. When using voting algorithms for replicated directories some problems may arise with version numbering of directory items. In [3] it is noted that since concurrent updates of a directory are possible version numbering is difficult and produces a lot of traffic. The authors suggest weighted voting algorithm for directories where version numbers are given to both the items in the directory and the "gaps" between them.

A gap is the "space" between two ordered objects where another object might be inserted (in their examples the items are ordered alphabetically). When an item is inserted it splits a gap in two. When the object is removed the two gaps are merged together and the "new" gap gets the latest version number. If one node has missed the removal of the file it can see that the gap has a higher version number than the file and so the file must have been removed.

In systems where update messages are logged, like Bayou and many epidemic systems, the version numbering problem of directories can be overcome by sending and logging delete messages for files. If all log entries and sent messages are ordered correctly there will be no question whether the file is deleted or not.

B.3 Volatile Witnesses

In [20] Pâris describes an available copy protocol with volatile witnesses. Volatile witnesses reside only in volatile memory, which makes them faster. They will lose all information at a server failure. In Pâris' protocol the network is assumed to consist of segments separated by gateways. One segment is the main segment, containing the largest number of nodes. The segments are assumed to have failsafe connection between the nodes, so that the only possible partition points are the gateways. Nodes within a segment are guaranteed to get all updates within that segment.

Nodes with replicas are divided into *local nodes*, which reside on the main network segment and can communicate directly with each other, and *non-local nodes*, which communicate with other nodes through one or several gateways. All non-local nodes keep a *witness* on the main segment or on a gateway directly connected to the main segment.

All nodes and witnesses keep track of the current version numbers for the data and the set of available nodes and replicas at the latest update or reconciliation (the *was-available* set). Nodes are considered available if they have synchronized correctly after the last crash.

Updates are allowed as long as there is at least one of the following constellations available:

- one available local node
- one available non-local node and one available witness with the same version number
- a node with a was-available set consisting of only itself, which is equivalent to that a node is the last available replica.

Updates are sent to all available replicas and witnesses. Data can be read from any of the above constellations. Also a recovering node can synchronize its data from the same combinations.

This protocol is a good example of the use of witnesses to enhance the performance of a protocol. The available copy protocol with witnesses is better than the simple available copy protocol since it can handle network partitions and still be pessimistic. The availability is increased without increasing the number of nodes.

B.4 Dynamic Voting

Dynamic voting is an extension to the voting algorithm. The idea is to count only the members in the latest majority partition if it becomes partitioned itself, thus allowing several consequent network partitions and still letting one partition continue operation. Dynamic voting has been presented and developed by several authors. The latest model was presented by Jajodia and Mutchler in [10].

When a network is partitioned the partition with the majority of the nodes, called *distinguished partition*, is allowed to continue service. At the same time the nodes in the distinguished partition note the number of nodes - the site cardinality - in that partition. If the distinguished partition becomes partitioned the part which contains the majority of the previous distinguished partition is appointed as new distinguished partition. The new distinguished partition updates its site cardinality to the current number of nodes in the distinguished partition.

For example, say that the nodes A, B, C, D and E hold the replicas of a certain file. At first, they all know that there are five nodes with copies of that file. All have the site cardinality value 5. If E then receives an update to the file and finds that it has lost contact with A and B it can propagate the update to C and D, since they form the distinguished partition, being the majority. Their new site cardinality is now 3. If E then loses contact with C and D, too, it can not continue service, since it is a minority of the previous three nodes. C and D can together form the new distinguished partition.

In the case of a tie, there must be some way to choose a partition which can continue service. For this Jajodia and Mutchler propose the extension *dynamic voting with linearly ordered copies*, also called *dynamic-linear voting* or *lexicographic dynamic voting*. The nodes are ordered in a priority order and the highest ordered up to date node in the distinguished partition is called the *distinguished site*. At an even partitioning the partition which holds the distinguished site forms the new distinguished partition. If an uneven partitioning occurs and the distinguished site does not belong to the distinguished partition a new distinguished site is appointed in the distinguished partition.

As an example, take the group of nodes A, B, C, D and E. Ordering is done in, say, alphabetical order, so that at first A is the distinguished site. If E loses contact with A it forms a new distinguished partition together with B, C and D, appointing B as distinguished site. If E and D then lose contact with B and C they realise that the partitions are even and that the distinguished site is in the other partition. Thus B and C form the new distinguished partition and D and E cease service.

The distinguished partition must also contain at least one node that is up to date. To determine this there are two version numbers associated with each copy of a file: the physical version number and the logical version number.

The physical version number is updated whenever the file is updated, either because of a write operation or when a node regains contact with other nodes and is updated by the other nodes.

The logical version number is updated when a node regains contact with the distinguished partition or when a write is committed in the distinguished partition. All nodes with the same logical version number have the same update site cardinality value (number of nodes in the distinguished partition they have last belonged to).

The distinction of logical and physical version numbers allow nodes to update each other even if they aren't in the distinguished partition. It also allows an up to date node to form a distinguished partition together with a node which is only logically updated but not physically. The physical update can then be done at any convenient time.

If a partition doesn't contain any node with the same physical version number as the highest logical version number in the partition, the partition cannot become the distinguished partition even if it contains the majority of the nodes.

A drawback with dynamic voting is that it generates a lot of network traffic when a new distinguished partition is formed. All nodes need to be aware of the current status of the other nodes.

B.5 Advanced Quorum Protocols

There have been many suggestions for advanced quorum protocols, in which the quora in a large system are much smaller than the usual (N+1)/2. The nodes are arranged in virtual patterns, like trees, grids or groups, and the quora are chosen from the patterns. These algorithms are often designed for database systems, which can have tens of replicas of the same data items. Few of these protocols show significant improvements for systems with only a few replicas. Since file systems seldom need that many replicas I will only show an example of such a protocol.

Pâris and Sloope suggested in [19] a dynamic group protocol suitable for distributed systems with many replicas at different nodes. It allows for $n-2$ successive replica failures in a system with n replicas. It also requires only $O\sqrt{n}$ messages per access.

The principle of the protocol is to organise the nodes into a number of groups. It is recommended that the groups are of equal size. A write quorum must contain one node from each group and one complete group. A read quorum consists of either a complete group or a node from each group. All nodes in the quora must be operational. This ensures that read quora and write quora intersect, and that all write quora intersect with each other.

The nodes are ordered lexicographically. Groups are assigned in lexicographic order, and to "fill up" the last group it is possible to let nodes from the first group to be members of also the last group. For example the nodes A-G could form the groups ({A B C}{D E F}{G A B}). Since A and B appear in two groups it is sometimes easier to find a write quorum, for example {A B C D} would qualify.

In addition the protocol uses dynamic group reordering. Groups are reordered whenever a node fails to respond to an access request or a node recovers from failure. At reordering the number of groups can change but the size of the groups remains constant. To assure mutual exclusion at regrouping a write quorum must first be achieved. This allows only one partition to regroup after a network partition.

Regrouping works in the same way as forming the original groups but using only the operational nodes. Suppose node B would fail in the previous example. Then the new groups would be ({A C D}{E F G}). The third group is removed, since the last node G moved to the second group. After regrouping a write quorum from the new groups is chosen for update, to ensure that all groups contain at least one current copy. In the case when there are only two groups left the protocol reverts to the previously described dynamic linear voting protocol.

The protocol has a few weaknesses. First of all it doesn't tolerate the simultaneous failure or partition of a whole group. It also requires a sufficient access rate to detect failures in a reasonable time. Both problems can perhaps be overcome, but the authors also conclude that further investigation is necessary.

B.6 View Consistency

Goel and Popek [9] have implemented view consistency guarantees on top of the Ficus distributed file system. Ficus is an optimistically replicated file system. A more detailed description of Ficus can be found in section D.

In their system clients or groups of clients, called entities, take care of the view consistency. Thus it can be implemented without changing the servers. The only requirement is that an entity can access version information for the data. This can be done separately, but to increase performance the version number should be sent together with the file data. In Ficus this requires a slight modification of the server.

Each entity stores a database of view-entries for each accessed file, containing information about the last accessed version and replica. View-entries are created for each file that is accessed when the file is accessed for the first time. Each time a file is accessed the corresponding view-entry is modified. View-entries are cached in the kernel together with the files' vnodes, which reduces the amount of lookups in the view-entry database. View-entries can be garbage-collected when all file replica versions are known to be later than the view-entry version. Entities can be persistent or transient. Databases for persistent entities are stored in stable storage while databases for transient entities are removed when the entity terminates. Entities can also be distributed.

It is sometimes necessary for a client to change access to some other replica. This occurs when the current replica becomes unavailable, gets too high load compared to other replicas or becomes inconsistent. In this algorithm access is changed if a view-consistent replica with significantly lower access time is found. In the case when no consistent replica can be found (the previously accessed replica becomes unavailable) the file becomes unavailable for that entity. It can still be available to other entities with other views.

Some details are not yet finished at the time of the publication of the article, for example distribution of entities and the deletion algorithm for files.

The modified Andrew Benchmark (mab) and a series of tests with commonly used file operations like cp and ls shows an overhead of 5 to 12 percent when comparing a view consistent Ficus implementation to a standard Ficus implementation. A grep benchmark shows an overhead of 185 percent if version information is obtained separately, but less than 5 percent if version information is received together with the data. This is because the cost of running grep on a single file is only about twice as much as the cost of getting version information for the file.

B.7 Dynamic Replica Placement

For better performance in large systems replicas should be able to be created where they are needed most. It is even better if this is done automatically. In [16] this is called *fluid replication*. Replicas are automatically created when a client gets too low access performance. When the replica is not in use anymore it is automatically deleted.

Each file has a home server, which acts as a kind of master replica. In addition there are several *way stations*, which can hold replicas on demand. The way stations are spread out in the network so that they can give fast access to the clients.

Performance measurements are made all the time, and when the client finds that the access time to a file has grown too much it tries to locate a way station with shorter access time. For this a special multicast is used. The client asks for a way station with a predefined maximum latency. It also tells what service it wants replicated. After receiving replies from the way stations it chooses the closest way station or a way station with acceptable latency that has already got a replica of the service asked for.

Three different models for consistency can be used. Latest-write lets writes be ordered by timestamps, and in the case of conflict the latest write wins. Optimistic consistency lets conflicts happen but detects them soon and calls for resolution. Pessimistic consistency demands that the writer gains exclusive access to a file before writing. The consistency model can be decided individually for each file.

The way stations are responsible for forwarding updates to the home server. This is done periodically. Updates are forwarded as operations and are logged. As an optimization self-cancelling operations don't have to be forwarded. An example of this is when a file is created and deleted a short while later. The home server can remove old entries from its logs when it sees that all replicas have received the particular updates. When a replica gets outdated it can either be invalidated or actively updated by pulling updates from the home server. The strategy can be chosen depending on the situation.

Replicas are destroyed when no clients no longer access them. Replicas can also migrate from one way station to another if the client using it wishes to use the other way station. The first way station must first send all updates to the home server. The second way station will then retrieve a replica of the file from the home service.

At the time of the publication of [16] only a prototype has been implemented. Tests and simulations still need to be done. Still, the concept is worth notice.

Appendix C

Evaluation of Algorithms

Some amount of work has been done to evaluate and compare different replication algorithms. Evaluation has been done either by simulation, with "real life" tests or mathematically. Often simulations have been done by the authors of the algorithms, but there have also been several "external" evaluations and comparisons of algorithms. Another topic for studies is optimization of variables in the algorithms.

Many of the algorithms have been designed and evaluated several years ago on hardware and software that today seem nearly antique. Thus many simulations and tests can be assumed unreliable. Even comparisons between algorithms might show different results today, since the bottlenecks might have moved to favor other algorithms. Thus it is difficult to find relevant studies. Here I will present the results from three papers evaluating, comparing or optimizing algorithms.

C.1 Evaluation

In [11] Johnson and Raab present a tight upper bound on the performance of mutually exclusive replication protocols. They show that if the "availability" is $A, 0 <= A <= 1$, for a single copy protocol with optimal placement of the copy, then the upper bound for availability in a replicated system which maintains consistency and mutual exclusion is $\sqrt{A}$.

Three protocols for mutual exclusion are considered. The reference protocol all other protocols are compared with is the *single copy protocol*, where all access to a file goes through one server. This can either be a primary copy of multiple replicas or a non-replicated single server. Later the single copy protocol is generalized to a *relocatable single copy protocol*, where the copy can be moved. The availability of the single copy protocol is equal to the probability that the single copy is available.

Secondly, an optimal *best component* protocol is considered. The best component protocol is purely theoretical, since it requires complete knowledge of the network state. In the paper the best component protocol is shown to give equal or better availability than any other mutual exclusion protocol.

The third protocol considered is the majority consensus voting protocol. It is used as an example of a simple protocol that provides mutual exclusion. The majority consensus protocol is not optimal in most cases and is thus used to show that the upper bound of availability is tight.

In the paper it is proved that there is some sequence of primary copies that if a relocatable single copy protocol is used the accessibility for the best copy protocol is the square root of the accessibility of the single copy protocol. They also show that this upper bound is tight and that the upper bound can not be improved if the availability is more than 0.25.

In addition, finding the optimal placement for a single copy is discussed. Though the problem is impossible in theory, it is often feasible in practice.

The value of this work is that it shows that replication (with guaranteed consistency) is not always worth the trouble, since it might in well-planned networks give very little improvement to the performance.

C.2 Comparison

Comparison of protocols by simulation has been done (among others) by Pâris, Long and Glockner in [18]. They simulated a network environment with eight nodes. For eleven different configurations with replicas on three to five nodes they measured the unavailability proportions and mean duration of unavailabilities for three protocols: majority consensus voting, dynamic voting and dynamic linear voting (a.k.a. lexicographic dynamic voting).

The network was designed so that it contained a few, well defined possible partition points. Software and hardware failure intervals and durations were different for different nodes. In this study a file was considered available if there existed a distinguished partition, not counting the fact that users might not have contact with this partition when the network is partitioned.

The eleven configurations were designed so that some could be partitioned at one or two points. Four of the configurations used three replicas, four had four replicas and the last three had five replicas. More than five replicas were considered too costly to uphold for a network this small.

The result contained few real surprises, showing that dynamic linear voting gives higher availability than majority consensus voting and dynamic voting for all configurations. Dynamic voting showed in some configurations lower performance than majority consensus voting. This was the case in configurations where the network is often partitioned into two equally large parts.

Though the comparison between the three protocols gave few surprises the study was of value by pointing out the value of placing the replicas optimally. Some protocols would give very low performance for certain configurations, for example if a majority consensus protocol is used in a system with many replicas and many possible partition points. Still, on some points the study was incomplete. For example, measuring response times or message overheads to uphold the protocol were not a part of the study. High protocol overheads might well be considered worse than lower availability.

C.3 Optimization

One reason for evaluation of algorithms is to find optimal solutions. It can also be important just to show that there is a limited, optimal solution.

Optimal replication degrees and quorum assignments are studied in [1]. For two protocols, read one/write all and majority consensus voting, the authors try to determine the optimal quorum assignments and degrees of replication. Two types of behavior when service is unavailable are studied: either the transaction aborts or it stalls until service becomes available. Their work is mainly focused on distributed database systems, but can be interesting also for file systems.

Their system model consists of a number of identical nodes, each having a replica of the data and each having one vote. Transactions are either read or write transactions and arrive as a Poisson process. They use a standard quorum consensus protocol, so that a read quorum must be accessed to read a file and writes must be done to a write quorum. Service is assumed to be instant.

In the first model, where a transaction aborts if a quorum is unavailable the degree of successful operations is to be optimized. For read one/write all they find mathematically a limit for the optimal number of replicas. If majority consensus voting is used performance will increase when the number of nodes is increased. Also they find a mathematical formula for the optimal quorum assignment. These results are also tested numerically.

In the second model, where transactions stall until they can be completed, the mean transaction time needs to be minimised. For this model they only use numerical analysis to find the optimal assignments. Not surprisingly they find that read one/write all gets lower performance at higher degrees of replication while higher degree of replication for a majority consensus protocol increases availability. Depending on the mix of read and write transactions different values for read and write quora are optimal. A majority consensus protocol (majority needed for both reads and writes) is optimal for many mixes, while read one/write all is best for systems with mostly reads and few writes.

This work is interesting since it shows that quorum assignments and degrees of replication can be optimised depending mainly on the mix of read and write operations.

Appendix D

Case Studies

D.1 Overview

Many distributed file systems using read-write replication have been created throughout the years. Most of them are not in use anymore, and some never even got used more than in the lab that made them. Some rely on specific hardware configurations and are thus quite difficult to adapt to other environments. Still, it is interesting to look at what has been done and how well the projects have succeeded.

In this appendix I will present some distributed file systems. First, NFS is described briefly. Though it doesn't use replication it is very important, since many other file systems are based on NFS. Thereafter two optimistic replicated file systems, Ficus and Coda, are presented. The next two file systems, Echo and Harp use primary copy schemes. Harp also uses witnesses. Deceit uses a kind of quorum-consensus approach, combined with write tokens. Another token-based system is Huygens, which uses a virtual ring for communication and failure detection. The next system, Bayou, is a storage system using epidemic update propagation. The Frolic replication scheme gives replication-on-demand in wide-area systems. The last file system, HA-NFS, is an example of a file system using special hardware configurations to achieve higher availability.

D.2 NFS

Sun's Network File System Protocol (NFS), described in [31] and [32], forms the basis for many distributed operating systems. NFS is a protocol for the client-server communication. It hides the system specific details, like transport protocols and data representation formats. Servers using completely different operating systems and hardware can communicate with each other using NFS.

NFS is based on Remote Procedure Call (RPC) which provides an abstraction above the transport layer. This allows for many different transport protocols to be used. Also there is a specified eXternal Data Representation (XDR) format to provide a standard data representation.

The NFS protocol is designed to be stateless. This increases failure tolerance, since no state needs to be restored after a recovery. Separate services implement stateful operations like locking. All procedures in NFS are synchronous. In NFS 3 there is one exception, which is an asynchronous write. Some state can be kept to increase performance, like a read-ahead cache.

The file system consists of files and directories with string names. Since different operating systems may have different path representations NFS looks up one component of the path at a time.

Several types of authentication are supported. With standard UNIX authentication the servers must use the same user ID lists or map user and group ID:s locally. It is also possible to use DES encryption or Kerberos authentication (in NFS 3), in which case the user and group ID:s are global.

D.3 Ficus

D.3.1 Overview

Ficus ([17], [22]) is an optimistically replicated file system developed at the Department of Computer Science at University of California, Los Angeles. The goals of the project are both to test the concept of optimistic replication and to build an usable system.

Ficus is based on *single-copy availability*, which means that a file can be updated as long as there is at least one replica available. It provides a guarantee that no updates are lost. Conflicts are reliably detected and dealt with either automatically or manually. Ficus is a client-server system, but is designed so that there can be both a client and a server running on each machine. This makes disconnected use easy to provide, since you only have to replicate the files you need to the file server on your laptop and disconnect.

The Ficus file system implements the VFS interface to the user and uses the standard Unix File System (UFS) for storage. Files are kept in volumes, similarly to the AFS file system. For connecting volumes to the file system Ficus uses a technique called grafting, which is similar to mounting. Volume location information is kept in the grafting point, so a separate volume location database is unnecessary.

D.3.2 Replication

In Ficus each file is replicated individually. A replica is chosen by the client when the file is opened, and will serve consequent requests from the client unless it goes down or is partitioned from the client. This ensures consistency to the client as long as the chosen replica is available.

The replication scheme is fairly simple. When a file is updated the client sends the update to the chosen replica, which then sends out update notification messages to all other replicas. The update is pulled by the other nodes asynchronously. If a node is unreachable and does not receive the update notification the file system will be inconsistent until the node becomes reachable by some of the updated nodes.

The optimistic replication makes the system highly available to the users. The reconciliation scheme described below ensures that all updates are eventually spread to all replicas as long as no replica becomes permanently disconnected. While reconciliation is epidemic, the updates are at first spread with an available copy scheme, which makes update propagation faster than if a purely epidemic scheme was used.

D.3.3 Dealing with Conflicts

The no lost updates guarantee leads inevitably to conflicts when for example a file is edited in both partitions of a partitioned network. The authors claim that such conflicts are rare, showing conflict statistics in [17] to confirm their theory. During the nine month long statistic gathering time the update/update conflict rate was 0.0035 % for all volumes. Also other conflict types are shown to be rare.

Dealing with conflicts consists of two parts: detecting a conflict and resolving it. Conflict detection is done pairwise between the nodes. Periodically each node contact one other node to reconciliate. The nodes compare their version vectors, which contain information about all updates known to the node. If the version vectors are identical there is no need to reconciliate. If the version vector of one node is strictly higher than the other, the other node will pull the latest version and update its version vector to a copy of the higher. If the vectors are different but none of them is strictly later, there is a update/update conflict and a resolver is invoked to solve the conflict.

If the update semantics for a file is known an automatic resolver can be made to handle conflicts in files of that type. For directories, the resolver is supplied by the file system. There are many types of conflicts that can occur in directories, and Ficus's solutions for these are discussed at length in [17]. When a resolver can't be found for a certain conflict an e-mail is sent to the user(s) involved and they will have to resolve the conflict manually.

D.3.4 Evaluation

In tests with the Modified Andrew Benchmark (MAB) operations in Ficus show a medium time overhead of about 10-25 % compared to non-replicated UFS. Operations like copy show much higher overheads, since they are highly dependent on the number of replicas.

Since the whole file is pulled when an update notification is received high update rates give very high load on the replicating servers. This could be optimized by allowing the nodes to pull only parts of the updated files or using a log-based update propagation.

Ficus is an example of a optimistic distributed file system that works. Conflicts are reasonably rare and the automated resolvers take care of most of them. The main administrative costs lie in deciding which files to replicate and designing the network layout.

D.4 Coda

D.4.1 Overview

Coda was developed at Carnegie-Mellon University by partly the same team that developed AFS. Coda has inherited many features from AFS, like the client-server model, caching clients and the consistency model in the absence of failures. The main difference is that Coda allows optimistic read-write replication in addition to read-only replication and that Coda has been designed to allow disconnected use, which increases availability even more.

Coda has been described briefly in [27]. An older but more detailed description can be found in [25]. More information can also be found on the WWW at Coda's web site [www.coda.org] and in [5].

D.4.2 Replication and disconnected operation

The replication scheme in Coda is an optimistic *read one write all*, which means that reads can be done from any available replicating node, and writes are done to all available nodes which hold a replica of the current volume. This makes the system highly available, since both reads and writes to a volume are allowed as long as any node with a replica of the volume is reachable. In addition, the client will go into disconnected mode if all nodes become unreachable.

As in AFS, the replication unit in Coda is the volume. Each volume is associated with a Volume Storage Group (VSG), which is the group of servers replicating the volume. The system also keeps track of which servers in the VSG that are accessible at the moment, calling this group the Accessible VSG (AVSG). The client has to keep track of the AVSG by periodically probing the members in the AVSG and VSG.

A file in a replicated volume can be read from any server in the AVSG. At a cache miss in the client a *preferred server* is appointed to that file, chosen randomly or by some criteria like proximity or load. The preferred server contacts other servers in the AVSG to check that it is consistent. If it isn't, the inconsistency must be resolved before operations can be done to the file. If the preferred server finds more recent (strictly newer) updates a new preferred server is appointed among the updated servers and the system is notified that a refresh is needed.

When a file is opened a callback is established to the preferred server. The callback is broken whenever the preferred server gets an update or the client detects mismatches in the volume version vectors. Also when the AVSG grows all callbacks are dropped, since the new member might be more recently updated.

Disconnected operation is handled as an exceptional state. At first possible occasion the client goes into normal connected state. During disconnected operation all files are stored in and read from the cache. A cache miss will mean that work cannot proceed until the client becomes connected again. Thus cache misses are to be avoided. To diminish the probability of cache misses the user can specify a set of files which should be retained in the cache as much as possible. It is also possible to let Coda determine which files are needed for a specific set of actions and then mark these files as "sticky".

Since many operations are to be done on several servers at the same time Coda uses a special MultiRPC parallel remote procedure call together with hardware multicast.

D.4.3 Dealing with Conflicts

When the network is partitioned or during disconnected operation a file can be updated simultaneously on nodes without contact with each other, which can lead to inconsistencies. Inconsistencies are dealt with by detection and resolution. A version vector is used to compare the updates of servers, and in the case of a difference the system will try to repair it. Strictly newer versions can update older versions easily, and conflicts can be resolved by Coda or by application specific resolvers. When the system is unable to resolve the conflict the user is notified for manual repair.

As a curiosity the authors mention another practical use for resolution. At a disk failure the broken disk can be replaced with a new, empty disk. Then resolution is ordered and the disk is automatically brought up to date.

D.4.4 Evaluation

In tests with benchmarks non-replicated Coda gets about 21% time-overhead compared to the standard Unix file system. With replication this rises with a few percent for each replica. Copy is the slowest operation, giving 73% overhead with unreplicated coda and over 100% extra with a few replicas. The authors compare this to AFS and point out that AFS has similar overheads compared to UFS.

D.5 Echo

D.5.1 Overview

The distributed file system Echo, presented in [4] uses a combination of optimistic and pessimistic primary copy replication. In Echo files are stored in two types of *volumes*. *Name service volumes* contain mainly directory hierarchies while *filestore volumes* contain files. The name service volumes are optimistically replicated to give high availability while filestore volumes are pessimistically replicated to maintain consistency. There is also a global root volume.

Location of a file is done in three steps. The first part of the path is the domain name of the site where the file is kept, for example "/com/dec/src" for "src.dec.com". The Echo name service uses ordinary Domain Name Service (DNS) for resolving this part. The next part of the path consists normally of name service volumes and is resolved by the Echo name service. Finally the last part of the path consists of filestore volumes and a filename, which are resolved by the Echo file service.

Echo caches files on the client computers. The cache is kept in volatile memory for speed. Updates are write-behind, so that they are flushed to the server at file close. A token system is used for keeping cache coherence. A write token must be obtained to write to a file, and for reading a read token must be obtained. Several read tokens can exist simultaneously for a file, but the write token is exclusive and neither another write token nor any read tokens may exist for the same file simultaneously. This is not the same as locking of the file, since the token is only needed when the read or write is actually committed.

When a write is to be committed to the server the client requests a write token from the primary server. If a write token is already given to another client the token is revoked. The client which previously held the token commits all writes in queue before giving back the token. Read tokens are simply revoked. When no other client holds a token for the file the requesting client can get a write token. Read tokens are handed out in a similar way. This approach is inefficient if many clients share the same file but was sufficient in the environment where Echo was used.

D.5.2 Replication

The original replication algorithm in Echo is described in [14], but some modifications have been done in the algorithm described in [4].

As mentioned earlier there are two forms of replication in Echo. Name service volumes are optimistically replicated - reads and writes can be done at any replica. Updates are propagated asynchronously to the other replicas. This increases the availability of directories.

Filestore volumes are pessimistically replicated using a primary copy scheme. There are several possible configurations for the servers and disks: one server and disk, two servers and one disk, two servers and two disks, two disks but only one server etc.. Election of the primary server is done with weighted majority consensus voting, where the votes are assigned with regard of the configuration of disks and servers. Witnesses can be included to make the number of servers odd.

All file system calls are sent to the primary copy. Updates are written to disk at the primary server and recorded to a log. Log entries are sent to all replicas before the write operation returns. The other replicas commit log entries asynchronously.

Read and write tokens for files are also replicated, so that they will not be lost if the primary server crashes.

D.5.3 Election and Recovery

The primary (master) server keeps track of the slave servers by periodically contacting them. If either the master fails to receive an answer from a slave or a slave doesn't get contacted in time it starts election. Also if the master manages to contact a slave which is not recorded as alive it will start election.

Periods between elections are ordered with *epoch* numbers. After each election a new epoch number is recorded. Several epoch variables are used to keep track of how recently a node is updated. At an election the node that has the highest priority among all reachable up to date nodes is voted for as the new master. If a node can collect a majority of the votes it becomes the new primary copy.

Reconciliation of a recovering node is done simply by sending it the update logs for all updates it has missed. If the update logs aren't old enough reconciliation is done by completely copying the contents of an up to date node.

D.5.4 Evaluation

Neither of the articles referenced above show any figures for the performance of Echo. Echo was in use for almost two years and was used by about 50 researchers. Echo offered good reliability: during the period there were 10 disk failures, none of which caused any loss of data. Performance was thought to be good and the bottleneck was in the hardware. The development of Echo was stopped because the operating system on which it was running would not be ported to modern platforms.

D.6 Harp

D.6.1 Overview

Harp is a pessimistic replicated file system to be run within a distributed file service based on NFS. The replication scheme in Harp is a pessimistic primary copy scheme. It can be used with any VFS-based NFS compatible implementation and guarantees the same semantics as NFS. Harp was designed at the Laboratory for Computer Science at MIT and is described in [13].

Every file server in Harp logs the updates to files. To increase performance logs are kept in volatile memory. Every server is equipped with an UPS to give the system time to save the logs in permanent storage in the case of a power failure. There are also mechanisms for recovery of the logs after a software failure.

D.6.2 Replication

In harp each file is managed by a group of servers. One server in the group is the primary and the rest are backups. Both read and write operations are handled by the primary server. As in most pessimistic systems a majority of the nodes must be operational in order to continue service.

One of the servers in the group is appointed as *designated primary*, which means that it will act as primary server whenever it can. Half of the rest of the servers are *designated backups*, which store full copies of the files. The rest of the nodes are *witnesses* and store only version information. The designated primary and the designated backups form together a majority and thus there will always be at least one full copy of a file available if at least a majority of the nodes are up.

A write operation consists of a two-phase commit protocol. First the primary server sends log entries with the modification to the backups. When the backups receive the log entries they append them to their logs and send acknowledgements to the primary. To distinguish a committed part of the log from uncommitted parts each node indexes the last committed operation with a *commit point*. When the primary has received acknowledgements from the backups it advances its commit point and sends messages with the new commit point to the backups.

Committed changes are taken care of by a *apply process*, which performs the system operations for the operations in the log. Writes are carried out by the Unix file system. Another process keeps track of which writes have been committed to the disk. Records for updates that are completed are removed from the log. The nodes exchange information about how far they have committed.

Reads can be processed by the primary as long as it has had contact with the backups at most a certain time interval ago. The backups will always wait this long before they start an election. This is to prevent that a partitioned primary services a read while the other nodes have elected a new primary and have committed new writes.

D.6.3 Election

When a node crashes, becomes partitioned or recovers from a crash or network partition an election is done. In Harp this is called a view change. In the case of recovery from crash or network partition the recovering node will try to come up to date by communicating with a node with a higher view number and will then initiate a view chance. If a node finds that it cannot communicate with another node in the current view it also initiates a view change.

If the primary node has crashed or lost contact with the backups the backups will appoint a new primary among the designated backups. To compensate for the loss of a backup, both when a backup has crashed and when a backup is appointed as primary, a witness is *promoted* and will take part in all file operations. A promoted witness starts storing log messages in volatile memory and will not erase the log until it is demoted. If the log grows too large the oldest parts of it can be stored in non-volatile memory.

At the view change the coordinator (the initiator of the view change) suggests a view change to the other group members. When it has received enough answers from members which agree to a group change it stores a new view number on disk and updates nodes which lack parts of the initial state of the new view, for example by sending log records for uncompleted operations to promoted witnesses. Unpromoted witnesses can be kept in "standby" by letting the primary send log messages to them as well as to the backups.

D.6.4 Evaluation

Consistency in Harp is guaranteed by the replication algorithm, which gives the same semantics as for pure NFS. Availability is increased by allowing failures as long as a majority of the nodes are operational. When a witness is promoted it receives logs for all operations that haven't been completed at all the backups and the primary. This makes makes recovery quicker, since a failed node can copy the logs of a witness provided that it hasn't had a media failure.

Using volatile memory for storing logs increases performance but also introduces some risks. If both the primary copy and the backups crash all changes not yet written to disk will be lost. This sort of crashes were considered unlikely when Harp was designed.

In measurements using the Andrew and Nhfsstone benchmarks a Harp system with one group showed lower response times than unreplicated Unix, since disk writes are replaced with message roundtrips.

D.7 Deceit

D.7.1 Overview

Deceit [28] is a distributed file system which was developed at Cornell University. Deceit uses the standard NFS protocol, and can therefore be used with standard NFS clients. File servers are grouped in cells within the global name space. Replication is done within cells on demand with at least a defined minimum number of replicas. Each file is associated with a token holder replica, managing the number of replicas for that file. The token holder changes to whichever replica server got the latest update request. Requests to a file can be sent to any file server. If the server doesn't have the file it propagates the request automatically to a server which has.

D.7.2 Replication

For each file in Deceit there is a minimum replication level, a token holder and a file group. The file group is a group of servers which hold a replica of a file or have the file cached. The token holder is responsible for keeping up the number of replicas. Whenever it notices that there are too few replicas it will generate new replicas.

New replicas can also be created when a file is read from a file server that hasn't got a replica of the file. In addition to fetching the file from another node it asks the token holder to create a replica locally to increase future performance. Users can also ask for replicas on their local servers. Excess replicas are deleted in least-recently-used order by the token holder.

When a server receives a write request for a file it holds it must acquire the write token for that file. It sends out a token request to all other nodes in the file group. The node holding the token releases it and sends out a token pass message. Reads can served by any server holding a replica of the file.

After a node has acquired the write token, or when the token holder receives a write request, it synchronously sends out a message to all replicas marking the file as unstable. If it gets answers from at least a certain number of replicas assumed to be the safety level for that file (a kind of write quorum) it commits the update. If some nodes fail to answer new replicas are created. When the file hasn't been written to for a while it is marked stable.

D.7.3 Recovery From Failures

If a token is lost due to crash or when the network is partitioned a new token can be generated. A majority of the nodes must be reachable when a new token is generated. The number of nodes is assumed to be equal to the minimum number of replicas. To protect from the possibility that the file gets two tokens when a new token is generated a token generation implies that a whole new file is created.

Each file is associated with a version number pair ($v1v2$). When a new token is created while the network is partitioned the new file gets a higher version number $v1$. $v2$ is increased at each update. To see if a node is up to date only the version numbers need to be compared. A user can access different versions $v1$ of a file by indexing the file name with the version number. The version number are stored together with the files in non-volatile memory.

Since the number of replicas can be more than the minimum number of replicas it is possible to have a partition where both partitions can have a write token. If a file is updated in both partitions there will be two versions available of that file when the network partition is resolved. Resolution of conflicts in the file is then up to the user.

D.8 Huygens

D.8.1 Overview

Huygens [8] is a replicated file system implementing pessimistic replication with tokens. Files can be either unshared or read-shared, where writes are expensive for read-shared files.

Files are organized in volumes. Each volume has a set of servers which hold replicas of the files in the volume. Files in the volume are replicated individually, so that all files aren't stored at the same nodes. The nodes in a volume are ordered in *server groups*. All members in a server group can communicate with each other and know which other nodes are in the group. All nodes are members of one and only one server group.

Communication between the servers in a server group is sent through a virtual ring passing all servers. A keep-alive message is kept circulating through the ring to detect failures. When a failure is detected the ring will be rebuilt excluding the failed node. Separate rings (server groups) will also eventually merge if they gain contact with each other.

D.8.2 Replication and Token Management

There are two kinds of tokens in Huygens: read tokens and exclusive tokens. A node holding a replica of a file can either have a read token or an exclusive token. Read tokens can be held by all nodes for a file while only one node can hold an exclusive token for a file.

To serve a read request for a file a node must have a read token or an exclusive token. Nodes holding a read-shared file normally have a read token for the file. When a write request is issued, the node receiving the request must withdraw the read tokens from all other nodes to gain an exclusive token. The other nodes will remember which node holds the exclusive token.

Unshared files are simply files which reside on only one node holding permanently the exclusive token for the file. Unshared files can be transformed into read-shared files, but the process is expensive. Since writes to read-shared files also are expensive it is recommended that frequently written files are kept unshared.

D.8.3 Dealing with Failures

It is desirable that as much service as possible can be offered during a partition. When a partition occurs there can either be read tokens or an exclusive token for a file. The level of service that can be offered depends on the location of the tokens and on the configuration of the partitions.

If the server group for a file finds that it is complete even though parts of the volume have been partitioned it can continue normal service for the file.

When some nodes in the group are missing the level of service depends on the number of missing nodes. If the number of missing nodes in a volume are too few to form an active partition (less than a preset *threshold* value) service can continue as usual. If both partitions are found to have more than *threshold* number of nodes, more replicas will be generated so that there are *threshold* number of replicas in both partitions and the file is marked read-only until the network partition resolves. If a partition finds it has less than *threshold* nodes it will not offer service for that file.

When the network is partitioned during an update the write is attempted to be committed in both partitions. Then the file is marked read-only for the duration of the partition. If the node holding the exclusive token becomes completely isolated it cannot know whether the other partition can complete the write. In that case it must suspend operation until the partition is resolved.

D.9 Bayou

D.9.1 Overview

The Bayou storage system [21] is designed for use in a collaborative environment and uses anti-entropy to lazily propagate updates between servers. Bayou allows dynamic change of the number of replicas and reconciliation between any pair of replicas. The replication is optimistic, since inconsistencies can occur. Application-specific resolvers are used for conflict detection and resolution. Anti-entropy sees to that all replicas eventually receive all updates as long as they aren't permanently disconnected from each other.

Bayou uses access control based on public keys. A single well-known server signs certificates which grant different levels of access to file groups for users. Users with certain certificates for a file group may sign and revoke certificates for other users for the same file group. Authentication is done once per session between the client and the server.

D.9.2 Replication and Reconciliation

Replication in Bayou can be done with any number of nodes. Reads and writes can be sent to any replica. When a replica receives an update from a client it logs it and updates the local copy of the file. The nodes keep up an ordered log of the writes they have received form a client or by reconciliation. The logs are stored on disk. To reduce storage they can be truncated by the nodes at will. Because of this it is sometimes necessary to reconciliate by copying the whole database. The nodes may also choose not to perform reconciliation.

Each replicating node chooses one or several other nodes with which they reconciliate. Reconciliation is a one-way process done pairwise between the nodes. At reconciliation the receiving node sends its version vector to the sending node. The version vector maintains information of the versions known to be known by all other nodes. The sender then sends all writes which the receiver doesn't know of. Anti-entropy will see to that all updates will eventually reach all connected nodes.

Writes received from clients by a node are ordered in *accept order* by the node. Thus all write operations received by the same node will be totally ordered. This order is preserved during reconciliation. Writes committed concurrently at different nodes must be merged in a suitable way. Resolution of such conflicts is application-specific. Rollback of the log might sometimes be needed to insert writes earlier into the log. An undo-log, stored in volatile memory, is used for undoing the effects of write-log entries. A write is considered *stable* when it is certain that it will not need to be rolled back.

Replicas can be created and removed dynamically. A new replica announces itself by sending a *creation write* to another node. In return it gets a server ID based on the other server's ID and a timestamp. The new server will be allocated a place in the version vector and the creation write is propagated to other nodes in the same way as ordinary writes. Removal of a node is done in a similar way by sending out a *retirement write*.

The one-way nature of the reconciliation in Bayou makes it possible to reconciliate using portable media like a floppy disk. Log entries are copied to the disk and any server containing at least up to the oldest entry on the disk can be reconciliated from the disk.

D.9.3 Evaluation

Simulations have showed that the time it takes to perform anti-entropy between two nodes is linearly dependent of the numbers of missing writes. The anti-entropy time is also dependent on the size of the version vector. The creation pattern of the replicas determines the size of the version vector, since the server ID:s are recursive, each containing the server ID of the server first contacted when creating a new replica.

As usual there is a trade-off between storage and bandwidth - either large write-logs must be kept or full database transfers may become more frequent. Each server decides for itself how long logs to keep.

Bayou is interesting to look at since it is an example of a storage system based on anti-entropy, though the demand for application-specific resolution makes it unsuitable to be used as a base for a file system.

D.10 Frolic

D.10.1 Overview

Frolic [23] is a high-level scheme for replication in wide-area networks. It shows a slightly different approach to file usage than many other file systems. The model for a network is assumed to be a set of clusters, each containing file servers, connected via a backbone. The clusters can be spread far away from each other like the different offices of the same company. Users stay normally within the same cluster. Data is shared between clusters and people in different clusters work on the same projects and access the same files. In such a network the file access time may be high for a file located at another cluster. To improve availability Frolic uses "replication-on-demand" to create a replica in the local cluster. Locating a valid copy of a file and keeping consistency among the replicas are also discussed.

D.10.2 Replication

When a client requests a file that has its owner in another cluster a local replica is made in the local cluster. The file can be read from any replica, but only one cluster at a time has write access (called the *owner*). Frolic operates only on cluster-level, so access within the owner cluster is controlled by a lower level.

Four different techniques for locating the owner replica are discussed:

1. A central server is master for each file. This has the drawback that availability and latency is same as that of the central server.

2. All servers are asked with a multicast message. This is a simple technique but quite expensive.

3. The server wanting to find the owner asks the server which it last knew to be the owner. If that server isn't the owner any more it asks the server which took over the ownership from it etc. This has the drawback that in the worst cases all other servers must be asked before the owner is found.

4. A location server is used to keep track of file ownerships.

Three techniques for maintaining consistency are discussed:

1. Send updates to all servers when a file is closed. This is quite expensive.

2. Get updates when a file is accessed in a replica. This is less expensive since multiple updates are fetched at the same time. The drawback is that the access time suffers.

3. Send an update to one other server where the file is accessed often and let the other servers fetch updates when needed. This is a hybrid between 1 and 2.

No technique or policy for changing the owner of a file is discussed.

D.10.3 Evaluation

Simulations of the system were done with a simulator implemented in Csim. The simulation showed that with replication the access time remained almost constant while workload intensity increased. Without replication the access time increased notably. Also the backbone network utilization increased less with replication than without.

Increasing the number of clusters increased the access times, but replication gave still a more constant access time than no replication. Also the size of accessed portions of the files and the cluster locality were varied in the simulation. The results for these simulations also showed better performance than a non-replicated system in most cases.

Finally the different consistency and locating algorithms were studied. Of the consistency algorithms immediate update performed best at low workloads but worst at high loads while invalidation and partial update showed quite similar performances, with partial update slightly better. The locating algorithms were all very similar in performance.

D.11 HA-NFS

HA-NFS [2] stands for Highly Available Network File Server. It is based on NFS and implements NFS semantics. It was run on AIX version 3. HA-NFS increases availability through implicit replication.

In HA-NFS two different servers get access to the same string of SCSI disks. One server acts as primary and the other as backup. In normal operation only the primary has access to the disks. If the primary fails, the secondary will take over its IP address and the disks and thus impersonate the primary. The servers have two network interfaces, since they act as primary for one set of files and secondary for another and may thus need two network addresses.

If two networks are available the replication can be extended to include networks. The primary and secondary servers will be on different networks. If a client looses contact with a server it contacts the backup which will then impersonate the primary.

AIXv3 supports disk mirroring to give the impression of failsafe disks. This in combination with higher server and network availability increases availability during failures. The cost is slightly slower reads and clearly slower writes (with mirroring) than for standard NFS. The operating system and hardware dependent nature of HA-NFS makes it quite inflexible and thus not so interesting for this work.

Bibliography

1. M. Ahamad, M. H. Ammar: Performance Characterization of Quorum-Consensus Algorithms for Replicated Data. *IEEE Transactions on Software Engineering* 15(4): 492-496, 1989.
2. A. Bhide, A. N. Elnozahy, S. P. Morgan: Implicit Replication in a Network File Server. *Proceedings of the Workshop on Management of Replicated Data, November 1990.*
3. J. J. Bloch, D. S. Daniels, A. Z. Spector: A Weighted Voting Algorithm for Replicated Directories. *Journal of the ACM* 34(4): 859-909, 1987.
4. A. D. Birrell, A. Hisgen, C. Jerian, T. Mann, G. Swart: The Echo Distributed File System. Digital SRC Research Report 111, 1993.
5. P. Braam: The Coda Distributed File System. *Linux Journal* No 50 1998, pages 46-51.
6. R. Chow, T. Johnson: *Distributed Operating Systems & Algorithms.* Addison Wesley, 1997.
7. A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, D. Terry: Epidemic Algorithms for Replicated Database Maintenance. *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing,* pages 1-12. ACM, 1987.
8. G. Dini, S. J. Mullender: A Replicated File System for Wide-Area Networks. University of Twente 1993.
9. A. Goel, C. Pu, G. J. Popek: View Consistency for Optimistic Replication. *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems.* IEEE, 1998.
10. S. Jajodia, D. Muthcler: Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database. *ACM Transactions on Database Systems,* 15(2): 230-280, 1990.
11. D. B. Johnson, L. Raab: A Tight Upper Bound on the Benefits of Replication and Consistency Control Protocols. *Proceedings of the 10th ACM SIGACT-SIGMOD-SIGART Symposium of Database Systems*, pages 75-81. ACM, 1991.
12. M. L. Kazar: Quorum Completion. ITC, Carnegie-Mellon University, 1988.
13. B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shirira, M. Williams: Replication in the Harp File System. *Proceedings of the Thirteenth SOSP, pages 226–238, October 1991.*

14. T. Mann, A. Hisgen, G. Swart: An Algorithm for Data Replication. Digital SRC Research Report 46, 1989.

15. J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, F. D. Smith: Andrew: A Distributed Personal Computing Environment. *Communications of the ACM*, 29(3): 184-201 1986.

16. B. Noble, B. Fleis, M. Kim: A Case for Fluid Replication. University of Michigan. Not released yet.

17. T. W. Page Jr., R. G. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, G. J. Popek: Perspectives on Optimistically Replicated, Peer-to-Peer Filing. *Software - Practice and Experience*, 28(1): 155-180, 1998.

18. J-F. Pâris, D. D. E. Long, A. Glockner: A Realistic Evaluation of Consistency Algorithms for Replicated Files. *Proceedings of the 21st Annual Conference on Simulation Symposium*, pages 121-130. ACM, 1988.

19. J-F. Pâris, P. K. Sloope: Dynamic Management of Highly Replicated Data. *IEEE 11th Symposium on Reliable Distributed Systems*, pages 20-27. IEEE, 1992.

20. J-F. Pâris: A Highly Available Replication Control Protocol Using Volatile Witnesses. *International Conference on Distributed Computing Systems*, pages 536-543. 1994.

21. K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, A. J. Demers: Flexible Update Propagation for Weakly Consistent Replication. *Proceedings of the sixteenth ACM Symposium on Operating System Principles*, pages 288-301. ACM, 1997.

22. G. J. Popek, R. G. Guy, T. W. Page Jr., J. S. Heidemann: Replication in Ficus distributed File Systems. *Proceedings of the Workshop on Management of Replicated Data*, pages 20-25. 1990.

23. H. S. Sandhu, S. Zhou: Cluster-based File Replication in Large Scale Distributed Systems. *ACM Performance Evaluation Review,* 20(1):91-102, 1992.

24. M. Satyanarayanan: A Survey of Distributed File Systems. CMU-CS-89-116, Carnegie Mellon University, 1989.

25. M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, D. C. Steere: Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4): 447-459, 1990.

26. M. Satyanarayanan: Distributed File Systems. *Distributed Systems*, 2nd edition. Ed. Sape Mullender. Addison Wesley, 1993.

27. M. Satyanarayanan: Mobile Information Access. *IEEE Personal Communications*, February 1996, pages 26-33.

28. A. Siegel, K. Birman, K. Marzullo: Deceit: A Flexible Distributed File System. Cornell University, 1989.

29. A. S. Tanenbaum: *Modern Operating Systems.* Prentice Hall International, 1992.

30. E. R. Zayas: AFS-3 Programmer's Reference (several parts). Transarc Corporation 1991.

31. RFC 1094: NFS: Network File System Protocol specification. Sun Microsystems Inc, 1989.

32. RFC 1813: NFS Version 3 Protocol Specification. B. Callaghan, B. Pawlowski, P. Staubach, 1995.