UPPSALA
UNIVERSITET

IT 13 031
Examensarbete 15 hp
Maj 2013

Designing a distributed peer-to-peer file system

Fredrik Lindroth

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Designing a distributed peer-to-peer file system

Fredrik Lindroth

Currently, most companies and institutions are relying on dedicated file servers in order to provide both shared and personal files to employees. Meanwhile, a lot of desktop machines have a lot of unused hard drive space, especially if most files are stored on these servers. This report tries to create a file system which can be deployed in an existing infrastructure, and is completely managed and replicated on machines which normally would hold nothing more than an operating system and a few personal files.

This report discusses distributed file systems, files, and directories, within the context of a UNIX-based local area network (LAN), and how file operations, such as opening, reading, writing, and locking can be performed on these distributed objects.

Handledare: Justin Pearson
Ämnesgranskare: Karl Marklund
Examinator: Olle Gällmo
IT 13 031

Contents

Chapter 1

Introduction

Currently, most companies and institutions are relying on dedicated file servers in order to provide both shared and personal files to employees. Meanwhile, a lot of desktop machines have a lot of unused hard drive space, especially if most files are stored on these servers. This report tries to create a file system which can be deployed in an existing infrastructure, and is completely managed and replicated on machines which normally would hold nothing more than an operating system and a few personal files.

File sharing networks, such as gnutella [8], have shown that it is possible to provide files to a vast network of connected machines. Is this possible for a network of a smaller size? There are a number of requirements that must be met for this solution to be completely relied on, such as data availability and system-wide redundancy and security.

When designing a multi-user file system, other issues come into play which do not need to be considered in designing a single-user file system. Such an issue is file locking [1, 16]. GNU/Linux [2] has traditionally solved this by making file locks advisory, as provided by the fcntl(2) [16] system call; software needs to check for file locks when relevant, the system will not deny access to a locked resource. There are two kinds of locks: Read and Write. These are not mutually exclusive; both types of locks may exist simultaneously on the same object.

Another multi-user specific issue is that of access permissions: how do we verify that a user may access a certain resource when there is no central location to verify access requests against? This, too, will have to be addressed.

Another issue is that of consistency. We have a set of machines which may be separated into two or more sets at any time as a result of any number of failures. How can any node be certain that it views the latest data for a requested resource?

1.1 Focus of the report

This report will focus on the *theoretical* considerations and deliberations of implementing such a file system under GNU/Linux and other UNIX-like systems and flavours, due to the author's familiarity of such implementations. Because of this limitation, this report will only consider advisory file locking, and the

user/group/other access scheme commonly found in such systems. It is certainly possible to consider mandatory locking and other access control schemes in future work, and it is encouraged.

Chapter 2

Existing approaches

Quite a few UNIX-based solutions for distributed file systems exist. The most traditional approach is Network File System, or NFS [9]. This file system has been used for exporting file systems from a centralised server to clients, but this does not provide redundancy.

NFS [10, 9] is designed in an hierarchical manner, meaning that there is a clear difference between clients and servers. NFS servers are designed to be as stateless as possible, meaning they do not keep track of their clients. This also causes file locking to be implemented as a separate service, as it is inherently stateful. Being stateless also requires the server to check permissions on a file for each read/write call.

Andrew FS (AFS) [11] is somewhat related to NFS, being that AFS file systems may be accessed through NFS. The difference, however, is that AFS is designed to make clients store files locally, and through this mechanism enable a much larger degree of scalability, since files do not need to be transferred each time they are accessed, as opposed to NFS. For the purpose of this report, AFS can be thought of as a caching version of NFS. Some redundancy is gained by the local storage of requested files.

For actually distributing the file system among the peers of a network, Co-operative File System, or CFS [14], comes a fair bit closer to what is required, but with the major drawback that it is read-only; only the user exporting the file system can add, change, or remove files. It uses the Chord DHT system (see section 4.1: Overview) for routing and communication, and DHash for storage of the data blocks themselves.

Distributed file systems have also been used extensively with clustering, and so there exists both GFS [18] and OCFS (Oracle Cluster File System) [4], which are aimed at providing files to a computing cluster and distributed databases. This relies on a network of master and slave servers, which will require certain macines to be powered on permanently.

GFS2 [18] is Red Hat's approach to a clustered file system. It is designed to work in a SAN (Storage Area Network) rather than a normal LAN. It is classless; all nodes perform an identical function. It does, however, require the Red Hat Cluster Suite, a set of software only available through Red Hat Enterprise Linux. The nodes on which GFS2 is running are also meant to be

Name	Read-Write	Class-less	Dynamic	Redundant	Lock managing
CFS	No	Yes	Yes	Yes	N/A
Freenet	N/A	Yes	Yes	Yes	N/A
GFS	Yes	No	No	Yes	Yes
Gnutella	N/A	Yes	Yes	Yes	N/A
NFS	Yes	No	No	No	Yes
AFS	Yes	No	No	Yes	Yes
OCFS2	Yes	No	No	Yes	Yes
Proposed soln.	Yes	Yes	Yes	Yes	Yes

Table 2.1: A table comparing file system capabilities. Both FreeNet and Gnutella do not work with traditional file system semantics, and will not provide directly writeable objects nor file locks. CFS is read-only, and so does not need lock management. The proposed solution aims to have all these capabilities.

in a server role, and the file system is designed to cater to those needs, as it assumes all nodes have equal access to the storage, something a LAN cannot provide. GFS2 uses a distributed lock manager, which is based on the VAX DLM API.

Gnutella [8] is a popular file sharing network which is completely decentralised. It relies on a number of leaf nodes, which are connected to so-called ultrapeers. Ordinary leaf nodes may promote themselves to ultrapeer status if they deem themselves fit. Due to its use of ultrapeers to handle routing, many nodes can be reached in a few hops. Ultrapeers are themselves able to share files, but their primary job is to route messages to and from other ultrapeers, and to their leaf nodes when they are the intended destination.

Freenet [5] is yet another solution, which uses a key-based routing scheme and focuses on anonymous routing and storage. It does not support file system primitives, and should therefore be considered a file sharing solution rather than a file system.

Chapter 3

Requirements

To achieve both goals of data redundancy and lack of permanent servers, any given node must be allowed to crash or leave the network without bringing it down, while still maintaining high availability for all files. The network should react to any such event and reach a state where such an event may reoccur without impeding availability; in other words, the network should continuously keep itself at a sufficient level of data redundancy.

As the goal is to store file metadata much in the same way a traditional, local file system would, the metadata structures should be able to store information about file ownership and access permissions. It is also desirable for the metadata to be extendable in order to fit other attributes, such as access control lists and metadata applicable to additional operating systems, such as Microsoft Windows.

Table 2.1 lists some relevant capabilities of existing distributed storage solutions. FreeNet and Gnutella are not file systems, but rather file sharing services, and therefore lack facilities that are required of file systems, such as file listing.

3.1 File operation requirements

For file operations, we need to be able to do the following:

- Create - create a new file and associated metadata, and distribute it through the network.
- Delete - removes a file from the network, updating metadata to reflect this.
- Open - open the file for reading and/or writing.
- Close - close the file handle and announce this to all parties concerned. This operation will also release all held locks on the file.
- Read - read bytes from the file.
- Write - write bytes to the file.

- Lock - Obtain a lock on the file. Both read and write locks may be requested. Locks are advisory to software and will not deny access.
- Unlock - Remove locks on the file.

These requirements reflect what is possible to achieve on local file system.

3.2 Security requirements

3.2.1 Messages

These two properties must hold for a message routed from one node to another:

- Data integrity - The contents of the message must be proven to be correct in order to protect from tampering and/or failed transmissions.
- Data privacy - The contents of the message should not be readable by nodes along the path between the two nodes

Data integrity can be achieved by signing messages with a private key, while data privacy can be achieved by encrypting them with a public key. The only information that needs to be left in the clear are the source and destination node identifiers, as these are of vital importance for being able to route a message forward, and to provide a way for intermediate nodes to return a response to the source node in case of failure.

The use of public key encryption also provides a mechanism for authentication upon joining the network, since public keys can be signed by a trusted party, and clients may be configured to discard messages not signed by that party, effectively denying entry for unauthorised clients.

3.2.2 Storage security

Unlike traditional file systems, a distributed file system of this kind will require storage of data originating from other nodes for replication purposes. This data should not be read by local users, much like the personnel of a bank should not be able to access the possessions that their customers are storing in the safe; personnel should be able to bring the box to a customer, but it is the customer who opens the box in private.

3.3 Availability requirements

As this file system is intended to replace a centralised server, the availability of the files is a high priority. However, there are a few things that cannot be sacrificed in order to satisfy this goal.

The CAP theorem [7], or Brewer's theorem, states.

> In a distributed computer system, only two of three of the following may hold true:

- Consistency: all nodes see the same data set
- Availability: the system is always available
- Partition tolerance

Availability is a high priority, but should not be considered critical. In a file system that is used for many critical files, consistency is far more important, as lack thereof may cause different processes to see different data, thus making corruption of the file collection possible. Therefore, the service should be able to avoid answering requests if answering them would cause an incorrect reply.

The event of a node departing should never put the network in such a state that a file becomes unavaiable. This requires a replication factor of at least two for each file chunk, with mechanisms in place to maintain that factor. If a node departs, its data should be replicated from another node holding that data in order to maintain the replication factor.

Routing should also not be affected; a departing node should not disrupt any communications between other nodes, even though the departing node used to route messages between the two nodes.

For these reasons, it is not unreasonable to sacrifice total availability for consistency and partition tolerance. The system should naturally strive to maintain availability, but not at the expense of consistency, which is mandatory, and partition tolerance, since partitioning is unavoidable given enough time.

Chapter 4

Proposed solution

This chapter will specify a design that attempts to implement the given requirements. It derives its routing mechanism from the one used by the Cooperative File System (CFS) [14], as well as the local caching properties of Andrew FS (AFS) [11], in order to achieve scalability.

4.1 Overview

For this solution, we will use the DHT (Distributed Hash Table) [6, 15] design as a means of routing and replication management, as CFS has previously done using the Chord DHT implementation [14, 15]. Using a DHT network means that we do not have to consider the physical network topology, and the nodes may even be on different subnets or different networks altogether, enabling for file systems to exist across multiple office facilities. This solution recognises the ideas behind the Cooperative File System (CFS) [14] concerning routing and message passing.

A DHT design provides an overlay network and a means to route messages between nodes. It also allows for redundancy, and it is suitable for handling applications with large amounts of churn, which is the term for nodes joining and departing from the network [13].

This solution also builds on the scaling advantage that is provided by local caching, as shown by Andrew FS (AFS) [11].

4.2 DHT

As mentioned above, DHT is an overlay network that is used for creating a logical network topology on top of a physical one, for storing pairs of keys and values. This overlay network is used for passing messages between nodes, storing key-indexed values, and to facilitate the joining of new nodes. Since routing on this network is independent from the physical topology, routing can happen on any scale, from a LAN to a global network.

A DHT is laid out as a ring, called the DHT ring. This comes from the fact that DHT uses a circular keyspace; if n_1 is the node with the smallest numerical identifier (key) in the DHT ring, and n_n is the node with the greatest key, the

successor of n_n is n_1, since we have gone full circle in the keyspace. This enables a useful property for replication, which will be touched on in section 4.13: Replication.

4.2.1 Terminology

- *Key* - A numerical identifier for an object in the DHT ring. Usually derived from some property of the object, such as creation time, content or name.
- *Keyspace* - An interval $a \leq b$, in which a key o may be placed. In case of circular DHT systems, such as Chord and Pastry, this keyspace wraps around in a circular fashion. In this case, a may be greater than b, but still preceed b in the ring.
- *Leaf set* - In the Pastry DHT system, the leaf set L is the set of nodes with the $|L|/2$ numerically closest larger and $|L|/2$ numerically closest smaller node IDs, or node keys. These are used at the end of routing to hand the message over to the correct node.
- *Predecessor* - The predecessor of a node or object is the node immediately preceeding it in the keyspace of the DHT ring.
- *Root node* - For a given object O, its root node is the node, n, that has that object in its keyspace, which is $n_p < O \leq n$, for predecessor n_p.
- *Routing table* - A table containing entries which describe what path to take to reach a certain destination. In the Pastry DHT system, the routing table for a node is used to address node IDs with a shared prefix to the node in question.
- *Successor* - The successor of a node or object is the node immediately following it in the keyspace of the DHT ring.

The Pastry DHT system [6] uses a 128-bit, circular keyspace for nodes and objects. Every Pastry node keeps a routing table of $\lceil log_{2^b} N \rceil$ rows, with $2^b - 1$ entries each, where b is a configuration parameter specifying the number of bits in a digit, where digits are used to divide the keyspace, yielding a numbering system with base 2^b. This numbering system is then used in routing, as described in section 4.8: Routing.

The Pastry DHT system is a good choice for this file system since it is relatively simple to derive an implementation from. It is not really necessary to use the Pastry software library, but to use it as a model for creating a new DHT system in whichever language you please. Furthermore, much research into DHT-based systems use Pastry as a base, including research into replication [13].

4.3 Messaging

If any node wishes to communicate with another node, it does so by using message passing on the DHT ring. Each node in the ring has a responsibility to pass along messages if it is not the intended recipient.

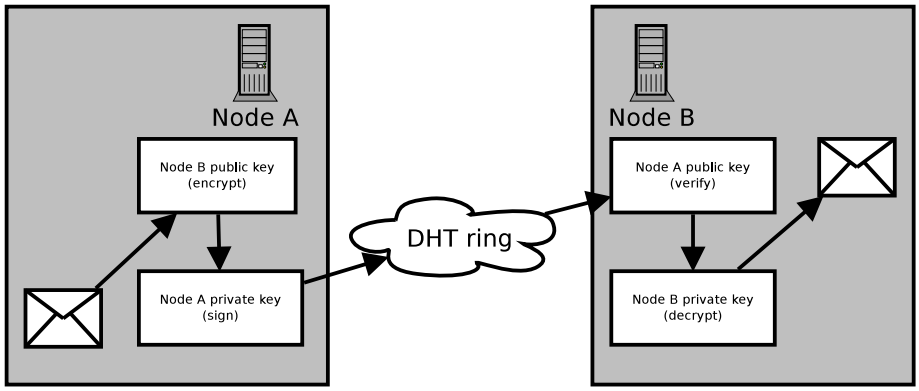

Figure 4.1: Using encryption and signing when passing messages

4.3.1 Message structure

Each message sent in the ring has a few mandatory attributes. These are as follows.

- *Source Node ID* - The identifier of the node which sent the message. Used by the receiving node when replying.
- *Sequence* - A sequence number set by the source node when sending the message. Used to differentiate between replies from other nodes.
- *Operation/Message type* - Used to distinguish between different kinds of messages.
- *Arguments* - Arguments to the message, and supplementary data to the operation defined by the message type.

This message is passed by means of nodes passing the messages along using routing. This is explained more in section 4.8: Routing.

4.4 Security

For security purposes, this design makes use of the GNU Privacy Guard (GPG) public key encryption scheme, both for verification purposes by use of signatures, and for data privacy through encryption. This scheme makes use of a public key and a secret private key. For our purposes, two keypairs will be used: one pair for nodes, and one pair for users. Figure 4.1 provides an overview of how encryption and signing is used during message passing.

4.5 The node key pair

The node key pair is used for sending and receiving messages to other nodes in the network. Any message that is to be sent from one node to another is

to be encrypted using the recipient's public key. The message is also signed with the private key of the sender. The recipient will then verify each message against the sender's public key in order to verify that the message has not been tampered with, and also that it originates from the alleged node.

4.6 The user key pair

When a file is created on the file system, it is encrypted using the public key of the user that created the file. Depending on permissions, the file may also be encrypted using the public keys of the users that belong to the group identified by the UNIX group ID (GID). This applies to directories as well, and affects mainly the read (r) permission, which will prevent the listing of directory contents.

The user key pair must be present on the user's current node, as files and messages intended solely for the user must be delivered to this node only.

4.7 Key management

The node key pair is generated and distributed to machines which are to become nodes in the network, along with a public root key. The public key of each node is signed by the root key, effectively granting the node access to the network, since nodes will look for this signature upon receipt of messages.

When a node is joining the network, it presents its public key as part of the join request. This means that any node routing the request has a way of verifying the authenticity and authorisation of the originating node, effectively blocking any intruders. Upon receipt of a message, the public key of the originating node or user is not guaranteed to be known. Such keys may be requested from any given node by sending a public key request message.

The Cooperative File System (CFS) [14] authenticates updates to the root block by checking that the new block is signed with the same key as the old one, providing an appropriate level of file security for a read-only file system.

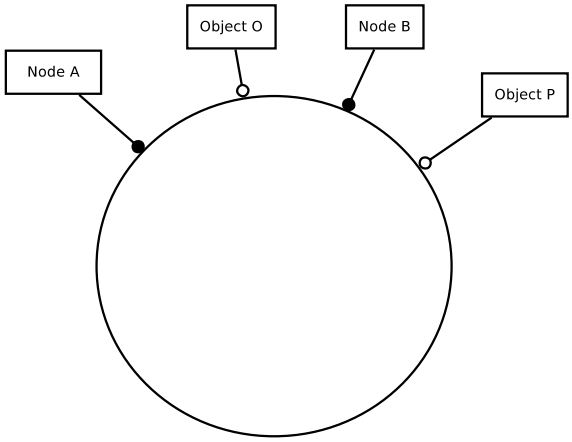

Figure 4.2: A DHT ring. Object O has Node B as its root node, while Object P has Node A as root.

4.8 Routing

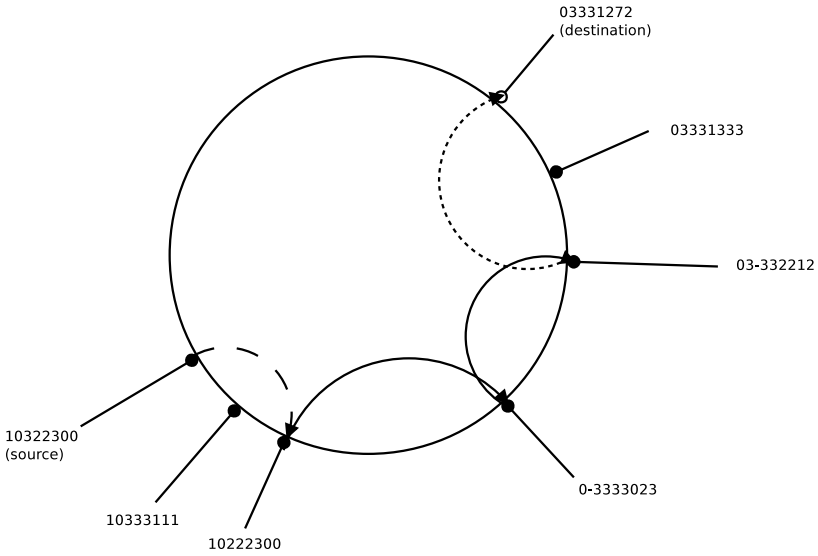

Figure 4.3: Routing example: The first route is to a node numerically closer to the destination. The two following is to nodes with at least one more digit in the prefix common to the destination. The last route is found in the leaf table.

All routing happens through the overlay network, which is gradually constructed as nodes join. Each node has its own numerical identifier which signifies its position in the DHT ring. Objects which are to be stored on the network are

assigned to the node which numerical identifier follows the numerical identifier of the object. For two nodes, a and b, and one object o, with $a < O \leq b$ holding, b is chosen as being responsible for storing o. [6]

The process of routing an object O from node n_s to destination key k is as follows:

Algorithm 4.1 Message routing

```
1:  procedure ROUTE(SourceNode, LastNode, Dest, Seq, Op, Args)
2:      REPLY(LastNode, received, Sequence)
3:      Result ← LEAFTABLEROUTE(SourceNode, LastNode, Dest, Seq, Op, Args)
4:      while Result = timeout do
5:          Result ← LEAFTABLEROUTE(SourceNode, LastNode, Dest, Seq, Op, Args)
6:      end while
7:      if Result = delivered then
8:          return delivered
9:      end if
10:     Result ← PREFIXROUTE(SourceNode, LastNode, Dest, Seq, Op, Args)
11:     while Result = timeout do
12:         Result ← PREFIXROUTE(SourceNode, LastNode, Dest, Seq, Op, Args)
13:     end while
14:     if Result = delivered then
15:         return delivered
16:     end if
17: end procedure
```

Algorithm 4.1: Message routing works like this:

1. Before any routing occurs, the last node in the chain, *LastNode*, is notified about the successful receipt of the message (line 2).

2. The local leaf table is searched for two nodes, a and b, such that $a < d \leq b$ for destination d. If such a nodes is found, the message is delivered to b and the procedure ends (lines 3 to 9). This procedure is detailed in algorithm 4.2.

3. If no such node is found, the routing table is searched through for a node which shares a common prefix with *Dest* by at least one more digit than the current node. If such a node is found, the message is forwarded there, and the procedure ends. If no such node is found, the message is forwarded to a node that is numerically closer to *Dest* than the current node (lines 10 to 16).

The Leaf table routing procedure (algorithm 4.2) passes the message on to a suitable leaf node. If no reply is received within the set time, the leaf node is deemed as failed, and the procedure will remove the leaf node from the leaf table, and return *timeout* to the *Route* procedure in order to let it know that it should try again.

Algorithm 4.2 Leaf table routing

```
1: procedure LeafTableRoute(SourceNode, LastNode, Dest, Seq, Op, Args)
2:     PreviousNode ← null
3:     for all LeafNode ← LeafTable do
4:         if PreviousNode = null then
5:             PreviousNode ← LeafNode
6:         else if PreviousNode < Dest ≤ LeafNode then
7:             Reply ← Deliver(SourceNode, LeafNode, Seq, Op, Args)
8:             if Reply = false then
9:                 RemoveFailedLeafNode(LeafTable, LeafNode)
10:                return timeout
11:            else
12:                return delivered
13:            end if
14:        else
15:            PreviousNode ← LeafNode
16:        end if
17:    end for
18:    return notfound
19: end procedure
```

Algorithm 4.3 Prefix routing

```
1: procedure PrefixRoute(SourceNode, LastNode, Dest, Seq, Op, Args)
2:     PrefixClosestNode ← LocalNode
3:     LongestPrefixLength ← SharedPrefixLength(LocalNode, Dest)
4:     NumericallyClosestNode ← LocalNode
5:     for all Node ← RoutingTable do
6:         NodePrefixLength ← SharedPrefixLength(Node, Dest)
7:         if NodePrefixLength > LongestPrefixLength then
8:             LongestPrefixLength ← NodePrefixLength
9:             PrefixClosestNode ← Node
10:        else if NodePrefixLength = LongestPrefixLength&|Node −
   Dest| < |NumericallyClosestNode − Dest| then
11:            NumericallyClosestNode ← Node
12:        end if
13:        Reply ← Forward(SourceNode, Node, Dest, Seq, Op, Args)
14:        if Reply = false then
15:            RemoveFailedNode(RoutingTable, Node)
16:            return timeout
17:        else
18:            return delivered
19:        end if
20:    end for
21:    if PrefixClosestNode = LocalNode then
22:        Reply ← Forward(SourceNode, NumericallyClosestNode, Dest, Seq, Op, Args)
23:        if Reply = false then
24:            RemoveFailedNode(RoutingTable, Node)
25:            return timeout
26:        else
27:            return delivered
28:        end if
29:    end if
30: end procedure
```

Algorithm *4.3: Prefix routing* attempts to find a node that shares at least one more prefix digit with the destination, *Dest*. It continually records the node that is numerically closest to *Dest*, $NumericallyClosestNode$, so in the case where a node with longer common prefix is not found, the message can be routed there instead. This procedure, like the leaf table routing prodedure, will remove a node if it does not respond, and will return a "timeout" status if that happens, letting the *Route* procedure know that it should try again.

Due to the circular nature of the DHT ring, finding a closer node will always succeed, since it is possible to walk around the ring towards the destination.

4.9 Joining and departing

This mechanism is used for all messages sent over the DHT ring. Joining the network simply means sending a join message with the ID of the joining node as the destination. Given two successive nodes n_a and n_b and a joining node n_j, which ID is in the keyspace of n_b, n_b will eventually receive a join message, and will place n_j as its predecessor. It will then pass the join message along to its current predecessor, n_a, which has n_b in its leaf set, and therefore knows to add n_j as its successor.

An example of object ownership with two objects and two nodes can be seen in figure 4.2.

In order to join the network, a node must know of at least one previously existing node. This node may be retrieved from a list of frequently-active nodes, such as the company web server, or it may be located by scanning the network for appropriate resources, which requires only a network connection.

4.10 Controlled departure

When a node wishes to depart, it sends a departure message to all of its leaf nodes.

Algorithm 4.4 Node departure: departing node

```
1: procedure DEPART(LeafNodes, StoredObjects)
2:     for all Object ← StoredObjects do
3:         Reply ← SENDMESSAGE(Successor, replicate_request, Object)
4:     end for
5:     for all Node ← LeafNodes do
6:         Reply ← SENDMESSAGE(Node, self_depart, "")
7:     end for
8: end procedure
```

First, the departing node asks its successor to replicate all of its data, since its position in the ring relative to the departing node will make it responsible for those objects once the departing node departs. Then, the departing node

sends *self_depart* messages to its leaf nodes in order to make them remove it from their leaf tables.

Algorithm 4.5 Node departure handler

```
1: procedure DepartHandler(SourceNode, LeafNodes)
2:     for all Node ← LeafNodes do
3:         if Node.ID ≠ SourceNode then
4:             NewLeafNodes[Node.ID] ← Node
5:         end if
6:     end for
7:     LeafNodes ← NewLeafNodes
8: end procedure
```

Upon receipt of the *self_depart* message from one of its leaf nodes, say n_d, the receiving node, say n_r, will then remove that node from its leaf set. If n_d is the predecessor of n_r, then n_r would have already received replication requests from n_d regarding all of its stored objects. As such, n_r is now ready to serve requests for objects previously on n_d.

4.11 Node failure

There is also a possibility that a node will stop responding to requests. In this case, the unexpected departure will eventually be detected by other nodes during an attempt to route a message through this node, since each node expects a *received* message back within a given timeout period as a confirmation of receipt in routing. See section 4.8: Routing. When a node detecs this, the non-responsive node will be removed from its leaf table or routing table, whichever is applicable.

4.12 Local storage

Each node will store file data and metadata for files. As with local file systems, files are stored in a hierarchical manner, with directories being a collection of files, and the root directory containing multiple files and directories.

After a file has been requested and downloaded to the local drive, it remains there, effectively creating a transparent cache, as with Andrew FS (AFS) [11]. Since all files in the network are stored on the participating nodes, this solution scales very well, as the total storage space increases with each joining node. However, space must be put aside to maintain redunandcy, and to allow nodes to depart without losing data. This is discussed in section 4.13: Replication

CFS stores and replicates inode and data blocks separately onto nodes [14].

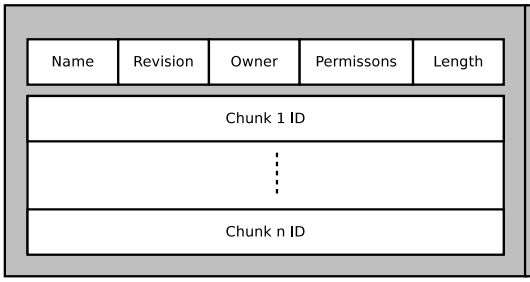

Figure 4.4: File data structure.

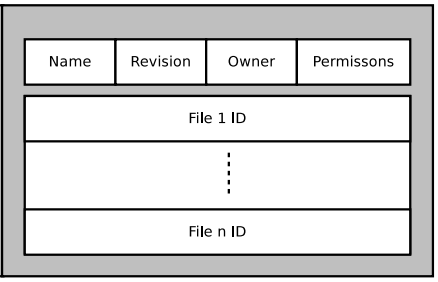

Figure 4.5: Directory data structure.

4.13 Replication

Replication is without a doubt the most important part of this file system, since it would achieve neither redundancy nor decentralisation without it.

For each file in the file system, there is a set of nodes where each node contains a copy of the file. Such a set is called the replication set for that file. In order to maintain a sufficient level of replication, the replication set needs to be sufficiently large. If a node in a replication set departs, a new node will be chosen and included in that set.

The replication factor is an integer value chosen to be larger than or equal to two, and may be adjusted depending on the amount of churn, nodes joining and departing, the network is expected to handle. A higher factor means more network overhead, since data needs to be replicated to a larger number of nodes. A lower factor means less overhead, but less tolerance for churn (nodes arriving and departing from the network). Situations which may call for a higher replication factor is the departure of nodes at the end of a working day. The replication factor may thus be increased in anticipation of such an event.

For any given replication factor rf and replication set rs with regards to file f, these are actions taken by the network to ensure that $|rs| = rf$.

- If $|rs| < rf$, find a new node candidate and add it to rs, ask it to replicate f.
- If $|rs| > rf$, remove a node from rs after instructing it to remove f from local storage.

The Cooperative File System (CFS) [14] handles replication by copying blocks onto the k succeeding nodes following the successor of the node containing the original object. The successor is responsible for making sure that the data is replicated onto the k succeeding nodes.

With regard to storage space efficiency, this scheme could be improved upon by allowing for other nodes than the successors to be used. Kim and Chan-Tin write in their paper [13] about different allocation schemes which could prioritise certain nodes based on available storage space. One could replicate smaller files unto nodes where larger files would not fit.

Choosing candidate nodes may be done using four main allocation criteria, according to Kim and Chan-Tin [13], which vary in suitability for different scenarios [13].

- *random-fit* - replicate the object onto rf random nodes.
- *first-fit* - replicate the object onto the rf succeeding nodes (after the root node for f with sufficient storage space available.
- *best-fit* - replicate the object onto the rf nodes which have the smallest adequate storage space among the set of nodes.
- *worst-fit* - replicate the object onto the rf nodes which have the largest remaining storage space among the set of nodes.

In theory, each individual node may use its own method of allocation, and additional criteria, such as file size, may be used to determine how replication sets are to be allocated. It would, however, be difficult to predict the allocation behaviour in such a network, and it may be desired that all nodes agree on the allocation scheme.

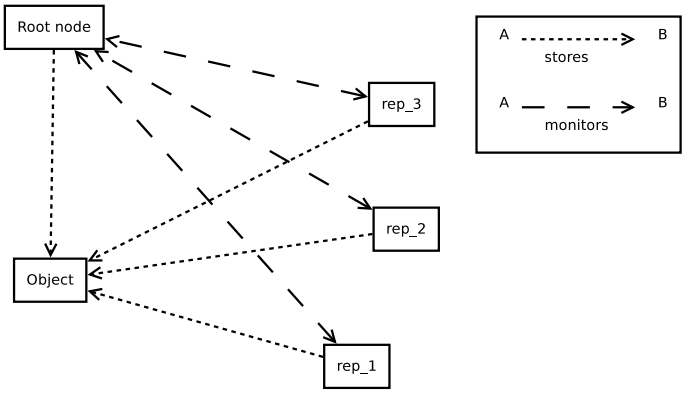

Figure 4.6: The replication relationships: The nodes in the replication set monitor the root node to take action in case it fails. The root node monitors the replication set in case one of the nodes fail, in which case it will choose a new candidate for the replication set.

Expanding on the CFS solution, each root node has a table over its associated objects and their replication sets. If a node is in a replication set for a file, it keeps an entry pointing towards the root node of the file. This allows for two-way monitoring, and efficient restoration in case of node failure (see fig. 4.6). This makes it possible for the root node itself to include a new node in the replication sets for its files upon receipt of a departure or crash message (for crash messages, see section 4.8: Routing, algorithms 4.2 and 4.3) .

In case of root node failure, the remaining nodes will copy their data to the successor of the defunct root node, which will then take over the duties from the old root node.

In order to find candidate nodes for file replication, the root node looks in its leaf set and routing table for nodes manifesting the required traits. If not found there, a message can passed along in the ring, selecting candidates along the way, while decrementing a counter once a candidate is found, reaching zero when a sufficient number of candidates have been selected.

Intuitively, the *first-fit* replication heuristic is the easiest to implement, since it will continuously jump from node to node by means of each node passing a probe message along to its successor, eventually ending up where it started, the root node. This message carries a counter as an argument, which is initially set to the number of candidates needed, and decreased each time a receiving node satisfies the storage requirement, that is, the object fits. This search is bounded by $O(N)$ for N nodes in the ring, in the case where the last candidate is the predecessor to the root node, which means the probe message would have gone the whole lap around the ring. When the file system is less populated in the beginning, this search would be more likely to finish early, near the lower bound of $\Omega(R)$, where R is the amount of candidates sought.

As for the *best-fit* and *worst-fit* heuristics, a query limit may be imposed in order to limit the amount of messages that are sent while querying for the nodes that best fit those criteria. Otherwise, the entire ring will have to be visisted in order to find the most suitable nodes. If a limit, n, is imposed, the algorithm will stop once it is found n nodes that are physically able to accomodate the object, and chooses $rf - 1$ of them, to satisfy the replication factor rf, excluding the root node. The query will start at the successor of the root node, and continue for at least n nodes, and possibly for all the nodes in the ring, depending on where the suitable nodes are located. This solution will run in the same amount of time as *first-fit* with this solution.

Algorithm 4.6 is a message handler for circular space queries, which is used for the first-fit, best-fit, and worst-fit storage allocation schemes.

Algorithm 4.6 Handling storage space queries.

```
1: procedure HandleSpaceQuery(RemNodes,SpaceRequired,Asker,MessageID)
2:     if SpaceRequired ≤ LocalNode.SpaceAvailable then
3:         ReplyMessage(MessageID, Asker, space_available, LocalNode.SpaceAvailable)
4:         if LocalNode.Successor ≠ Asker then
5:             Forward(MessageID, LocalNode.Successor, space_query, RemNodes−
   1, SpaceRequired)
6:         end if
7:     else
8:         if LocalNode.Successor ≠ Asker then
9:             Forward(MessageID, LocalNode.Successor, space_query, RemNodes, SpaceRequired)
10:        end if
11:    end if
12: end procedure
```

This procedure is executed by each node upon receipt of a space query

message. This message is sent to a node in order to find $RemNodes$ nodes among the receiving node and its successors, that are able to store an object of size $SpaceRequired$. For each receiving node, there are two possible cases:

- *case 1*: The object fits - The node sends a positive message to the querying node, informing that in can indeed store an object of the requested size, and of the remaining storage space in its local storage. The node then passes the message on to its successor, with a decremented $RemNodes$, if the successor is not the asking node (Lines 2 to 6).
- *case 2*: The object does not fit - In this case, the node passes the query message along to its successor, but with the $RemainingValidNodes$ remaining at its current value, if the successor is not the asking node (Lines 7 to 11).

4.13.1 Files

Files are responsible for storing the data itself, and in this file system they are comprised of the file data structure, and the data chunks.

The role of the file data structure is to contain all the metadata associated with the file, such as the file name, access permission and ownership information. This structure will also contain links to the file chunks which comprise the data itself, and there is also a revision number for handling updates, as well as a field that contains the total file size.

The file chunks are pieces of the complete files, identified by a hash of their content.

The file data structure, as seen in figure 4.4, is referenced by its UID (Unique identifier). The referrer can be any type of directory object, including the root node. Each file chunk has an unique identifier which is equal to the SHA1 hash of the chunk data. These identifiers are then stored in the chunk list of the file's data structure.

The identifier for the file structure is generated by concatenating the hashes of the file chunks, appending the full path of the file and the current UNIX time without separators, and then hashing the resulting string using SHA1. This identifier will be used to identify the file during its existence. Subsequent file updates do not change the file identifier, but merely increases the revision number. This is to avoid the recursive updates that would result from having to update the parent directory with the new file identifier, and having to move up the tree to perform the same updates, ending with the root directory. This solution makes it sufficient to just increment the revision number.

Andrew FS (AFS) [11] caches entire files at once. Working with chunks allows for opening large files, and only fetching the required data as seeks occur on the file handle.This allows for more selective caching, with reduced space usage in the case where many large files are opened, read for a few bytes, and closed.

4.13.2 Directories

In this design, a directory is not much different from a file. The main difference is that a directory is a collection of files rather than file chunks, as can be seen in figure 4.5. Directory UIDs are generated in the same way that file UIDs are, although without hashing the files contained, since there are none upon creation. Thus, only the full path and UNIX time will be used.

4.14 Creating files

When a file is written for the first time, the following steps are performed:

Algorithm 4.7 File creation and upload

```
1:  procedure FileCreate(FileName, FileData)
2:      ContainingDir ← StripFileName(FileName)
3:      CreationTime ← GetUNIXTime
4:      ChunkIterator ← 0
5:      UID ← ""
6:      file[size] ← 0
7:      for all ChunkData, ChunkSize ← Split(FileData) do
8:          ChunkHash ← SHA(ChunkData)
9:          UID ← UID + ChunkHash
10:         file[chunks][ChunkIterator] ← ChunkHash
11:         file[size] ← file[size] + ChunkSize
12:         StoreLocally(ChunkHash, ChunkData)
13:         SendMessage(ChunkHash, ChunkData)
14:         ChunkIterator ← ChunkIterator + 1
15:     end for
16:     UID ← SHA(UID + FileName + CreationTime)
17:     file[name] ← FileName
18:     file[ctime] ← CreationTime
19:     file[revision] ← 0
20:     StoreLocally(UID, file)
21:     SendMessage(UID, file)
22:     DirObjectUID ← GetObjectByPath(ContainingDir)
23:     DirObject ← SendMessage(DirObjectUID, get)
24:     if DirObject = false then
25:         abort
26:     end if
27:     DirObject[files][UID] ← file
28:     DirObject[revision] ← DirObject[revision] + 1
29:     SendMessage(DirObjectUID, put, DirObject)
30: end procedure
```

Line 1 extracts the directory name from the path. Line 2 gets the current

system time.

For each file chunk (line 6), the hash of that chunk is calculated (line 7). This hash is then appended to the UID of the created file (line 8), and the chunk is inserted into the file data structure, and the file size is updated accordingly (lines 9 and 10). The chunk is then stored locally, and is then sent as a message to the network.

The full file name and path is then appended to the UID, along with the UNIX time (line 15). The file name and creation time is then stored in the file metadata object (lines 16 and 17). The revision number is then set to 0, indicating a new object (line 19). This object is then stored locally, and then uploaded (lines 20 and 21).

As a last step, the file is added to the directory object, and the resulting directory object is uploaded again with its revision number increased to reflect the change (lines 26-28).

The receiving node is responsible for replicating the created object.

4.14.1 Analysis of algorithm 4.7

The amount of messages required to be sent are primarily dependent on the amount of chunks, which in turn is dependent on the chunk size and file size. For each chunk, there is one message sent to upload the chunk (line 13).

Except for the chunk uploads, the containing directory needs to be downloaded, updated and uploaded. This makes for another two messages, or four including replies. Furthermore, in order to download the containing directory, it is necessary to navigate the path from the root, through subdirectories, until we get to the containing directory. This work is done by the *GetObjectByPath* call on line 22, which will return the UID of the containing directory. This operation depends on the length of the path.

We thus have $NoChunks*(RepFactor-1)+4(RepFactor-1)+PathLength = (RepFactor-1)(4+NoChunks)+PathLength$ messages to be sent, since all file chunks need to be replicated. $(RepFactor-1)(4+NoChunks)$ is probably larger than $PathLength$, so the final estimate will be $(RepFactor-1)(4+NoChunks$ messages for the entire operation, when replication is taken into account.

Algorithm 4.8 Handling of file upload by root node

```
1: procedure UploadHandler(FileUID)
2:     RemainingReplications ← RepFactor − 1
3:     while RemainingReplications > 0 do
4:         RepSet ← GetRepCandidates(RemainingReplications)
5:         for all Node ← RepSet do
6:             ReplicationResult ← SendMessage(Node, replicate_request, FileUID)
7:             if ReplicationResult.status = success then
8:                 RemainingReplications ← RemainingReplications − 1
9:             end if
10:        end for
11:    end while
12: end procedure
```

This procedure is run by any receiving node that becomes the root node for a new object. It calls the *GetRepCandidates* procedure to get $RepFactor - 1$ candidates, where *RepFactor* is the replication factor for the network. It then continuously tries to replicate the object onto these nodes, until *RemainingReplications* reaches zero, which means that the wanted number of copies exist.

The *GetRepCandidates* procedure may differ from network to network, but its goal is to return a given numbers of possible candidates that satisfy the replication criteria outlined in section 4.13: Replication

On the receiving end of the message, the receiving node will handle the replication request.

Algorithm 4.9 Handling of replication request

```
1: procedure ReplicationHandler(SourceNode,Sequence,ReplicationObject)
2:     Object ← SendMessage(ReplicationObject, get)
3:     ReplyMessage(SourceNode, Sequence, replicated)
4: end procedure
```

4.14.2 Analysis of algorithms 4.8 and 4.9

The upload handler defined by algorithm 4.8 is tasked with finding replication candidates for the uploaded file. The theoretical upper bound for such a task is $O(N)$ messages, where N is the number of nodes in the network. For practical reasons, this is seldom the case, however. As described in section 4.13: Replication, the *GetRepCandidates* function is unpredictable when it comes to the actual amount of messages that need to be sent, but a minimum of $RepFactor - 1$ nodes need to be contacted to satisfy the replication factor. It depends on which method is used to find suitable canditate nodes.

This algorithm only sends two messages: one to ask the root node, identified by *SourceNode* for the object to be replicated. It then acknowledges the replication by replying to the *replicate_request* message with a *replicated* mes-

sage. Thus, the algorithm will only ever send two messages, which gives us an $O(2) = O(1)$ bound on messages sent.

The *GetRepCandidates* procedure may differ from network to network, but its goal is to return a given numbers of possible candidates that satisfy the replication criteria outlined in section 4.13: Replication.

4.15 Locking files

As described in the requirements, all locks in this file system are advisory. In other words, the locking system won't enforce locks, and it is up to the software to make sure that locks are considered. These locks are made to be compatible with the fcntl [16] system call in order to facilitate easy integration into UNIX-like operating systems. The locks are kept in a lock table for each node. In order to maintain a consistent lock state over the replication set for each file, the root node will distribute the lock changes to its replication set, maintaining the current lock state for the file should the node fail. In a traditional UNIX file system, locks are held on a per-process basis. To maintain this in a distributed file system, we must be able to identify an unique process, which can reside on any node, making sure that any process ID is unique. To achieve this, the node ID is concatenated with the process ID number, creating an unique identifier, which is passed along when requesting a lock.

The fcntl system call defines locks for byte ranges, but for now these locks are for the whole file. Any inquiry on the lock status of a byte range will return the lock status for the file.

There exists four modes in which a file may be locked: *unlk*,*rlk*,*wlk*,*rwlk*, which is unlocked, read locked, write locked, and read/write locked respectively.

In order for Process PID on node *SourceNode* to establish a lock on a file $FileUID$, the steps shown in algorithm *4.10: Lock request* will occur for the requesting node.

Algorithm 4.10 Lock request

1: $ProcessUID \leftarrow \text{SHA}(NodeID + PID)$
2: $reply \leftarrow$ SendMessage($FileUID, lock\_request, ProcessUID, lockmode$)
3: **if** $reply = ok$ **then**
4: **end if**

The corresponding algorithm for the receiving node is outlined in algorithm *4.11: The Lock Request Handler*.

4.15.1 Analysis of algorithms 4.10 and 4.11

Algorithm 4.10 is trivial, since it only generates a process UID and asks for the relevant lock, resulting in $O(1)$ messages.

The next algorithm, 4.11, on the other hand, is not trivial. It needs to propagate the lock changes to the replication set of the locked file, to facilitate a smooth failover in case the root node dies. Therefore it is upper-bounded by

Algorithm 4.11 The Lock Request Handler

```
1: procedure HandleLockRequest(SourceNode, FileUID, ProcessUID, LockMode, LockTable)
2:     ExistingLock ← LockTable[UID]
3:     if id ≠ null then                    ▷ There is currently a lock held on this file.
4:         if ExistingLock.ProcessUID = message.ProcessUID then
5:             if LockMode ≠ ExistingLock.LockMode then
6:                 LockTable[FileUID].LockMode ← LockMode
7:                 ReplyMessage(SourceNode, Sequence, lock_success)
8:                 for all Node ← GetRepSet(FileUID) do
9:                     SendMessage(Node, update_lock_table, FileUID)
10:                end for
11:            else
12:                ReplyMessage(SourceNode, Sequence, lock_exists)
13:            end if
14:        else
15:            ReplyMessage(SourceNode, MessageID, lock_denied)
16:        end if
17:    else                                 ▷ No existing lock exists, we are free to lock.
18:        LockTable[ProcessUID].LockMode ← LockMode
19:        ReplyMessage(SourceNode, MessageID, lock_success)
20:        for all Node ← GetRepSet(FileUID) do
21:            SendMessage(Node, update_lock_table, FileUID)
22:        end for
23:    end if
24: end procedure
```

the size of the replication set for the given file, which is often $RepFactor - 1$, since the root node is not counted amongst the replication set. Algorithm 4.11 will therefore send approximately $O(RepFactor - 1)$ messages.

4.16 Reading and seeking files

Reading and seeking files are implemented in the same way they are in a local file system. The only difference is that reaching the end of a chunk will prompt the download of the next one.

Algorithm 4.12 File read

```
1:  procedure READ(fd, length)
2:      LeftToRead ← length
3:      ReadBuffer ← ""
4:      while LeftToRead > 0&fd.filepos ≤ fd.length do
5:          if fd.chunkpos + 1 > fd.chunk.length then      ▷ Time to move to
    next chunk?
6:              fd.chunk ← fd.chunk.next
7:              fd.chunkpos ← 1
8:          end if
9:          ReadBuffer ← ReadBuffer + fd.currentbyte
10:         fd.chunkpos ← fd.chunkpos + 1
11:         fd.filepos ← fd.filepos + 1
12:     end while
13:     return ReadBuffer
14: end procedure
```

The fd structure represents the file descriptor. It contains everything you can read from the file data structure (Figure 4.4), as well as some state information, which is maintained locally on the node that has the file descriptor. This extra state information is as follows:

- *filepos* - The overall seek position in the file.
- *chunkpos* - The seek position within the current chunk.
- *currentbyte* - The byte at the current seek position.
- *chunk* - Information regarding the current chunk.

The *chunk* structure contains

- *next* - A pointer to the chunk that is immediately following this one.
- *length* - The length, in bytes, of the current chunk.

The major advantage of this approach compared to the whole-file-cache approach of Andrew FS [11] is that chunks are downloaded on-demand, potentially saving space when working with large files.

Forward seeking is equivalent to reading, except no data is returned.

4.16.1 Analysis of algorithm 4.12

Each read operation may result in $0..n$ messages, with associated replies, where n is the number of chunks in the file. A new chunk is downloaded if it does not exist locally, and the download is then triggered by reading through the end of a chunk, prompting the fetching of the next one.

If only a few bytes are read at a time, either zero or one download requests will be performed, but if more bytes than the chunk size are read, then it is even possible that two download requests are made as part of the read. In conclusion, it depends on which sort of file is being read, but up to one download request per read seems like a good guess for simple text file operations.

4.17 Deleting files

Deleting a file entails removing it from storage and from the containing directory structure. Upon receiving a delete request, the root node of the file will do the following:

Algorithm 4.13 Delete handler

```
1:  procedure DeleteHandler(FileUID)
2:      DirUID ← GetParent(FileUID)
3:      Dir ← SendMessage(DirUID, get)
4:      Dir.files[FileUID] ← null
5:      Dir.revision ← Dir.revision + 1
6:      SendMessage(DirUID, put, Dir)
7:      File ← SendMessage(FileUID, get)
8:      for all chunk ← File.chunks do
9:          SendMessage(chunk.id, delete)
10:     end for
11:     RepSet ← QueryRepSetForUID(FileUID)
12:     for all RepNode ← RepSet do
13:         SendMessage(RepNode, unreplicate, FileUID)
14:     end for
15: end procedure
```

First, the parent directory of the deleted object is established (line 2). This directory object is then fetched (line 3). The deleted object is then dereferenced from the directory object (line 4), and the revision counter of the directory object is increased to reflect the change (line 5), and the directory is then uploaded (line 6).

After this step, the chunks of the deleted file, and the file structure itself, are still stored on the network, even if they are not referenced to. To remedy this, the file chunks will have to be removed from their root nodes and replication sets, and thus the delete message will be sent for each of the chunks, and the unreplicate message will be sent to the replication set of the deleted file.

First, the file object is downloaded (line 7). Then, a delete message is sent to the root node of each chunk (lines 8 to 10). Following this, the replication set for the file data object is fetched from the replication table. $QueryRepSetForUID$ simply extracts all nodes that replicate the given object from this node. (line 11). Finally, these nodes are asked to stop replicating the file object (lines 12 to 14).

4.17.1 Analysis of algorithm 4.13

In order for the root node to delete a file, it needs to advise the file's replication set of the deletion. This applies to both the file data structure and the file chunks. Thus, it is safe to say that approximately $O(NoChunks * RepFactor)$, the number of chunks times the replication factor, messages are sent for file deletion.

Chapter 5

Differences from existing solutions

5.1 CFS

The most obvious comparison is to the Cooperative File System, or CFS [14]. The major difference is that the proposed solution is read-write, while CFS is read-only. Otherwise, the file system structure is strikingly similar. Directory blocks point to inode blocks (or file blocks), which in turn point to data blocks. Routing-wise, CFS uses the Chord [15] lookup service, which scales logarithmically in regards to the lookup cost over the number of nodes.

CFS uses DHash as a distributed storage, in addition to the routing service provided by Chord. The proposed design is using the DHT design for both routing and replication, requiring just the one overlay network, and messages to facilitate these tasks, making this a simpler approach.

5.2 AndrewFS

AndrewFS [11] caches whole files locally in order to access them, while the proposed solution only stores the necessary chunks, which may well be all of them in some cases. This provides better local storage efficiency.

While AndrewFS relies on workstations for caching, the primary storage still occurs on dedicated servers, while the proposed solution relies on the workstations for metadata, primary storage, and replication. This difference makes the proposed solution the only possible choice of the two for a true server-less environment.

5.3 Freenet

Freenet [5] is another solution which uses key-based routing [5]. Freenet emphasises on anonymous routing and storage, which are goals not required by this design. Routing-wise, Freenet uses its own protocol which uses TTL (time-to-live) on its messages, which means that it is possible that any given resource may never be accessed due to a large hop distance. This is a desireable compromise for large-scale networks, but it makes little sense otherwise. The circular nature of the DHT keyspace makes it possible to guarantee successful routing,

with a worst case scenario of $O(N)$ for N nodes, the cost of following each node's successor all the way around the keyspace.

5.4 OCFS2

OCFS [3] is a file system with focus on local storage and clustering. It is designed to avoid abstraction levels and to integrate with the Linux kernel. It does use replication, but it is not based on DHT, and it does not support a dynamic network structure, but rather relies on a configuration file to describe each file system and its participating nodes. Updating and distributing such a configuration file is impractical in a dynamic network setting. It also defines fixed points in the network, machines that needs to be always on for the file system to be accessible. OCFS2 also supports only 255 nodes in total [4]. OCFS2 also does not encrypt its local storage, which makes replication impossible with regards to file security. The proposed solution encrypts all objects, file data and metadata, using the public key of the creator, to be decrypted by the corresponding private key.

Chapter 6

Future extensions

6.1 Key management

This solution lacks a method of reliably storing and distributing the cryptographic keys associated with this file system. Some method of distributing private keys to users must be established in a secure manner for this solution to be effective security-wise.

6.2 File locking

The file locking scheme used in this design is thought to be sufficient for its intended purpose, but it is a naive approach which fails to take into account some distinct use cases, such as cases where mandatory locking is required. In order to expand the usability of this file system in other environments than UNIX-like systems, a new file locking scheme may be necessary to achieve local file system semantics on those systems.

6.3 Replication

Algorithms for choosing the replication set for a given file is heavily dependent on the type of data which will be contained within the file system, and developing an efficient algorithm for allocating replication sets is a challenge by itself. Without a doubt, several advances may be made in this area.

Chapter 7

Thoughs on implementation

While the work presented in this report is purely theoretical, an implementation is certainly possible, as the concepts have been established, and pseudocode has helped define the protocol.

7.1 Programming languages

Erlang [21]would be a suitable programming language to use for implementing most of this file system, since it has a high level of fault-tolerance built-in, and is effective at leveraging multi-threaded approaches to problems, such as spawning a new process for each incoming request.

In order to interface with the operating system, a mid-level language, such as C or C++ would be preferred, since most operating system kernels are written in such a language. The FUSE (File System in Userspace) system [20] is written in C, and is a common way to handle file systems in userspace.

7.2 Operating System interfaces

As mentioned under Programming languages, FUSE [20] is a common way to make file systems function as userspace processes. This provides great freedom with regards to size restraints, for instance, and allows for the init system to handle processes, since they are part of the normal userspace processes. A kernel module called fuse.ko is responsible for interfacing with the kernel.

7.3 Implementation structure

In an implementation using Erlang, C and FUSE, the code will be split into essentially two parts. One part will be written in Erlang and handle everything that has to do with DHT, local storage, locks, key managament and permissions. The other part will be written in C and use the FUSE libraries to provide callbacks to system calls such as open(), close(), read(), write(), fseek() etc., so that the operating system will know how to use the file system.

The C part will communicate with the Erlang part to provide file chunks, which can then be access by normal read() system calls on the FUSE side. A

call to write() will cause the C part to create and hash chunks, and tell the Erlang part to upload them to the network and handle the metadata.

These calls are only examples. A complete implementation will as such find a way to implement all system calls that a FUSE file system may be expected to handle, but even a partial implementation may be sufficient; it depends on the application.

Chapter 8

Conclusion

It is the author's belief that this report has sufficiently demonstrated the viability of a distributed, scalable file system intended for use in a heterogenous, class-less environment, using workstations as storage nodes. Certainly, there are improvements to be made, and a couple of those have been pointed out in chapter 6: Future Extensions. A real-world test would be the best proof of concept, and as chapter 7: Thoughts on implementation describes, a bilingual implementation combining the suitability of Erlang for higly-distributed and parallelised processes, with the interoperability and practicality of C, is a viable candidate. As with many things, the choice of languages is very much up to debate, as well as many other aspects of the file system design.

It is the author's hope that this report will inspire a debate that will question the role of the file server in a modern office or school environment.

Bibliography

[1] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne *Operating system concepts 7th ed.*, 2005, Section 10.1: File Concept, page 437, ISBN: 0-471-69466-5

[2] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne *Operating system concepts 7th ed.*, 2005, Section 21.1: The Linux System, page 737, ISBN: 0-471-69466-5

[3] Mark Fasheh *OCFS2: The Oracle Clustered File System, Version 2* [online], Oracle, http://oss.oracle.com/projects/ocfs2/dist/documentation/fasheh.pdf [Accessed 2013-02-27].

[4] Sunil Mushran *OCFS2: A Cluster File System For Linux - Users guide for release 1.6* [online], Oracle, https://oss.oracle.com/projects/ocfs2/dist/documentation/v1.6/ocfs2-1_6-usersguide.pdf [Accessed 2013-03-11].

[5] Clarke, Sandberg, Wiley, Hong *Freenet: A Distributed Anonymous Information Storage and Retrieval System* [online], École Polytechnique Fédérale de Lausanne, http://lsirwww.epfl.ch/courses/dis/2003ws/papers/clarke00freenet.pdf [Accessed 2013-03-11].

[6] Antony Rowstron, Peter Druschel *Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems* [online], IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany: 329–350, http://research.microsoft.com/ antr/PAST/pastry.pdf [Accessed 2013-03-07].

[7] Seth Gilbert, Nancy Lynch *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services* http://lpd.epfl.ch/sgilbert/pubs/BrewersConjecture-SigAct.pdf [PDF, Accessed 2013-02-28].

[8] T. Klingberg, R. Manfredi *Gnutella 0.6 RFC* [online], http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html [Accessed 2013-02-26].

[9] NFS working group *RFC 1094: Network File System Protocol Specification* [online], IETF, http://www.ietf.org/rfc/rfc1094.txt [Accessed 2013-03-07].

[10] George Colouris, Jean Dollimore, Tim Kindberg *Distributed Systems: Concepts and Design 3rd ed.*, 2003, Section 8.3: Sun Network File System, ISBN: 0-201-61918-0.

[11] George Colouris, Jean Dollimore, Tim Kindberg *Distributed Systems: Concepts and Design 3rd ed.*, 2003, Section 8.4: Andrew File System, ISBN: 0-201-61918-0.

[12] *GnuPG: GNU Privacy Guard* [online], GnuPG website, http://www.gnupg.org [Accessed 2013-02-26].

[13] Jinoh Kim, Eric Chan-Tin *Robust Object Replication in a DHT Ring*, 2007, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.125.7646 [Accessed 2013-02-27].

[14] Frank Dabek, M. fransk Kaashoek, David Karger, Robert Morris, Ion Stoica *Wide-area cooperative storage with CFS*, 2001, Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP) '01, http://pdos.csail.mit.edu/papers/cfs:sosp01/cfs_sosp.pdf [Accessed 2013-02-27].

[15] Stoica, Morris, Karger, Kaashoek, Balakrishnan *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications* [online], MIT CSAIL, http://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf [Accessed 2013-03-11].

[16] *fcntl manual page* [online], http://linux.die.net/man/2/fcntl [Accessed 2013-02-28].

[17] *RFC 1813* [online], IETF, http://www.ietf.org/rfc/rfc1813.txt [Accessed 2013-02-28].

[18] Steven Whitehouse *The GFS2 Filesystem* [online], Proceedings of the Linux Symposium 2007. Ottawa, Canada. pp. 253–259., http://kernel.org/doc/mirror/ols2007v2.pdf [Accessed 2013-03-11].

[19] George Coulouris, Jean Dollimore, Tim Kindberg *Distributed Systems: Concepts and Design 3rd ed.*, 2003, Section 7.3.2: Public-key (assymetric) algorithms, ISBN: 0-201-61918-0.

[20] *FUSE: File System in Userspace*, FUSE project page, http://fuse.sourceforge.net [Accessed 2013-03-07].

[21] *Erlang home page: http://www.erlang.org* [online] [Accessed 2013-03-11].