Literature Study

Noora Peura

23rd March 2000

Abstract

Read-write replication is a challenge accepted by many researchers. This study gives an overview of the most important work done in this field. The study is mainly oriented towards read-write replication of files and directories. The most common algorithms are described and some examples of replicating file systems are shown. There is also an introduction to file systems and to replication.

This study is a part of the degree project *Read-Write Replication in Arla*.

Contents

Chapter 1

Overview of this Study

1.1 Introduction

This study is a part of a degree project at KTH, Stockholm. The goal of the project is to find ways to implement read-write replication in the file system Arla. Arla is a free clone of the distributed file system AFS. AFS allows only replication of read-only files.

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 network grows the load on the file servers increases. There is also always the risk that a server fails and the data on that server becomes temporarily unavailable. A remedy for both load and server failure 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.

This study covers the most common algorithms for replication, including consistency models and update propagation methods. Some examples of distributed file systems using read-write replication are also studied. There is also a short introduction to file systems.

1.2 Plan

Section two of this paper is a short introduction to file systems. The third section goes through the reasons for replication and the problems with read-write replication. In section four the most important replication algorithms are described, including the topics optimistic and pessimistic replication, primary copy algorithms, voting algorithms, two-phase commit, view consistency, update propagation and replica placement. Section five accounts for some evaluating studies of replication algorithms. In the last section some replicating file systems are described.

1.3 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

Introduction to File Systems

2.1 Overview

The file system is a part of the operating system. Sometimes it is solidly built-in, while in other cases it is a separate unit and can be exchanged without changing the operating system. Many file systems are even available for several operating systems.

The file system is responsible for mapping a file name to a position on a physical disk. For this it keeps a table of file ID:s and their corresponding disk positions. The file system also stores attributes for files, like size, creation time, time of latest access and flags of different kinds.

The design of the file system often depends on the operating system. The simplest operating systems are used by one user at one site running one process at a time. For these the design of a file system is relatively straight-forward. Modern operating systems allow several concurrent processes to be run by several different users at the same computer. This puts higher demands on the file system for concurrency control and security. Users should not be able to freely read each other's files and the semantics for concurrent writes to a file must be clear.

Distributed file systems are even more advanced. In a distributed file system files are stored at one or several file servers on a group of machines and can be accessed from any workstation that is a part of the system. Users should be able to access a file without knowing which file server it resides on (location transparency).

Some file systems can operate on different levels, so that the high level file system uses a lower level file system underneath. The high level file system provides new features like security, replication etc and lets the low level file system deal with disk and memory access. Many distributed file systems rely on lower level file systems that are built into or shipped with the operating system.

More about file systems can be read in [26]. [22] is a survey of the current distributed file systems in 1989. Much has changed since then, but the survey is still interesting as a history lesson.

2.2 Basic File Operations

The most common file operations are *create, open, seek, read, write, append, close, delete* and *rename*. Here follows a description of each of them.

create The create call creates an empty file by allocating a position in the file table and setting the attributes of the file.

open To access a file it must be opened, which means that the file system gets disk position information and attributes for the file. Some file systems map the file to the memory and return a pointer to the memory segment when a file is opened. If the file system uses caching the file or file attributes might be cached when the file is opened.

seek Many systems use a position pointer to point out which part of the file is currently read or written. Seek is used to move the position pointer.

read A read call returns a specified amount of data from a specified position (often the current position) in an open file. Some file systems that use caching copy the file or parts of the file to the cache when the file is read.

write/append Write and append modify the contents of the file. Write adds data to the current position, while append adds data to the end of the file. File systems that use a cache do not always commit the data to disk when it's written. Instead the data is flushed to disk periodically or when the file is closed. If a write-through cache is used the data is written to disk at one.

close Close is used to remove the file attributes (and the file in some cases) from volatile memory when the current file access is done. This is done for example when an application that writes to a file is closed. If the file system uses caching all changes are flushed to disk.

delete Delete removes the file and its attributes from the file table. After this the file can no longer be referenced. The disk space taken up by the file can then be overwritten by other files.

rename Rename is used to change the name of a file.

Many file systems provide several other operations, like commands for getting the attributes of a file or a flush command to commit cached changes to disk.

2.3 Directories

Most file systems organize the files into directories. Directories are special objects which can contain files or other directories, creating a directory tree. Directories can be created, deleted, opened, closed, read and renamed just like files. In addition, files can be linked and unlinked to the directories. In some systems linking and unlinking is automatic and a file can only be linked into (contained by) 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.

Some file systems use *mounting* 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 start point in the tree is called the root and the first disk partition to be mounted is often called the root partition. Some file systems have a whole set of 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.

2.4 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 in 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. Also there are several 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.

2.5 Security

In multi-user file systems security is often achieved by giving each user a user name and associating all files with their owner's 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.

Chapter 3

Replication in General

3.1 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.

3.2 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 it's 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 4

Replication Algorithms

4.1 Introduction and Plan of the Section

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 stored 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.

The notion of automatic - fluid - replication will also be discussed in this section. Automatic creation and migration of replicas means better performance and less work for the system administrators. Replicas are automatically created where they are most needed and deleted when they aren't accessed anymore.

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

4.2 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 side 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.

4.3 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 section 6, where some optimistically replicated file systems are described.

4.4 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. 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 section 6.

4.5 Voting Algorithms

Voting algorithms are based on the requirement that a certain amount 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 still allowing operation 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. Voting can also be used in a primary copy system to elect the primary copy.

4.5.1 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 is 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 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. This is often used for election of the master replica in a primary copy protocol. It can also be used directly as a replication protocol, but is not very optimal if there are more than three replicas of a file.

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 in systems where reads are frequent and writes rare. It is not efficient if writes and node or 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 isn't suitable for networks where network partitions occur, since that can lead to inconsistencies. Optimistic available copy protocols allow writes even if the network is partitioned.

4.5.2 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.

4.5.3 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.

4.5.4 Witnesses

To improve availability *witnesses* can be added to the set of replicating nodes. Witnesses store only update information, like a version vector. They do not store the data itself. Witnesses are allowed to participate in voting to some extent and can sometimes be promoted to "normal" nodes when a "normal" node becomes unavailable. Witnesses increase availability by increasing the number of nodes without requiring more storage. There has been a lot of work done regarding the use of witnesses, but for brevity only one example is described here.

In [18] Paris describes an available copy protocol with volatile witnesses. Volatile witnesses reside only in volatile memory, which makes them faster. They will loose all information at a server failure. In Paris 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 it's 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.

4.5.5 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 it's 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.

4.5.6 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.

Paris and Sloope suggested in [17] 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 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 fist 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.

4.6 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 it's 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.

4.7 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. It is also possible that a user gets data older than they have previously accessed, in the case of unfortunate network or server failures. View consistency guarantees that you only access the same version or later of data you have previously accessed. Still, availability is better than in pessimistic systems, since different users can access different versions of the data and even make conflicting updates, depending on their views.

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 6.

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.

Entities can be persistent (view-entries are stored in stable storage) or transient. They can also be distributed. The entity stores a database of view-entries, containing information about the current 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. Databases for transient entities are removed when the entity terminates.

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 publishing 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.

4.8 Update Propagation

4.8.1 Introduction

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.

4.8.2 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 the protocol. This gives some form of voting protocol where the client is the coordinator. The client can also send the update to only one node, which returns the call when it has sent the update to enough of the other nodes. 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. It will also 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.

4.8.3 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. 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 mean 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.

4.8.4 Epidemic Update Propagation

Epidemic algorithms rely on mathematical models which guarantee that all nodes will eventually get the update. [7] is a study of two common epidemic algorithms, *anti-entropy* and *rumor mongering*.

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 effective 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 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.

4.9 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 [14] 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-canceling 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 it's 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 [14] only a prototype has been implemented. Tests and simulations still need to be done. Still, the concept is worth notice.

Chapter 5

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.

5.1 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 better or equal availability than any other mutual exclusion protocol.

The third protocol considered is the majority voting protocol. It is used as an example of a simple protocol that provides mutual exclusion. The majority voting protocol is not very 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.

5.2 Comparison

Comparison of protocols by simulation has been done (among others) by Paris, Long and Glockner in [16]. They simulated a network environment with eight nodes. For eleven different configuration 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.

5.3 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 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 optimised. For read one/write all they find mathematically a limit for the optimal number of replicas. If majority 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 voting protocol increases availability. Depending on the mix of read and write transactions different values for read and write quora are optimal. A majority voting 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.

Chapter 6

Case Studies

6.1 Introduction and Plan of the Section

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 section 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.

6.2 NFS

Sun's Network File System Protocol (NFS), described in [27] and [28], 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.

6.3 Ficus

6.3.1 Overview

Ficus ([15], [20]) 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.

6.3.2 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.

6.3.3 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 [15] 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 it's 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 [15]. 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.

6.3.4 Evaluation

In tests with the Modified Andrew Benchmark (MAB) operations in Ficus show a medium time overhead of about 10-25 % compared do 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.

6.4 Coda

6.4.1 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 [24]. An older but more detailed description can be found in [23]. More information can also be found on the WWW at Coda's web site [www.coda.org] and in [5].

6.4.2 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.

6.4.3 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.

6.4.4 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.

6.5 Echo

6.5.1 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.

6.5.2 Replication

The original replication algorithm in Echo is described in [13], 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 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.

6.5.3 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.

6.5.4 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.

6.6 Harp

6.6.1 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 [12].

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.

6.6.2 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 acks 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 it's 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 a backup 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.

6.6.3 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 i 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 al well as to the backups.

6.6.4 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.

6.7 Deceit

6.7.1 Overview

Deceit [25] 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.

6.7.2 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 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.

6.7.3 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 (v1 v2). 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.

6.8 Huygens

6.8.1 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.

6.8.2 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.

6.8.3 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. Depending on 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.

6.9 Bayou

6.9.1 Overview

The Bayou storage system [19] 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.

6.9.2 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 it's 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.

6.9.3 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.

6.10 Frolic

6.10.1 Overview

Frolic [21] 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.

6.10.2 Replication

When a client requests a file that has it’s 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.

6.10.3 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.

6.11 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 it's 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] 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.*

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

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

[15] 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.

[16] J-F. Paris, 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.

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

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

[19] 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.

[20] 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.

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

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

[23] 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.

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

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

[26] A. S. Tanenbaum: *Modern Operating Systems.* Prentice Hall International, 1992.

[27] RFC 1094: NFS: Network File System Protocol specification. Sun Microsystems Inc, 1989.

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