Keso - A scalable, reliable and secure read/write peer-to-peer file system

Mattias Amnefelt
mattiasa@kth.se

Johanna Svenningsson
mea@kth.se

25th May 2004

Abstract

In this thesis we present the design of Keso, a distributed and completely decentralized file system based on the peer-to-peer overlay network DKS. While designing Keso we have taken into account many of the problems that exist in today's distributed file systems.

Traditionally, distributed file systems have been built around dedicated file servers which often use expensive hardware to minimize the risk of breakdown and to handle the load. System administrators are required to monitor the load and disk usage of the file servers and to manually add clients and servers to the system.

Another drawback with centralized file systems are that a lot of storage space is unused on clients. Measurements we have taken on existing computer systems has shown that a large part of the storage capacity of workstations is unused. In the system we looked at there was three times as much storage space available on workstations than was stored in the distributed file system. We have also shown that much data stored in a production use distributed file system is redundant.

The main goals for the design of Keso has been that it should make use of spare resources, avoid storing unnecessarily redundant data, scale well, be self-organizing and be a secure file system suitable for a real world environment.

By basing Keso on peer-to-peer techniques it becomes highly scalable, fault tolerant and self-organizing. Keso is intended to run on ordinary workstations and can make use of the previously unused storage space. Keso also provides means for access control and data privacy despite being built on top of untrusted components. The file system utilizes the fact that a lot of data stored in traditional file systems is redundant by letting all files that contains a datablock with the same contents reference the same datablock in the file system. This is achieved while still maintaining access control and data privacy.

Abstract

I det här examensarbetet presenterar vi Keso, ett distribuerat och helt decentraliserat filsystem baserat på en peer-to-peer infrastruktur benämnt DKS. Under designen av Keso har vi försökt lösa flera av de problem som existerar i dagens distribuerade filsystem.

Traditionellt sett har decentraliserade filsystem byggt på idén om att ha centrala filservrar och för att klara belastningen och garantera tillräcklig tillgänglighet har ofta dyr specialhårdvara krävts. Systemadministratörerna förväntas vidare aktivt underhålla systemet för att balansera belastningen över olika maskiner, flytta data mellan olika hårddiskar och manuellt lägga till nya filservrar och klienter.

En annan nackdel med centraliserade system är att stora mängder lagringskapacitet slösas bort. Våra mätningar har visat att mycket data i ett produktionsfilsystem är redundant och att stora mängder diskutrymme på arbetsstationerna är outnyttjat. I det system vi undersökte fanns det mer än tre gånger så mycket ledigt diskutrymme på arbetsstationerna som det totalt sett fanns data sparat på filservrarna.

De huvudsakliga målen för Keso har varit att Keso ska tillvarata outnyttjade resurser, undvika att spara redundant data i onödan och att det ska vara ett skalbart, självorganiserande och säkert filsystem som är väl lämpat för verklig användning.

Genom att basera Keso på peer-to-peer-teknik har det blivit skalbart, feltolerant och självorganiserande. Tanken är att Keso ska köra på vanliga arbetsstationer och därigenom utnyttja tidigare outnyttjad lagringskapacitet. Keso tillhandahåller också metoder för att skydda privat data och tillhandahålla access-kontroll trots att det är byggt på opålitliga komponenter. Filsystemet utnyttjar faktumet att stora mängder data i ett traditionellt filsystem är redundant genom att låta filer som innehåller datablock med samma innehåll referera till samma fysiska datablock. Detta uppnås samtidigt som kryptering används för att uppnå säkerhet och accesskontroll.

Contents

1 Introduction **5**

1.1 Peer-to-peer environments 6

1.2 Making use of spare resources 6

1.3 Keso 7

1.4 Implementation 9

1.5 Achievements 9

1.6 Method 10

1.7 The report 10

1.8 Acknowledgments 11

2 Peer to peer **12**

2.1 Introduction 12

2.2 Structured peer-to-peer overlay networks 13

2.3 Distributed Hash Tables 13

2.4 Chord 16

2.5 Pastry 18

3 Distributed file systems **21**

3.1 Distributed file systems . . . 22

3.2 The Andrew File System . . . 25

3.3 CFS and Ivy . . . 29

3.4 Farsite . . . 31

4 The DKS Overlay Network **34**

4.1 Design objectives . . . 34

4.2 System overview . . . 35

4.3 Routing information . . . 36

4.4 Lookup algorithm . . . 36

4.5 Maintainance . . . 38

5 The Keso File System **40**

5.1 Design Objectives . . . 40

5.2 An overview of Keso . . . 46

5.3 Security and access control . . . 49

5.4 File versioning and conflict resolution . . . 53

5.5 Storing data . . . 54

5.6 Dealing with storage space . . . 57

6 Implementation **59**

6.1 DKS . . . 59

6.2 Localstore . . . 62

6.3 Keso 62

7 Statistics **65**

7.1 Description of the system 65

7.2 Free space 65

7.3 File sizes 66

7.4 Hashes of datablocks 67

7.5 Errors 68

8 Conclusions **69**

8.1 Main achievements 69

8.2 Evaluation 71

8.3 Future work 71

8.4 Conclusion 72

Chapter 1

Introduction

As computers become more and more important so does the need for environments where users can work together towards common goals in common projects. We have therefore chosen to investigate one of the essential building blocks for modern computer environments - highly scalable distributed file systems.

We believe that peer-to-peer techniques are a good foundation for these large, scalable computer systems and that research in peer-to-peer routing algorithms and application design will play an important part of tomorrow's computer systems. As a result of this we have looked into the existing routing infrastructures of peer-to-peer computing and implemented a research peer-to-peer overlay network named DKS that is being developed at the Royal Institute of Technology in Stockholm.

On top of DKS we have designed, and partly implemented, a peer-to-peer file system named Keso. This has included several challenges and we believe that the result is both interesting and encouraging.

The most important design criteria are that Keso should:

- make use of unutilized resources
- avoid unnecessary storing of redundant data
- scale well and support thousands of clients
- be self-organizing - as little human intervention as possible should be needed
- be a secure file system suited for a real-world environment

The security model of Keso is in large parts similar to that of Farsite[29]. The usage of a distributed hash table and the way files are stored is inspired by CFS[3] and Ivy[4].

1.1 Peer-to-peer environments

Peer-to-peer is a lot more than the file sharing applications that have been so debated due to their ability to let millions of users share copyrighted software illegaly. Peer-to-peer is a name that describes distributed systems in which participants share resources by direct interaction - all participants participate as equal peers. This type of design can be used for real read/write file systems, file sharing, messaging systems, databases and much more.

A peer-to-peer system is potentially much more resilient to system malfunction than centralized computer systems. The computers are distributed over a large physical area and data can be distributed and replicated over these computers. As a consequence of this a number of computers have to fail in order for the data to become unavailable.

Computer systems are becoming larger and larger - today systems with hundreds of thousands of users are not unheard of. Scalability has therefore become an important issue. Recent research has come up with new ways of efficiently locating other members in the network and finding the required information. By doing this, peer-to-peer has become an attractive solution to many of of the challenges of modern day computing.

1.2 Making use of spare resources

The last years' rapid development in the computer industry has led to that an ordinary workstation today is a quite powerful PC with excess computing resources, including storage space and processor time. In a large organization, hundreds or even thousands of gigabytes of storage space is unused on these workstations. In the IT-department at KTH 1.75 terabytes are unused out of the 3.5 terabytes existing disk space at the local workstations.

In a traditional centralized system, the components that make up the core system often are expensive. This is mostly because complicated hardware solutions are necessary to provide the required fault tolerance and performance. In peer-to-peer solutions ordinary hardware can be used to achieve the same performance and reliability, thus making it a cheap and attractive solution. If spare resources can be used, the hardware will even be "for free" - making it even more attractive.

1.3 Keso

The aim of Keso is to design a file system that is highly scalable, secure, reliable and makes use of spare resources on the participating workstations.

The main idea behind Keso is to divide the files and spread all the file blocks over the participating nodes. There will be no centralized parts and no dedicated components. Each data block is given a key and each participating computer (called a node) is responsible for all the keys in a given range.

Directories serve as a name/i-node lookup service and directories are also assigned keys. The node that is responsible for that key will act as a server for all the different types of directory operations that the file system needs to carry out.

In order for Keso to supply the necessary security, files are encrypted and data blocks are given a key that is generated from their contents (a SHA1 content hash). Directories are signed in a way that makes the client able to verify their correctness.

Keso is designed to be a versioning file system, which means that old versions of files are kept in the file system after then have been overwritten. The reason behind this is twofold - users are given a way to easily retrieve old versions of their files and the storing of old version also facilitates recovery from network partitioning and attacks from malicious users.

1.3.1 The DKS overlay network

The DKS (Distributed K-ary Search) overlay network is the foundation upon which the implemented parts of Keso are built. It serves as a routing infrastructure and provides two types of services - lookups for data or addresses and item insertion. It is designed to be very scalable and performs all operations in time logarithmic to the size of the network.

All nodes in the system are assigned identifiers which for example can be calculated from their IP-addresses. The nodes are ordered in a ring by their identifiers. Data items are assigned keys from the same identifier range. A node is responsible for storing and handling requests concerning data items with identifiers that are between the node's own identifier and the node with the closest preceding identifier. DKS also handles replication of data items.

1.3.2 Reliability

Keso is designed in order to achieve high reliability. Data is supposed to be replicated to a configurable number of nodes and if some of these fail, the data should automatically become available at the replication sites. Since it is completely decentralized there are no dedicated components whose failure would create more disturbance than any random node and Keso thus tolerates failure as long as no more than a configurable number of nodes fail at the same time.

Since it is not possible to ensure that all nodes are reliable, different kinds of cryptographic methods are supposed to be used in order to solve problems related to malicious nodes. The participating node that acts as a client also awaits acknowledgments in order to make sure that a sufficient number of replicas are kept.

1.3.3 Scalability and self-organization

In a centralized system a server or a server group is responsible for all or large parts of the file system. In Keso the granularity is at directory level and no operations require complicated agreement protocols for a larger group than the number of replica sites. These replica groups are different for different directories, in order to achieve scalability. Operations are only depending on the initial lookup cost, number of concurrent operations on a specific directory and the number of replica sites. This way the need for centralizing requests for larger parts of the file system is avoided.

In ordinary file systems the system administrator has to spend a lot of effort in balancing load and managing replication. The system administrator also has to quickly react when the system fails. The goal for Keso is that no operations needed to ensure that the system runs smoothly will need human intervention.

1.3.4 File versioning and reuse of storage

One of the most common reasons for data to be lost is human mistakes. Keso aims to avoid this by keeping old versions of files in the system until the storage space needs to be reclaimed.

Since data is stored and encrypted in a way that makes data blocks with the same contents end up at the same node, no storage space is wasted when files and file versions with the same contents in the data blocks are stored.

1.3.5 Security and access control

Traditional server-based systems are based on the assumption that the server can be trusted. In a peer-to-peer environment this is no longer true, since some of the participants might be compromised. Creating a secure file system on top of untrusted nodes thus provides new challenges and a need for a new way of thinking of security.

Keso solves these challenges by crypthographical means. Files are encrypted with a symmetric crypto and the key is encrypted with the public keys of the users who are supposed to be allowed to read the file. All metadata are signed and the block-lists contain content hashes of the blocks, making it easy for the client to verify data integrity.

1.4 Implementation

The version of the DKS system described in this document has been almost completely implemented, but only some parts of the Keso system has been implemented. Today it is possible to store files in the system and retrieve them again. Security, access control and replication is not implemented and the implementation also lacks kernel support.

1.5 Achievements

The main achievement of this thesis is that we show how a completely decentralized read/write file system with access control and data integrity can be built on top of an overlay network that provides a distributed hash table.

We have also shown that large amounts of diskspace are wasted in traditional computer environments with file servers and workstations. As much as 24 percent of the data stored on the file servers of an existing distributed file system is redundant without providing extra fault tolerance. A large amount of unused disk space is also available on workstations. Our results show how to utilize these resources by using a distributed hash table as a foundation.

1.6 Method

The project has been divided into four separate stages. We began by reading research papers related to this area. This included both papers relating to peer-to-peer structures in general and papers on distributed file systems with a focus on peer-to-peer file systems.

In the second stage we focused on gathering information about the needs of real-world environments. Here we made some statistic investigations of the file system in the IT department (IT-enheten) at KTH, a AFS file system with 5000 users and 400GB read/write data.

The third stage was the actual design of the Keso system. This included using the knowledge gained from the two previous stages.

After the design had been done we did the DKS and the partial Keso implementation.

The evaluation that has been done is mostly a theoretical analysis of whether Keso meets the design goals or not. When the Keso implementation is more complete a full performance analysis should be made as well.

1.7 The report

The report is divided into two major parts. The first part is mainly concerned with related work in the areas of peer-to-peer computing and distributed file systems. Each chapter is concluded with summaries of the most relevant work that we have encountered. In the peer-to-peer chapter this is Chord[1] and Pastry[2], two exciting research projects in peer-to-peer infrastructures. In the file system chapter we cover the Andrew file system[7], that is a widely used distributed file system, and CFS[3], Ivy[4] and Farsite[29], three recent peer-to-peer based approaches to making distributed file systems.

The second part describes our contribution, including both the DKS implementation and the design and partial implementation of Keso. Chapter 4 covers a description of the DKS overlay network, which we have implemented and used as a foundation for Keso.

Chapter 5, 5.2 and 5.2.2 are concerned with the design of Keso. In Chapter 5 we describe the motivation for Keso and our design goals. Chapter 5.2 is mainly concerned with giving an overview of the Keso system and the main design choices that has been made. Chapter 5.2.2 goes into a more in-depth explanation of some of the

more interesting parts.

The following chapter is about the implementation, describing the different modules and the API of Keso and DKS.

In Chapter 7 we present a summary of some statistics we have made on the file system at the IT department at KTH.

The last chapter summarizes the major results from this project.

1.8 Acknowledgments

First of all we would like to thank our supervisor, Luc Onana Alima. He has encouraged us to think and maintain a critical view of our own work. Ali Ghodsi has provided lots of valuable feedback and encouragement during the project. Rasmus Kaj has helped us with C++. We have also used a communication package written by him. Our examiner, Thomas Sjöland, has given us much support and encouragement. Per-Johan Svenningsson and Jimmy Magnusson have given a lot of useful comments on the report.

We would also like to thank all others who have been supporting us along the way. You know who you are.:)

Chapter 2

Peer to peer

2.1 Introduction

As more and more people and organizations use computers the demands on computer systems have changed. Today there are systems with hundreds of thousands of participants and scalability has therefore become an important issue in computer system design.

In this light the old server-based solution is no longer satisfactory - there is a limit of how many clients a single server can serve and if a centralized system becomes large enough replication and load-balancing will create too much overhead. Decentralizing control and letting the computers participate as equal peers will reduce this overhead in very large systems.

Peer-to-peer also has several other advantages. Today PC's are cheap and often more powerful than needed by the users and peer-to-peer techniques provide ways of accessing these unused resources. Decentralizing control and building in replication potentially makes peer-to-peer systems very fault tolerant.

Due to these reasons it is not surprising that peer-to-peer techniques are becoming more and more popular. This chapter intends to give an overview of the peer-to-peer problem setting and common techniques for achieving the design goals. It also includes case-studies of Chord [1] and Pastry [2], two peer-to-peer infrastructures built as distributed hash tables (DHT). Later chapters will show how a DHT is possible to use as a base for designing a highly scalable file system.

2.2 Structured peer-to-peer overlay networks

2.2.1 Peer-to-peer characteristics

The main characteristic of a peer-to-peer system is that it is fully decentralized. There are no servers and no central control and thus all nodes participate as equal peers.

Participating nodes can be diverse in several aspects; e.g. CPU, memory, disk-space and bandwidth. Nodes can also be distributed over a physically large area, making latency between some nodes very high. This requires a peer-to-peer system to deal with differences in both node configuration and connectivity.

Another characteristic is that all nodes cannot be trusted - node failure is a normal situation and malicious participants are likely. Nodes also join and leave the system arbitrarily, thus peer-to-peer infrastructures have to adapt to a changing environment.

2.2.2 Basic types of infrastructures

Peer-to-peer networks can be partitioned into two different classes according to their basic design. They can either be unstructured or structured overlay networks. In an unstructured overlay network, such as Gnutella [23] or Freenet [24] data is not guaranteed to be found even if it is available in the system. Unstructured systems also cannot give definitive negative answers. Another problem with unstructured overlay networks is that they do not scale very well. The probability of finding data decreases with the number of nodes in the system.

The lack of guarantees is fine for some types of applications, but for others it is necessary to get definitive answers and therefore more is required from the underlying infrastructure. Structured overlay networks is, as the name suggests built with an internal structure which is maintained by the algorithm. Research systems such as Chord[1] Pastry[2] and DKS[13], have shown that it is possible to construct infrastructures that find the requested data in a peer-to-peer network in time logarithmic to the number of nodes in the system as long as there are no failures.

2.3 Distributed Hash Tables

Several interesting peer-to-peer infrastructures are based on the idea of Distributed Hash Tables (DHT) [1] [2] [13]. A Distributed Hash Table, or DHT, can be used as a basic building stone for creating peer-to-peer applications for purposes such as

distributed file systems, data sharing and group communication applications.

In a DHT nodes and data to be stored are given identifiers in the same identifier space. Nodes are given responsibility for parts of the identifier space and for the data which is stored with identifiers in its part of the space. This means that in order to find some data you contact the node which is responsible for the identifier of the data. Since these identifiers are assigned randomly, load will be distributed uniformly over all participating nodes.

This key can be used to identify and retrieve the data. The hash table provides two operations, put(key, data) which inserts the data associated with the specified key into the system, and get(key) which retrieves the data.

2.3.1 Structure

Computers that are part of the distributed hash table are called nodes. Each node that participates in a distributed hash table is assigned a node identifier which is unique within the system. Both data and node identifiers are mapped into the same address space. Node identifiers are chosen to spread out randomly across the address space. This means that two nodes that are physically close do not have node identifiers that are numerically closer than any other two random nodes.

Nodes are defined as responsible for parts of the identifier space. How this is done varies with the kind of overlay network. In some systems this might be the node which numerically follows the data identifier while in others it might be the node which is numerically closest, either preceeding or succeeding.

2.3.2 Routing

In order to find a piece of data in the system, all that is required is a method to find out which node is responsible for that piece of data. Since one of the aims of a peer-to-peer system is to limit the amount of knowledge about the global state of the system at each participating node we cannot contact the responsible node directly. Instead we must go through other nodes to contact the responsible node.

Peer-to-peer system usually have different ways to come closer to the responsible node by at least half the distance in each step. This causes lookups to be made in time logarithmic to the number of nodes. One way to implement this is to let each node keep a set of pointers to nodes exponentially growing distance away. This is shown in figure 2.1, where pointers from one node to another is shown as arrows.

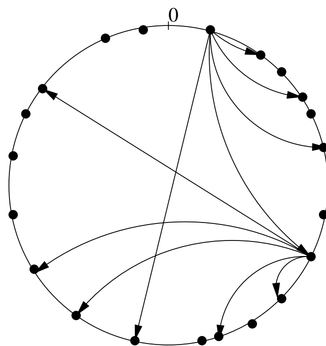

Figure 2.1: A typical DHT ring structure

2.3.3 Lookups

When an application wants to retrieve a certain piece of data it can construct a lookup request for the key associated with that piece of data. It will then forward the request to a node which it has knowledge about and which is as close as possible to the lookup key. This node will have more, or at least not less, knowledge about the vicinity of the key and will be able to send the request even closer. When the request reaches the responsible node, the answer is returned to the requestor.

2.3.4 Dealing with heterogeneity

There are basically two types of differences that a peer-to-peer infrastructure has to deal with - node configuration and connectivity - and they require quite different approaches.

Nodes in the system may differ in how much memory, storage space or processing power they have available for the distributed system. This difference in node configuration in a peer-to-peer system can easily be dealt with by introducing the concept of virtual servers [34]. For each virtual server a node is running, it acquires the responsibility for a small portion of the node identifier space. A resourceful node can thus be responsible for a larger total part of the node identifier space than a node with less resources.

Nodes may also differ in the connectivity they have with other nodes. One node may be located on a high speed network while others may be located behind modem links. Handling connectivity is a tricky issue. Some systems, like Pastry [2], take network locality into account when generating routing tables. This is described further down in section 2.5.

2.3.5 Tolerating failure

There are two components that can fail in a peer-to-peer network - communication channels and nodes. Failing communication can result in three different scenarios; the node becoming unavailable from all other nodes, the node becoming unavailable from some nodes and finally the network splitting into two or more separate partitions. Recovering from these types of failures both requires data to be replicated to nodes that are unlikely to lose network connectivity at the same time and routing-tables that provides means for recovery and alternate path discovery in the case of a communication failure.

Node failures can have different results depending on the node failure semantics. If a node fails by completely ceasing to reply (fail-stop) it is sufficient to detect the failure and have replication and algorithms for alternative path discovery. It is significantly more difficult to handle failed nodes that replies incorrectly to requests (Byzantine failures), especially if the node is malicious and tries to cause damage by purpose.

2.4 Chord

Chord [1] is a distributed hash-table which provides a way of looking up the node that is responsible for a given key. It is designed to be fault-tolerant and adapt to a changing environment. The system was developed by the Parallel & Distributed Operating Systems Group at Massachusetts Institute of Technology Lab for Computer Science. Several applications such as CFS [3] and Ivy [4] have been built on top of Chord.

The protocol is intended as a lower-level routing layer for applications that want to use a peer-to-peer infrastructure. These applications create a hash-table key by any mean suitable and can associate data with this key. Chord will then tell the application which of the nodes in the system that are supposed to store the data.

According to the authors Chord is very robust in a changing environment and scales very well in large systems. The amount of work needed to handle joins and departures are both on the order of $O(log^2\ n)$. Lookups also scale well and the number of messages exchanged are in the order of $O(log\ n)$.

2.4.1 Routing

Chord assigns node identifiers based on an SHA-1-hash of the nodes IPv4-address and organizes the nodes in a virtual circle based on the node identifiers. Both keys

and node identifiers map onto the same virtual ring and the first node that follows a certain key in the ring is responsible for that key.

Each node keeps track of its successor and predecessor on the ring. In addition to this, a data structure which is called a finger-list is maintained. Only the successor is needed for a correct search-result. However, the predecessor makes it possible to join and leave the network in an efficient manner. The finger-table is used to speed up operations by introducing "short-cuts" across the ring. This finger-list works pretty much like a skip-list. This means that each node keeps a number of pointers to nodes at an exponentially increasing distance away. These pointers make it possible to do lookups $O(log\ n)$. All of these pointers are periodically checked, and updated if needed.

2.4.2 Join

When a new node joins the network it asks an arbitrary node to look up its successor, updates predecessor and successor pointers and finger-table-entries and transfers the relevant keys from its successor. It then proceeds to send notifications to the nodes it thinks should have itself in their finger-lists. This is done by notifying all nodes that are predecessors of keys which are 2^i steps before the node that just joined. This may occasionally miss, but that is only relevant for performance and will be corrected on the next periodic update.

2.4.3 Failure

Failures are detected periodically by the Stabilize Protocol. In the event of a node failure the ring can be traversed via the successor pointers only. In order to keep the ring intact, each node keeps track of a number of its immediate successors. If a node is detected as failed it will be removed from the ring. The extra successor pointers are used for this. The fix-fingers routine is executed periodically to make sure that the finger-lists are correct and up to date.

If a node in the path to the node that holds a specific key is broken, the lookup will fail. This can be handled by retrying after a short while, since the stabilize protocol will heal the ring. A different approach is to ask the previous entry in the finger-list. This should give a new and hopefully valid path, even though performance would decrease. It is also possible to keep track of the predecessors to the entries in the finger list.

In the case of network partitioning, the Chord ring might split into smaller rings.

2.5 Pastry

Pastry [2] is a distributed hash table, which means that given a key it looks up the node that is responsible for that key. Pastry is fully decentralized, scalable, self-organizing and fault-tolerant and in contrast to other DHTs like Chord, Pastry has built in replication of data and takes network locality into account. It is developed in a collaboration between Microsoft Research in Cambridge, UK and Rice University, USA.

2.5.1 Identifier

Each computer that takes part in a Pastry network is called a node and it is configured to use a unique node identifier. This identifier is usually computed by some kind of function, for instance a hash of the nodes IP-address. Two nodes are said to share an n digits long prefix if the n first digits in their node identifiers are the same.

2.5.2 Node state

Every node keeps a routing table which on row i column j keeps a reference to a node that shares an i digits long prefix and whose $i+1$ digit is j. Pastry tries to keep nodes which are physically close in the routing table. In order to achieve this a neighbourhood set with physically close nodes is kept and this set is used for finding good routing information.

References are also kept to L neighbours in the node identifier space with $L/2$ neighbours with higher and $L/2$ with lower node identifiers. This is called the leaf set and these nodes are used to linearly forward a message in the final steps of the routing algorithm and to make sure that a path always can be found even in the case of node failures and lack of routing information. In many ways the leaf set fills the same functionality as the successor pointers in Chord. Pastry also keeps pointers to a set of nodes which are close with respect to the underlying network layer. This set is called the neighbourhood set.

2.5.3 Routing

The node with a node identifier that is closest to a given key is responsible for that key, but the key is also replicated to a certain number of the nodes in the leaf set. When a request for a specific key arrives at a node, that node forwards the request to a node that shares one digit more in the prefix with the given key. If there is no such

node in the routing table, the message is forwarded to a node that shares the same prefix but whose node identifier is numerically closer to the key. That way the request is forwarded closer and closer to the responsible node. Since Pastry routes requests to a node that is supposed to be close in the network layer, it is likely that requests first arrive at a replica which is closer to the requesting node than the responsible node.

2.5.4 Join

When a node J wants to join the network it first sends a join request to a node X which it has prior knowledge about. This node is presumed to be close with respect to the underlying physical network. The node X forwards the request to the node which is closest to the identifier J in the node identifier space using the routing algorithm described above. All nodes which the join message passes through send their routing tables, leaf sets and neighbourhood sets to the new node and update their routing tables if necessary. The joining node then creates a routing table with as physically close nodes as possible from this information. When J has receives information from the node which is closest to its own identifier J will sent its resultant state to all nodes it knows about. These nodes can adapt to this information. J is now a part of the ring and can route messages.

2.5.5 Failure

A node is considered failed when its neighbours in the node identifier space cannot communicate with it. In order to recover from a failure, the surrounding nodes need to replace the failed node's entry in their leaf sets. This is done by requesting the leaf set from the node that is furthest away in the same direction as the failed node and comparing the partly overlapping leaf sets.

In order to keep the neighbourhood set up to date, a node tries to contact its neighbours periodically. If a failure is detected, the detecting node requests the neighbourhood sets from all its neighbours and updates its own set accordingly.

If a node notices that a node in its routing table is unreachable, it then sends messages to the other nodes on the same row in the table and asks them what node they have in that location in their routing table. If this does not yield a working node, then the node will start querying nodes further away in its routing table.

2.5.6 Differences between Chord and Pastry

Chord and Pastry are quite similar in many respects. They both provide a distributed hash table, and they both take the fundamental aspects of peer-to-peer into account. Some of these aspects are that nodes participate as equals, that each participating node has a limited amount of knowledge about the global state, and that nodes can join or leave the system without administrative intervention. There are a number of differences in how they try to accomplish this. The main differences are listed below.

Pastry and Chord have theoretically comparable performances. In practice Pastry may perform better, since it takes network locality into account. This does however depend on the topology of the network and how distance between two nodes is measured. In a worst-case scenario it will only cause extra overhead. In the work of implementing CFS [3] the ability to take network locality into account has been added to Chord.

Node failures in Pastry are handled by trying alternate routes. In Chord this is handled by retrying after the stabilize routine has been executed and thus Pastry adapts better when nodes fail. Since the routing algorithm will forward a message to any node with a prefix at least as long and numerically closer, there will almost always be several alternative routes available through the routing table or the leaf set.

The construction of the pastry routing table is not deterministic since it depends on which node contacts which during the joining phase. The routing table of pastry is deterministic since only the successor of a identifier 2^i steps away can be entered into the table.

Chapter 3

Distributed file systems

File systems have been around for a very long time and are a basic part of every computer system. The environment in which file systems exist have on the other hand changed dramatically over the last decades. In the beginning of the computer era, disk space was a scarce resource and there was very little focus on user-friendliness. Today the conditions are quite different - disk has become cheap and so has network bandwidth. The time humans spend however, is still valuable, so a modern file system needs to take new considerations into account.

What a user of a file system should expect today is high reliability and accessibility. The file system should be easily accessible from different workstations. If the system says that a file is saved, then the file should be there the next time the user wants to access it. Having good performance is also a basic demand. Since humans make mistakes and disk space is cheap in comparison to human time, the file system should also support some way of accessing older versions of files.

This chapter summarizes different aspects of distributed file systems which are relevant for the development of Keso. It begins by covering the main questions that are raised when implementing file systems for a distributed environment. This part is mostly based on Distributed Operating Systems by Chow and Johnsson [19]. In the second part a few case studies are made. AFS [7] is a centralized distributed file system that is widely used. CFS [3] and Ivy [4] are file system developed on top of Chord [1]. Finally Farsite [29] is a different approach to a peer-to-peer read-write file system.

3.1 Distributed file systems

The main goals behind designing a distributed file system (DFS) are performance and availability. Performance can be achieved through load-balancing and replication. Replication is also good for fault tolerance. Finally a DFS makes data available to multiple clients.

The main characteristic of a distributed file system is multiplicity of users, clients and possibly also servers and replicas of files. The dimension of timing is also important. Accesses can be made to different replicas at the same time and data therefore has to be protected from becoming inconsistent.

There are different kinds of distributed file systems - centralized and decentralized. All that are widely in use today are centralized, but there are a number of interesting research projects investigating the possibilities of decentralized file systems.

3.1.1 Semantics of distributed file systems

File systems can provide different semantics to the user. Many things, such as atomic operations and access control are fairly easy to achieve for local file systems, but they quickly become difficult when dealing with distributed file systems. POSIX [5] is a standard which, among other things, describes the most widely used semantics for file systems.

From an application point of view the semantics that are required of a distributed file system are the same as for local file systems and a goal for designing a distributed file system is often application transparency - applications should not be aware of that a file resides on a remote host.

3.1.2 Files and directories

Files are file system objects which contain the actual data. File data can be stored in many different ways, and is largely dependent on the underlying hardware and the application. The file system often associates files with information internal to the file system. This data is often called metadata. There are many different types of metadata that a file system might wish to store associated to files. Some examples include for instance owner, size, locking information, access control lists (ACL) and much more.

Files are ordered into directories which introduce the possibility of structuring files in an hierarchical way and letting different users have files with the same name.

3.1.3 Versioning

In a versioning system the original file is immutable. When a client updates a file it creates a new version. This way it is easy for users to recover old versions of files and the reason for doing this is that human mistakes is one of the most important reasons for data being lost. Elephant [17] is a versioning file system that keeps old versions of files until cleaned out. An external cleaning process keeps versions that are considered interesting (for instance only the last version from a rapid series of updates).

3.1.4 Replication

Replication is used in order to increase performance and availability. Replication transparency means that the clients are unaware of the replication.

Concurrency is an important issue in distributed file systems. Just as local file systems has atomic operations, so should distributed file systems. Operations to the file system from different clients should not interfere with each other. Reads from the file system can be concurrent and made from primary, all or any replica site. Only one client should be allowed to write at a time. All replicas cannot be identical at all times due to communication delays. However, the level of synchronization needed in the file system must be decided.

There are a number of different strategies to adopt. Some file systems aim at having all replicas identical at some points in time while others just update some of the replicas immediately and propagate changes to the others in a lazy fashion. A summary of the different techniques are listed in table 3.1.

3.1.5 Modern file system concepts

As distributed file systems have evolved, and more research has been conducted, several aspects has emerged which increases usability both for users and administrators. Location transparency and global namespace are some of these concepts.

Location transparency means that a client does not need to know where a file resides until it needs to access that particular file. It is also possible that a file may change location while the users continue to access the file without disruption.

If a file system supports a global namespace, it means that no matter from which computer a users accesses the file system, it still has the same structure. This may seem trivial, but most distributed file systems in use today do not support this, for

Write primary	A master is elected which handles writes. If it fails, a new master is elected
Write all	Writes are propagated to all replicas immediately. Decreases availability, since it will not be possible to write if one member fails
Write all available	Write to those nodes that can be reached. Might create problem in e.g. network partitions. Some kind of merge algorithm is needed
Quorum	A majority is needed for a write. Silent witnesses possible.
Gossip	Very relaxed. Updates are propagated when replicas communicate. Can take a long time to converge.

Table 3.1: Different replication semantics

instance NFS [20] and Samba[21]. The advantages of a global namespace is that it will be easier for users to roam around in the file system if a consistent view is provided. Users can also share files in a simpler manner by using paths in the file system as unique identifiers when communicating.

3.1.6 Access control

In a local file system, access control in its simplest form is fairly easy to manage. There is only one running kernel, which is responsible for keeping users apart. All privileged operations has to pass through the kernel which can then perform access control. In the distributed case however, things are not quite as easy. Now we possibly have misbehaving or even malicious clients, which should not be allowed to disrupt the entire system.

There are a number of different semantics for distributed access control.

client - server	The server has a list of trusted clients and delegates some of the authentication to the clients (NFS)
user - server	The user authenticates to the server directly, independently of which client the user is using (AFS)
user - user	The user authenticates to another user, bypassing the server. This can be achieved by using end-to-end encryption

3.1.7 Quota

Quota is an administrative limit set on the maximum size or the maximum number of files within a certain domain. Such a domain can be a user's home directory, all of a user's files within a file system or the files of a project group. The file system implements quotas by increasing and decreasing counters as operations are committed to the file system. The reason to have quotas is that computer resources are usually shared among many people and a single user on such a system should not be allowed to fill the whole disk.

3.2 The Andrew File System

AFS [7] is a distributed file system with centralized storage. It was developed in the late 80's and early 90's at Carnegie-Mellon University in Pittsburgh, USA. From the outset, AFS was intended to scale well, and this has influenced many of the design decisions.

AFS has been in use for about 15 years and is used by many large installations around the world. Some examples are Carnegie Mellon University, Massachusetts Institute of Technology, Intel and Morgan and Stanley.

3.2.1 Design goals

AFS was designed from the beginning with both scalability and security in mind [6]. Some of the principles considered when designing AFS were:

- Distributed processing - Use the computer power of workstations when possible. This causes the system to scale better, since the amount of computing power will increase as the number of participating nodes increase.
- Caching - File access patterns exhibit locality in both space and time and makes caching worthwhile. If a reference is made to a cached file, the client will not have to contact the server, which increases performance.
- Minimize distributed knowledge - Keep as little global state as possible. For instance, servers only have knowledge of the clients that have recently accessed files, and clients only have knowledge of the servers they have cached data from.
- Minimize trust - In AFS, all clients are untrusted until a user authenticates from that client. Only the servers trust each other, and this is implemented using cryptography to minimize probability of forgery.

- Group together operations - Grouping operations makes it possible to increase throughput at the cost of latency. AFS has thresholds where the client switches from single requests to batch requests, in order to provide both low latency and high throughput depending of the usage pattern.

All of these principles works together in making AFS a highly scalable file system. AFS also has several properties which cause it to be flexible and easy to use and administrate. These include

- Location transparency - A user does not have to know on which file server a file resides, but can refer to the file by a logical name, and the system will find the file.
- Global namespace - All computers which participate in an AFS system has the same namespace, meaning that if a user moves from one computer to another the file system will still provide the same view.
- Administrative delegation - Most operations can be delegated from the administrators to a user or groups of users. Users can then delegate rights to other users, if they have the privilege to do so.

3.2.2 Volumes

Files in AFS are organized into volumes. A volume is an administrative part of the file tree. A user's home directory, the installation directory of a software package or a project area are typical examples of data that is often organized as a volume. The layout of an example file tree is shown in figure 3.1.

All files in a volume are kept on the same file server and on the same partition. AFS is location transparent, which means that a client do not have to know where a file resides beforehand. To aid the client in finding the file, AFS uses the Volume Location Service. It provides a mapping between the name of a volume and the location of it. When a client wants to access a file, it will contact the Volume Location Database and will get a list of servers where the volume resides. Using this information, the file can be found.

Volumes are also the administrative units for quota. An administrator can set a quota on the volume, and the sum of the files in that volume may not be larger than the quota limit. This is enforced by the file server, and for each write operation, the quota usage for a volume is recomputed.

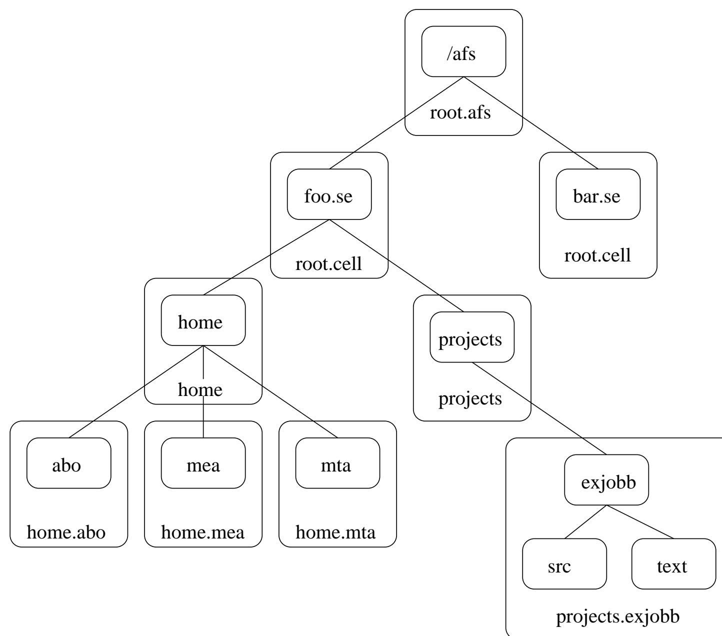

Figure 3.1: Layout of a file system, showing volumes

3.2.3 Access Control Lists

Each directory in the AFS file system has an associated Access Control List (ACL). An ACL is used to grant or limit the rights a user has to the resources the ACL is associated with. The ACL of a directory applies to all the files in that directory. The set of rights in AFS has a finer granularity than a traditional UNIX file system. The list of possible rights are shown in table 3.2.

r	read	May read files
l	list	May list files in the directory
i	insert	May add new files to the directory
d	delete	May remove files from the directory
w	write	May write to existing files
k	lock	May lock files
a	admin	May change the ACL

Table 3.2: AFS rights

3.2.4 Authentication

AFS relies heavily on kerberos authentication. A kerberos Key Distribution Central (KDC) is a trusted third party which shares secrets with all participants in the system (including the AFS file system). This enables users to authenticate to services in a secure manner. The details of kerberos authentication is outside the scope of this document. [8]

Since one design goals of AFS was to trust as few parts of the system as possible, all authentication is done between the user and the file servers. The client computer only forwards requests on the users behalf, but it has no privilege on its own, and if the users logs out, the client loses its privileges.

3.2.5 Callbacks

When a client requests information about a certain file the file server grants the client a callback for that file. The callback is a promise that the file server will tell the client if the file is changed. When a file server can no longer guarantee that a client has the latest version of a file it can issue a callback break to that client. A callback may be broken if a file has changed, but it may also be broken if the server cannot maintain the callback for any other reason. Such reasons may be if the server has too many outstanding callback and cannot accommodate any more clients. It may then break old callbacks in order to make room for the new clients.

When a client receives a callback break it marks the file as invalid. If the data relating to that file is requested again the client contacts the file server and requests the current status of the file. If it discovers that the file data version on the server is the same as the cached version, the client will receive a new callback and continue to use the cached version. Otherwise it will fetch the file from the file server again.

3.2.6 Write semantics

AFS uses the upload/download semantics for writing. This means that the client will download the file before the write starts, make changes to the local copy, and upload the file again when the write has finished. The write will only commit to the file server when the application that opened the file closes it. Concurrent writes to the file server from multiple clients are handled by letting the last writer win.

3.3 CFS and Ivy

Ivy[4] and CFS[3] are developed by the Parallel & Distributed Operating Systems Group at Massachusetts Institute of Technology Laboratory for Computer Science, which is the same group that developed Chord[1]. CFS is a publisher only file system, in the sense that there can be many parallel file trees, but only one publisher per tree and each tree is immutable. Ivy[4] is a read/write file system that provides NFS-like semantics when the network is fully connected and it is intended for a small group of participants.

CFS and Ivy have several similarities. They are both completely decentralized and they have thus no dedicated components. Both are based upon Chord and the DHash distributed block storage (described below) and both have similar ways of splitting files into blocks and using content hashes as keys.

Both file systems have been implemented using the SFS toolkit [9] and is modeled using the SFSRO [10] file system format.

3.3.1 DHash

DHash provides the block storage, caching and replication using Chord as a routing infrastructure. Data blocks are stored on the nodes that are responsible for the datablock's identifier in the Chord layer and DHash makes sure that the blocks are replicated to the k following nodes in the Chord layer. This replication scheme has the benefit that if the node responsible for a specific identifier fails its successor, that will take over the responsibility for that identifier, already has a copy of the data. Since replication is built into DHash rather than Chord, knowledge about the surrounding nodes must be present in both Chord and DHash.

DHash also provides caching throughout the system. When a client requests a piece of data, it will contact a number of nodes until it reaches a node which either is a replica of the data or which has the data cached. When the client has fetched the data it will send it to all nodes along the lookup path. DHash uses a least-recently-used order when reclaiming cache space. This will cause nodes which are closer to the replicas to maintain cached copies, since they will be referred to more often than nodes which are far away.

The configuration of nodes vary with respect to storage capacity and network bandwidth. DHash uses the notion of virtual servers [34] in order to handle this. DHash configures a number of virtual servers and each virtual server receives the responsibility for a part of the identifier space.

DHash is well suited for applications where the contents of blocks always determine where the block is stored. It is not well suited for applications where some other, external identifier is used.

3.3.2 Public key and content hash blocks

Both CFS and Ivy use the notions of public key and content hash blocks. A public key block is associated with an assymetric key pair and the integrity of the block can be verified by a signature with the private key. The public key of the block is used as the blocks identifier.

Content hash blocks use the SHA1[26] hash of the blocks content as the DHash key. Thus the client can verify the correctness by the data by calculating the hash of the received data and comparing it to the key. Since a content hash block always will contain the same data it can be cached without any problems.

3.3.3 CFS

CFS is designed to be a publisher write only file system. This means that there are, at one point in time, multiple file systems created by different people, in the same distributed system.

The publisher of a file system first inserts all the data for the file system as content hash blocks. Afterwards a root block is created and signed by the publisher. The public key should be the same as the key for the root block. Only the publisher knows the corresponding key and is thus the only one that can modify the file system.

Since all other data is refereed to by content hash keys it cannot be modified without it being detectable.

3.3.4 Ivy

The main goals for Ivy is to show how a distributed file system can be developed on top of untrusted components and to explore distributed hash tables as building blocks for more sophisticated systems.

The major difference between CFS and Ivy is that Ivy is designed as a read/write file system and it thus has to handle write-operations from several different users without the data becoming inconsistent. This is achieved by the use of logs. Each participant

keeps a log with its own file system operations and the log is stored in DHash to make it available even if the participant leaves the system. The log is divided into several blocks and the first entry in the log, called the loghead, is given a public key so the other participants know how to find it. The remainder of the log entries are content hash blocks.

When a participant wishes to access the file system it chooses which logs to trust and retrieves sufficient information from these logs. The set of logs that a number of participants has agreed to trust together forms a view. A view comprises the file system and it has a view-block that contains all the log heads of the trusted participants.

Traversing all the logs of all the other participants is a quite cumbersome operation and in order to avoid having to do this all nodes keep private snapshots that contains the current state of the file system. These snapshots are stored in DHash using content hashing, which makes Ivy reuse the storage space. In order to update a snapshot Ivy only collects log entries that are newer than the last snapshot.

Consistency and resolving conflicts

Each entry in the logs is given a version vector, that contains the information about the last known operation preceding the current operation. If two participants make simultaneous writes the conflict is resolved by comparing the participants public keys.

This makes Ivy have slightly unusual semantics. If ulink("a") and rename("a", "b") are made simultaneously both syscalls will return success, but only one will actually take place.

Ivy further provides close-to-open consistency, which means that Ivy writes the file at the close operation.

3.4 Farsite

Farsite [29] is a file system developed by Microsoft Research in Redmond, WA, USA. The designers of Farsite has taken many of the ideas behind peer-to-peer computing into account while designing the file system. Farsite facilitates such features as large-scale distribution, data replication, security, access control and self-organization. Farsite also uses public key cryptography for user authentication and authorization. The file system also tries to provide traditional file system semantics. In particular it tries to mimic the semantics of NTFS, the native file system of Windows NT as far as possible.

Unlike CFS and Ivy, Farsite is not based on the idea of Distributed Hash Tables. Instead, participating computers form groups dynamically and replicate data and metadata across these groups.

3.4.1 Files and directories

The two most important concepts in Farsite are those of a directory group and a file host. A directory group is a set of hosts who are collectively responsible for the metadata of a part of the file system. A client directs a file system request to the group as a whole, and each member of the group processes the request and responds. The directory groups maintain consistency using a variation of the Byzantine agreement protocol described in [30].

Metadata is only a small part of the data in a file system. The major part of the data is stored in the actual files. To allow for the division of load over more machines, a level of indirection has been added. A directory entry for a file contains a cryptographicly secure hash [12] of the contents of the file and a set of file hosts where the file is stored. The client can then retrieve the file from any of these hosts and use the hash to verify the contents. No agreement protocol is needed to ensure consistency among the file hosts since the hash can be used to ensure that no tampering is done.

3.4.2 Security

Security in Farsite is based on public key cryptography. Each user is assigned a pair of asymmetric keys [12]. In order to delegate rights to a user, that users public key is added to an access control list (ACL) for the file or directory. This ACL is stored by the directory group.

When a file is stored in the file system, it is encrypted to ensure privacy. First, the file is split into file system blocks. A one-way hash function is computed over the contents of the block and this is used as the key for encrypting the block using a symmetric crypto. This process is repeated for all the blocks of the file, giving one hash for each block. Another symmetric key is then generated using a random function, and the hashes are encrypted using this key. Finally, this symmetric key is encrypted using the public keys of all users who are supposed to be allowed to have access to the file. This gives the benefit of data privacy and access control, while it still gives the possibility to only store one copy of each unique datablock. This method is described in [31].

Metadata is protected by encrypting it before it is saved to the directory group. To ensure that a malicious client doesn't write syntactically incorrect metadata, a technique called exclusive encryption [32] is used. This technique ensures that metadata is

syntactically correct while the actual data is still protected.

3.4.3 Administration

To start up a Farsite file system, an administrator creates a file system root by designating a set of nodes which should make up the root directory group, and then delegates the file system to them by creating a certificate for the root. This certificate can be used by the directory group to prove to a user that the group is indeed responsible for the file system.

When the load of the root directory group becomes too high, it can delegate a part of the namespace to another set of nodes. A set of nodes are selected by the parent directory group and a certificate is issued giving the new set the authority over that part of the namespace. This certificate is signed using the certificate the parent group received from its parents.

When a client wants to access a file, it will contact the directory group responsible - which can be found by traversing the path from the root - and will ask for information on the file. The directory group will then answer with the file information and all certificates needed to verify that the group is responsible for that part of the namespace.

Chapter 4

The DKS Overlay Network

One of the major parts of our master thesis has been to implement the DKS[13] system in C++. DKS is a family of structured overlay networks that is being developed in a collaboration between KTH and SICS, the Swedish Institute of Computer Science. DKS is quite similar to the Chord system, the main differences are that DKS is more tunable, it has atomic joins and it uses a correction-on-use algorithm that handles failures without the periodic overhead messages created by Chord and Pastry.

4.1 Design objectives

DKS is intended for large scale peer-to-peer systems and the aim is to provide a reliable system. The main objectives of the DKS design are:

- Scalability: performance should be acceptable even for very large systems.
- Dynamism: DKS is intended for a dynamic environment in which nodes should be able to join and leave the system in a high rate without affecting the system too much. The system should self-reconfigure in a way that converges towards optimal (or near optimal) configuration.
- Low bandwidth consumption: Since network latency is one of the main bottlenecks in distributed systems bandwidth consumption should be reduced as much as possible. The aim is to have acceptable communication costs in both periods of high and low dynamism.
- Fault tolerance: The system should be able to perform the specified service, even if some nodes have failed.

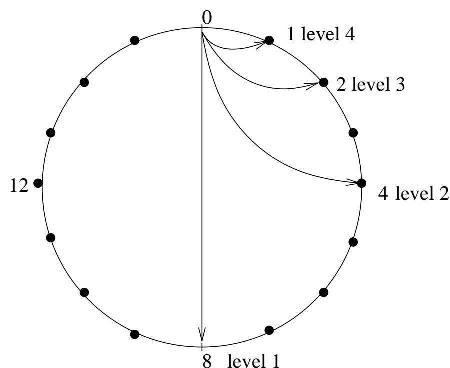

Figure 4.1: The pointers in a DKS routing table when the network is fully populated

- Strong guarantees: Data that has been inserted into the system should be possible to retrieve.
- Tunability: DKS should be easy to adapt to networks with different numbers of participants and be tunable in a way that creates a good balance between the diameter for the overlay network and the routing table at each node.

4.2 System overview

N is the size of the node identifier space and a DKS system can at most contain N nodes. Each node is assigned an arbitrary node identifier (for instance the SHA1-hash of the node's IP-address). The nodes are arranged in a circle with each node keeping pointers to its f followers according to node identifiers. As in Chord, DKS is depending on a correct successor pointer for the system to be in a consistent state. DKS also keeps track of its immediate neighborhood in backlists and frontlists.

Data is mapped onto the ring using the same address space as the node identifiers. The node with the closest matching identifier higher or equal to the data key is responsible for that piece of data.

In order to achieve lookups in $O(log(n))$, DKS uses routing tables. The routing table is divided into different levels and intervals in a treelike fashion. This way a node can calculate the optimal routing table entries at different levels and intervals for all other nodes.

Level	Interval	Nodes	Responsible
1	I_1^1	[0,8[	0
	I_2^1	[8,0[	8
2	I_1^2	[0,4[	0
	I_2^2	[4,0[	4
3	I_1^3	[0,2[	0
	I_2^3	[2,0[	2
4	I_1^4	[0,1[	0
	I_2^4	[1,0[	1

Table 4.1: A sample DKS routing table at node 0 (Figure 4.1). N=16 and k=2

4.3 Routing information

The DKS routing table at a node is arranged so that pointers are kept to $log_k(N)$ nodes at exponentially increasing distance. The entries for the first level are chosen by dividing the identifier space into k intervals and keeping pointer to the first node in each interval. The succeeding levels are created by repeatedly dividing the first interval from the previous level until it cannot be divided anymore. The current node is always responsible for the first interval. Figure 4.1 shows the resulting levels and intervals for k=2 and the node identifier space (N) of size 16. Table 4.1 shows the corresponding routing table.

The second interval on the first level (in our case the interval [8,0[) is denoted I_2^1. The first interval in the second level (i.e. after the second division of the identifier space) is consequently named I_1^2. Table 4.1 shows a routing table containing the levels, intervals and corresponding responsible nodes of the ideal network described above.

In most cases, however, the network is not fully populated and the optimal node for a specific place in the routing table does not exist. In this case DKS keeps a pointer to the immediate successor of that node. This is shown in figure 4.2.

4.4 Lookup algorithm

When a node wants to perform a lookup for a key in the network it first finds the interval within the first level the key belongs to. If the responsible node for that interval is the node itself, it continues to look at the second level and so on until it finds an interval for which the node itself is not responsible and the message is forwarded to that node. The interval that was used is attached to the message and is

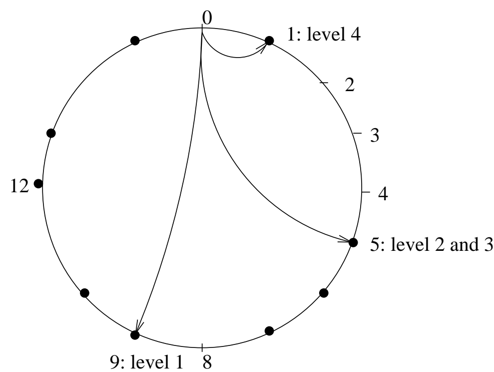

Figure 4.2: The pointers in a DKS routing table when the network is sparsely populated. Dots represent existing nodes and lines represent empty identifiers.

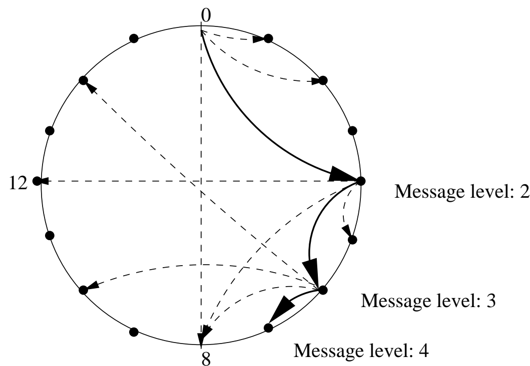

Figure 4.3: A lookup from node 0 for key 7 in a fully populated network of size 16 and k=2. Dashed arrows represent routing table information and thick arrows represent the path of the lookup message.

used for forwarding and correction on use (described in the next section).

When a node receives a lookup request that has been forwarded to it, it first checks whether it is responsible for that key or not. If it is not responsible it starts looking for the correct node to forward the message to. Since the size of the interval is divided by k for each level, the receiving node knows that the interval it is looking for cannot be larger than the previous interval size/k and thus it must look at the messages original level+1. The algorithm is described in table 4.2 and a picture is shown in figure 4.3.

```
level=MSG.level+1
target=MSG.target
Let I^{level}_{interval} in Routing Table[level] such that target belongs to I^{level}_{interval}
destination=Routing Table[level][interval]
while(d==n) do
      level=level+1
      Let I^{level}_{interval} in Routing Table[level] such that target belongs to I^{level}_{interval}
      destination=Routing Table[level][interval]
od
MSG.level=level
MSG.interval=interval
send(n:d:MSG)
```

Table 4.2: The core routing-algorithm in DKS

4.5 Maintainance

4.5.1 Join and Leave

Join and Leave operations are designed to avoid lookup failures generated from nodes leaving or joining concurrently and they are performed in an atomic fashion.

When a node A wishes to join it sends a lookup request message for its own identifier to an arbitary node B in the network. That node performs the lookup and sends the reply, node A's closest successor C, to node A. A then asks C to insert it into the network. C calculates initial routing tables and front-/ and backlists for A and transfers suitable state. The initial routing table calculated for the node might not be correct. If so, it will be corrected via the correction-on-use-algorithm.

A node can leave the network in two different ways - it can either leave in a controlled manner or it can fail. The reason for distinguishing between these alternatives is that a node that leaves in a responsible way can cooperate and help to avoid causing inconsistencies and data loss. Nodes that fail cannot cooperate and the surrounding nodes thus has to detect that it has failed and take action in order to remove the failed node from routing tables and back/front lists. The surrounding nodes also have to ensure that a sufficient number of replicas of data are still maintained.

4.5.2 Failure and correction on use

Since a node does not receive information about changes further away than the size of the back- and frontlists, a mechanism is needed for nodes to keep correct information

in their routing tables. Chord does this via the stabilize protocol and Pastry via a complicated mechanism including a heartbeat protocol for the neighbourhood set and searching in widening circles to heal the leaf set.

Both of these strategies create unnecessary overhead in the case of the network behaving correctly. Therefore DKS instead tries to make use of information embedded in the lookup messages to correct erroneous information when it is discovered.

Messages in DKS are forwarded in a predictable way and thus a node that receives a message can calculate the optimal entry for the sender to have in the routing table position corresponding to the messages level and interval. For instance the optimal entry at node 3 for I_2^1 is the same as I_2^1 at node 0 plus 3 modulo the size of the ring (i.e. N).

If the receiving node is not responsible for that key, it knows that the sender has incorrect routing information and can thus look at its own routing table and find a better candidate for that position and then tell the sender about this new candidate. The backlists are used in order to speed up this procedure - if the correct entry for that level and interval has a lower identifier than the receiving node, it would not know about it unless it kept a backlist and the sender would thus be directed to the other end of the ring.

This scheme makes DKS only send messages in order to heal and update routing tables if the network actually has changed. This is done in exchange for the small overhead of sending the level and interval embedded in the lookup messages.

If a node in the path is impossible to contact, DKS can send the message to any node that lies between the sending node and the destination - the choice only affects performance since all nodes will cooperate to forward the message closer to its destination. Choosing a node that is close to the optimal entry for the routing table does however increase the chance of receiving relevant information in order to heal the routing table.

Chapter 5

The Keso File System

5.1 Design Objectives

As stated in previous chapters the goal for this project was to create a secure peer-to-peer file system that takes advantage of unutilized resources and avoids unneccessary storage of data with the same contents.

There are also several other design goals and they include Keso to be:

- **suitable for a real world environment.** Keso should be designed to provide the features that are expected from a modern file system.
- **a reliable file system on top of untrusted components.** All participants in a completely decentralized file system cannot be trusted in the same way as a traditional server.
- **secure.** Users should be able to trust that data stored in the Keso file system will not be lost or compromised by malicious users.
- **fault tolerant.** As long as no more than a configurable number of nodes fail, Keso should still provide the expected functionality and no data should be lost.
- **self-organizing.** No human intervention should be needed in order to make the system recover from failure, and data replication and load balancing should be handled automatically.
- **scalable.** Keso is intended for large systems and must thus scale well. This includes that no operation can be in exponential order of the amount of data in the system or number of participants.

- **making use of spare resources.** The excess disk space at ordinary workstations should be taken advantage of.
- providing **access control** and **administrative delegation.** Only users with the right to do so should be able to read and modify data. Users should also be able to form their own groups and delegate the right to read and modify some of their own data to those groups.
- **versioning.** One of the most common reasons for work being lost is human mistakes and Keso should provide easy means for recovering old versions.

In this section we will specify these design goals further and present some of the reasoning behind the acctual design.

5.1.1 Assumptions

One of the basic assumptions in the design of Keso is that there is an organization behind the file system. This makes users known to the organization and creates a need for the file system to support identification and access control. Users in an organization also work together towards some common goals and they need to be able to cooperate. The file system should provide support for multiple users to have access to the same files.

We further assume that the file system will be used in an open environment and therefore accessible from any computer at the Internet. This makes it necessary to adapt Keso to a hostile environment.

Since we wish to make use of the spare resources at ordinary workstations we have to accept that the computers that store the data in the file system might be compromised.

We do not expect to achieve a system which outperforms some of the distributed systems in use today. There are several reasons for Keso to be interesting anyway. Keso will support more things than a traditional file system and some performance degradation will be acceptable. Another reason is that the traditional network file systems have been around longer and have become quite optimized. We will not focus on creating an optimized system - our focus is to show that this kind of system can be done. We do, however, expect our system to perform reasonably well.

5.1.2 A real world environment

Our goal with Keso is to design a file system which, properly implemented, has the ability to be usable in a real world environment. The file system should provide Unix-like semantics [22]. It has to provide all the features that users have come to expect from a file system, such as reliability, access control and POSIX compliance [5]. We have used AFS[7] as an usage model in order to see what other features that can be expected from a modern distributed file system.

The file system is designed for medium to large scale organizations, but should work with smaller groups of participants also. This means that it will need to handle large amounts nodes and data, and that it must scale well.

A file system needs to provide consistent versions of data and protect metadata consistency. Keso needs to handle this in the case of network partitioning. In order to do this we have tried to minimize the amount of sensitive data. Creating new versions after each update is one way of doing this - this way concurrent updates will in the worst case create two different versions. Using content hashes for file data also reduces the amount of updates that need protection. If there are two updates regarding a block with the same content hash key these two updates should have the same contents, since the keys would not match otherwise. Therefore it does not matter which order they execute in. This is discussed more in detail in section 5.4.3.

The data that has to be protected is the metadata consistency of the file system. We have chosen to distribute this as much as possible in order to achieve distributed processing and high scalability. Making nodes that store a directory responsible for updates related to this directory provides a serialization point for metadata updates.

5.1.3 Security

Security in Keso is based on three primitives:

- minimize trust
- trust users, not computers
- correctness of the received data should be verifiable

All systems are vulnerable to some extent. Our goal is that data should not be compromised or removed from the system as long as the attacker does not control more than a configurable number of participating computers and that these computers

should not be possible to pick by "random". If data is removed or compromised this should be easy to detect.

One assumption is that there is an organization which the file system is intended to serve and that there is someone who is responsible for configuring nodes participating in the file system. This will lead to that no node is a server in the system without a minimum level of trust.

The basic idea behind access control is to separate the *privilege* to access data from the *ability* to access data. The privilege to access files means that the user is allowed to access the file by the system, while the ability means that the user possesses the means to access the information.

It is reasonable to assume that one node is easier to crack than a traditional server, since ordinary and potentially malicious users will be allowed to log in on the nodes. Therefore security cannot be based on that all nodes are uncompromised. We cannot however completely avoid that a single client breach might be enough to cause the whole system to be compromised - as in other file systems Keso will be vulnerable if a machine used by an administrator is compromised.

5.1.4 Accept some level of denial of service vulnerability

If a malicious user has control over machines on the local network, they can always in one way or another cause servers in traditional systems to become overloaded and fail to reply to requests or reply very slowly. Hence some vulnerability to denial of service attacks might be acceptable, even if we have tried to avoid to integrate this kind of vulnerabilities in the design.

The available peer-to-peer routing-layers do not solve the problem that a single compromised machine in the path is sufficient to cause a denial of service. We have not focused on creating a system that is not vulnerable to denial of service attacks on the file system level, even if we have tried to avoid making it easily attacked.

5.1.5 Tamper protection

We aim to develop a system where it is impossible for a user to change data in the system without being discovered. This is important, since this can make users trust the system to a certain extent.

The system should provide means for data authenticity, integrity and confidentiality.

Data authenticity means that when data is retrieved it is the correct data. Integrity means that the data inserted into the system should remain unchanged while in the system. Confidentiality means that only those authorized to access the data should be able to do so.

5.1.6 Fault tolerance

Ordinary distributed systems usually suffer from availability problems. They are often designed with one or several single points of failure, and to reduce this, expensive hardware or cumbersome software solutions are often used. Keso should eliminate most or all single points of failure and make replication transparent to the users and administrators.

There is a built-in contradiction between fault tolerance and performance. [35] It is therefore a requirement that it is possible to configure the number of participants that can fail without data becoming unavailable.

The main thought behind fault tolerance in Keso is that there will be no dedicated components and that if a node fails, another one will be able to take its place. This introduces the need for replication.

5.1.7 Self-organization

The administrator of a traditional distributed file system has to spend a considerable amount of time managing the organization of the file system. By this we mean what files are stored on which servers, moving files if one server becomes too heavily utilized, and such tasks. If a server breaks down, the administrator has to quickly set up a new server and restore the data at the new location.

Keso should be self-organizing in the sense that no administrative intervention should be needed to ensure that data is layered out in an efficient manner throughout the system. The system should also ensure that the data remains available as participants leave and join. It should also be able to cope with a participant failing.

5.1.8 Scalability

It is not uncommon in modern world to have tens, or even hundreds of thousands of computers in a single computer system of a large corporation. The amount of data stored in distributed file systems grow with time and can be expected to do so for any

foreseeable future.

In order for Keso to be scalable an efficient overlay network is needed, since communication costs are expensive. The need for comunication between nodes, especially when it comes to expensive operations as distributed locking, should be reduced as much as possible - no operations should scale lineraly (or worse) with the number of participating nodes or the amount of data in the system. Minimizing the need for distributed knowledge reduces the need for operations that has to traverse a large number of nodes.

To achieve good performance under high load processing should be distributed as well and it should be possible to cache as much data as possible without complicated and expensive procedures.

5.1.9 Making use of spare resources

As stated in previous chapters as much as 50 percent of the local harddrive space might be unused at workstations in large organizations and one of the major design goals is to take advantage of these resources. Decentralizing and distributing processing are means to achive this.

5.1.10 Access control and administrative delegation

The traditional ways to handle access-control only include limited granularity and intervention from the system administrator. We want to be able to delegate and determine rights on a much finer level.

The AFS[7] usage model has proven suitable for many large scale organizations and we thus wish Keso to provide the same functionality.

5.1.11 Versioning

Computer resources are quite cheap compared to user time [17]. Therefore it is important that the main focus of a file system is to provide a user friendly environment and that data under no circumstances will be lost. Research has shown that a quite common reason for data loss is human mistakes. We plan to provide means for solving this by keeping old versions of the file-tree in the system in a way similar to the Elephant [17] file system. This also has a few other features in the case of a distributed file system, as shown in later chapters.

5.1.12 Using the DKS overlay network

There are several advantages with using DKS as the foundation for Keso. The main reason is that DKS provides an efficient lookup service in large networks. By using DKS we have limited both the amount of knowledge needed at each node and the amount of traffic needed for communication to be logarithmic in the number of nodes which are participating in the system.

Creating a self-organizing file system requires the design to be adapted for dynamic systems which change over time. In the DKS overlay network no administrative intervention is required when nodes join or leave the system and this does thus provide a good foundation for a file system with the same characteristics. The DKS is completely decentralized and handles nodes failing unexpectedly by replacing the failed node with its successor. Another aspect of self-organization is that as long as node and data identifiers are chosen in an uniform manner load will also be uniformly distributed over the participating nodes.

In order for nodes to be able to take the responsibilities of a failed node, replication is necessary. DKS replicates objects on the succeeding nodes in the identifier space. This makes the first replica automatically responsible after its predecessor has failed.

A "virtual harddrive" can easily be created by using the distributed hash table service from DKS, where the block key is used instead of a physical adress. This way existing knowledge about designing local file systems and organizing metadata can be reused in a peer-to-peer context.

5.2 An overview of Keso

The basic building blocks of Keso are files and directories. A directory provides a name to file location mapping service. The location for a directory is chosen at random when the directory is created and it remains the same throughout the lifetime of the directory.

Files consists of data blocks and inodes and these types of data are handled identically at the DKS layer. The keys are determined by the SHA-1[26] content hash of the file in a way that is inspired by CFS[3] and Ivy[4]. Since SHA-1 hashes are designed to have an uniform distribution this means that we get a statistically uniform distribution of file data among the nodes.

Security and data integrity is solved by using a combination of asymmetric and symmetric cryptography. Files are encrypted with a symmetric encryption algorithm, such

as AES[27] and the symmetric encryption key is encrypted with the public keys of the users and groups that are to be allowed to read the file. Data is also signed when written in order to make sure that the file was written by a legitimate user. We use a model of combining contents hashes and cryptography that is similar to Farsite[29] and it results in that we only have to store duplicated data once, and that way we can preserve storage space.

Keso keeps old versions of files in the file system in a way that is inspired by Elephant[17]. Each time the user writes to a file, a new version is created. This enables the user to go back and look at earlier views of the file system and retrieve files which have been overwritten or deleted.

5.2.1 Files

Traditional local file systems often have directories which map names to index nodes which in turn maps to the actual file data. This gives a level of indirection and a flexibility in where to place data on the physical disk. We have tried to use this idea in Keso and have a similar structure with datablocks, inodes and directories.

In order to store a file it is split into blocks and encrypted. These datablocks are then referenced from an inode. An inode represents a specific version of a file and it consists of a blocklist of the datablocks of the file and some metadata relating to the file. The inode also contains the keys needed for decryption of the datablocks and the whole inode is encrypted as well to ensure data privacy. The structure of an inode is shown in figure 5.1.

The metadata contains data about the file itself, such as the time this version of the file was created, the list of permissions to use when users tries to access the file and who created it.

The blocklist is a list of identifiers of the blocks which contains the data in the file. Since all blocks except the last are of equal size, it is easy to determine which block to fetch if the user wants to access data in the middle of the file.

5.2.2 Directories

Directories provide a mapping between names and inodes. A directory contains a set of metadata, a directory lookup table and a signature. An example directory is shown in figure 5.2.

The directory metadata is similar to that of files, with modification time, which user

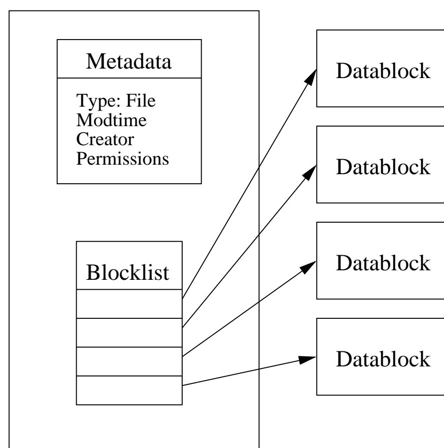

Figure 5.1: An inode, with metadata and blocklist

created the directory and the set of permissions.

The directory lookup table is a list of entries which each contains a name and a list of file change entries. Subdirectories are treated in the same way as files. Each entry consists of an operation, the inode this operation relates to, and the user which has performed the operation. Operation can be one of either Create or Delete, where Create means that a new file version has been created and Delete means that the file has been deleted.

Each subdirectory is referenced by a Create entry, until the time when the subdirectory is removed, in which case a Delete entry is inserted into the list. Each file update however results in a new inode version, and a new change entry is prepended to the list for that filename.

The integrity and correctness of the directory is ensured by a scheme of computing hashes and signing these hashes.

Since inodes are stored as separate structures, a move between two directories consists of adding a create entry in one directory and adding a delete entry in the other.

Directory data is the only data that needs to be protected from concurrent, conflicting updates in order to avoid the file system becoming inconsistent. This is made by the node that is responsible for a certain directory acting as a server for this directory. Updates are made in an atomic fashion, and a scheme that signs the last entries for all files in the directory update table makes sure that the node that requires the update is aware of previous updates in the directory.

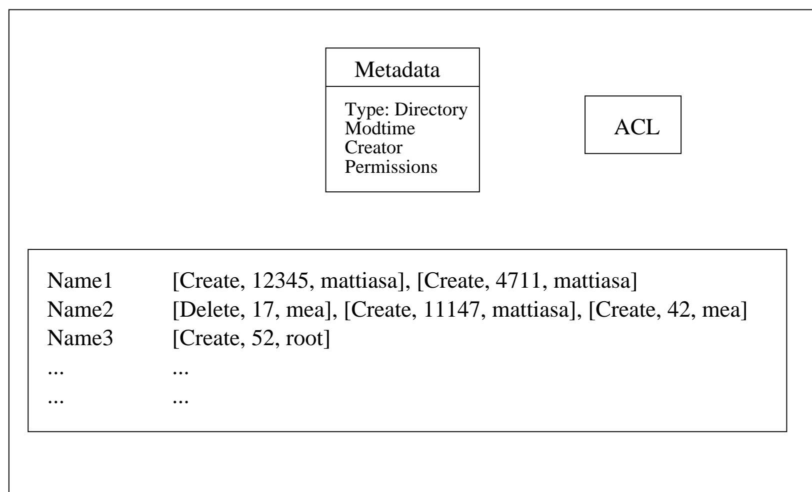

Figure 5.2: Layout of directory. It includes metadata, the list of file versions and a signature

5.3 Security and access control

5.3.1 Certificates and user authentification

Keso is designed with the assumption that there is an organizational structure behind the file system. This has lead us to base the authentication model on public key cryptography. This means that there is a certificate authority which can issue certificates for users and participating nodes. Each certificate consists of a public key and a private key. The public key can be known by everyone in the system but the private key is known only to the user. A user's certificate is signed by the certificate authority in such a way that if someone has prior knowledge of the public key of the certificate authority then validity of a user certificate can be verified. This is common practice and detailed description of public key infrastructure can be found in works such as "Applied cryptography" by B. Schneier [12].

5.3.2 Access control

Each directory contains an Access Control List which contains the public keys of those users who are supposed to be allowed access to that directory. For each of these keys the ACL contains a list of the rights the owner of the key should have in the directory. The list of access levels are listed in table 5.1.

r	read	May read directory and files
i	insert	May add new files to the directory
d	delete	May remove files from the directory
w	write	May write to existing files
a	admin	May change the ACL

Table 5.1: Keso rights

It is the responsibility of the node that stores the directory to enforce these access restrictions. The node will only send information in the directory to a user which can present proof that she really is a user which has read permission. Likewise, the node will only accept updates from a user which has the rights to update that part of the directory. However, as we will see later, only users which have knowledge about the private keys associated with the public keys on the ACL will be able to decrypt and read the actual data and make updates to the directory that can be verified as correct.

5.3.3 Data privacy

We have now ensured that only users who have permission to retrieve or write data should be allowed to do so. However, a user which has control of a node which stores a directory can still read and write the files in that directory. In order to prevent this Keso uses strong symmetric encryption.

The encryption scheme used in Keso is inspired by that of Farsite [29] [31] and an overview of it is shown in Figure 5.3. Each file is split into blocks of equal size. A cryptographically secure hash, in this case SHA-1, is then computed for each block. This hash is used as an encryption key and the cleartext block is encrypted using a symmetric cryptographic function such as AES [27]. Another hash is then computed of the encrypted block, and the block is saved into the DHT using this hash as the location.

For each block, both of these hashes are saved in the blocklist of the inode. Each directory contains a symmetric key which has been randomly generated. The entire inode is encrypted using this key.

The directory key is then encrypted with the public key of each user and group that is supposed to be able to read the file and all versions of the encrypted key are stored in the directory.

A user who can prove that she is a user with read access is sent the directory key which is encrypted with the users public key along with the relevant data from the directory.

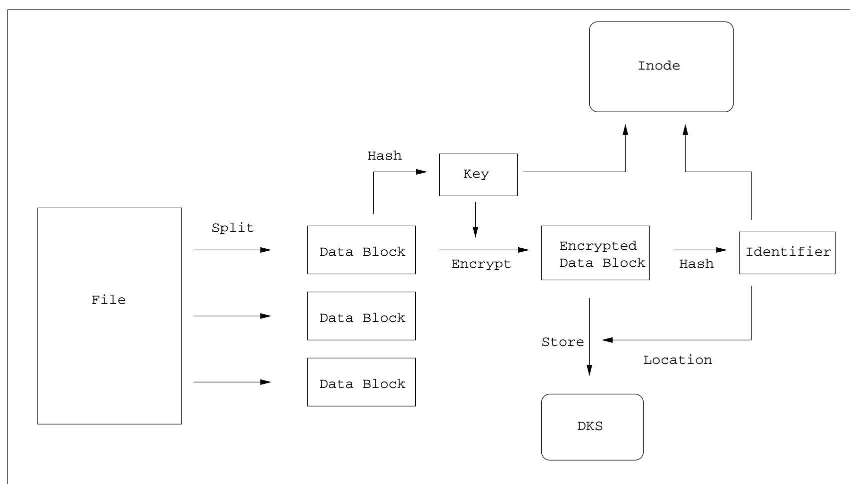

Figure 5.3: The scheme for storing a file.

The reason for this somewhat elaborate scheme is that this gives us the benefit that replicated data only is stored once even though it is encrypted. If the datablocks would have been encrypted with the symmetric key of the directory blocks of different files would have produced different ciphertexts although their cleartexts were the same.

Farsite uses a method to protect metadata called exclusive encryption [32]. This method ensures that metadata is syntactically correct while the privacy of data is ensured. It is possible that this method can be adopted to Keso in the future but no effort has been made to do so yet.

5.3.4 Tamper protection

There are several alternative strategies for implementing protection from data tampering. We have chosen to look at a few of these. The integrity of the disk blocks can be verified trough the hash which is listed in the inode. If the contents of the disk block has been changed, the hash will be different and the user of the file will be able to detect the tampering. Likewise, the integrity of the inodes are protected trough the hash listed in the directory.

The problem comes with the integrity of directories. We want to be able to track the history of a directory and have the ability to present a consistent view of the directory at any point in time. We also want to prevent anyone from changing the history of a directory. By this we mean that no one should be able to remove or change versions of files which have previously been written, without us being able to detect it. To

accomplish this we have several alternatives.

- Compute a hash of the entire directory, including all previous versions of files. Sign this hash using the writers private key and save it in a list of signatures. This means that the signer needs to retrieve the entire directory with all previous versions and has the possibility to fake history.
- Compute a hash of the entire directory as before, but include the previous signatures in the hash. This means that we can reconstruct a series of changes and verify that the differences between different versions.
- Each time a change entry is added to the directory, compute the hash of the current entry and the previous entry, and sign it. This makes it easy for us to verify each individual change. However, a malicious user may backdate certain files in a directory, while keeping the current version of others. This is undesirable.
- When an entry is added, compute a hash over all the current entries in the directory, and in addition, also compute a hash of the previous entry for the one just added and then she signs the hashes. This gives us the ability to track the history of the entire directory, and we can recreate the directory in any point in time.

Of these algorithms, the last one is the most appealing, since it gives us the ability to track the entire history, while only requiring us to sign a portion of the directory. The problem with all these solutions are that a malicious node still may give us an outdated version of the directory even if we ask for the latest version. This is a hard problem to solve. A solution could be to sign and update the parent directory or by using a voting protocol where the replicas ascertain the correctness of the reply. Both of these strategies are however expensive.

5.3.5 Groups

In order to increase the possibilities of administrative and privilege delegation some kind of group mechanism is needed. This can be implemented in Keso using the above described scheme of access control, with some additions. If a user wants to give access to a number of other users she can create a group and give access to that group. This is done by generating a public key pair for the group and encrypting the private key using the public keys of all the members of the group.

To delegate privilege to a file, the encryption key of the file now only needs to be encrypted with the public key of the group. This way members can be added to the group without having to iterate over all files where the group has access.

In order to securely remove a member from a group, nodes which store directories must verify that a person who requests an operation really is a member of the group.

5.4 File versioning and conflict resolution

5.4.1 File versions

Versioning in Keso is both a way of providing a service that users will benefit from and a way of handling conflicts when the file system for instance has been divided into separate sections due to network partitioning.

Each file can exist in a number of different versions. Each time a user writes to a file a new inode is created using the previous inode as a blueprint. Those datablocks that have been changed are written into DKS and the blocklist is updated. This inode is then inserted into the front of the version list for that name.

This means that if you want to view the file system at an earlier time you can iterate over each inode list and retrieve the relevant versions. This is a more expensive operation than just reading the latest version of a file, but it is also an operation which is performed more seldomly.

5.4.2 File deletion

If a file or directory is deleted by the user it is not actually removed from the file system. Instead, a marker is inserted into the inode-list, along with a timestamp. This means that you can recreate the exact state of the file system at any given time.

5.4.3 Network Partitioning and Disconnected Mode

After a network partition the version lists for each file are merged. If there are conflicts some kind of heuristics could be used in order to decide what version of the file is to be considered as the current version, for instance the version with the latest "real time" or the highest inode number. The user has to be alerted, however, in order to both manually verify the decision from the system and in order to resign the directory.

In order to use Keso in disconnected mode all relevant data has to be stored locally and this has to be done manually by the user. After the data has been modified while disconnected it can be reinserted into the system in the same manner as when the

network has been partitioned.

5.5 Storing data

In order for the DKS system to provide a reliable storage we have a layer between DKS and Keso that handles replication, caching and reliable communication. It is called the LocalStore and it is influenced by DHash[4].

5.5.1 Replication

Data stored at a node is replicated at the DKS layer to the f following nodes in the DKS layer, that is the nodes in the frontlist of the responsible nodes, and the responsible node is also the node that handles the replication. This way data will be found immediately after the responsible node has been discovered as failed, since the data already will be replicated to the following node, i.e. the new node that is responsible for the key. This is shown in Figure 5.4a.

Other possible replication schemes include replicating to the predecessors or to some more complex scheme. The advantage of replicating to the predecessors instead of the successors is that a request is more likely to end up at one of the predecessors before it arrives at the responsible node than it is likely to end up at one of the successors. The main drawback is that it ads the requirement of atomic updates of the replicas, which is significantly more expensive. Replication at the predecessors is shown in Figure 5.4b.

A more complex replication scheme could be designed to try to catch as many requests as possible as soon as possible. One example could be to keep all the replicas at an even distance away and spreading them out as much as possible over the identifier space. That way the requesting node can easily calculate the ideal position of the closest replica and send the request directly to that node. This is shown in Figure 5.4c.

5.5.2 Caching

Caching datablocks and inodes is completely uncomplicated. Since datablocks and inodes are identified by their content hashes they will be correct if they are found in the cache. Thus datablocks can be cached at all places they pass and be garbage collected using a least recently used scheme. Caching directories is much more complicated,

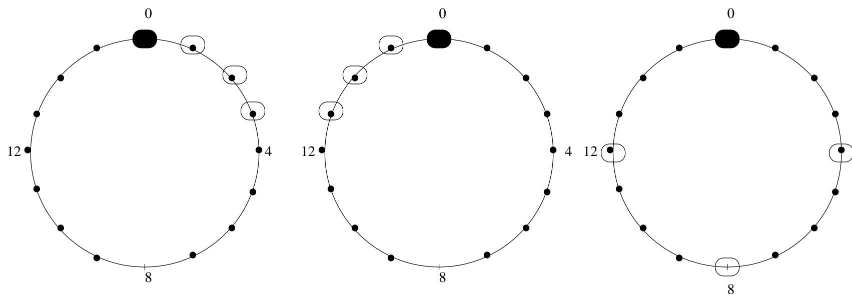

Figure 5.4: The different replication schemes at the DKS layer. Black box indicates the primary copy and white boxes indicate replicas. a) Replication at the successors. b) Replication at the predecessors. c) Replication at predetermined positions.

since they have the same location but change their content. We have therefore decided not to cache directories. This may lead to performance drawbacks and high latency.

5.5.3 Acknowledgments

In order to verify that a malicious node does not just throw away data that has been received the client node keeps data belonging to a request until acknowledgments from the replica nodes have been received. In order for the storing node to be certain of that it is correct nodes the acknowledgments are signed with the nodes private keys.

5.5.4 Storing of files - description

The table below describes the algorithm for storing a file.

```
split the file into blocks of appropriate size

foreach block B_i
    calculate the content hash H_i of B_i
    calculate key K_i from content hash
    encrypt B_i using K_i
    calculate hash of ciphertext CH_i
    store encrypted block using CH_i as identifier
end foreach

foreach block B_i
    await acknowledgement for B_i end foreach

create an inode using previous version as template
store all H_i and CH_i in blocklist of inode
encrypt inode using key from directory
calculate contents hash of inode H
store inode using H as identifier
await acknowledgement for H
send update-request to node responsible for directory

if the responsible node agrees upon the new name/inode list it
returns an acknowledgment. Otherwise, if some other
directory-operation had been performed, the holder of the directory
returns the latest version of the name/inode-list and the procedure is
repeated until the nodes can agree.
```

When a new datablock arrives at a node it replicates the block to its successors in the frontlist. All nodes that receive a replicate-message sends an acknowledgment to the source of that datablock. These acknowledgments are signed by the node that keeps the replica in order to prove that it is a legitimate participant in the system.

All data is kept at the node that tries to store the data until sufficient acknowledgments have arrived.

The reason for this scheme is that some nodes might be malicious and throw away data that they supposed to store.

5.5.5 Reading data

The table below describes the alogrithm for reading data from a file.

```
translate name to inode using directory
(the directory will check permissions)
fetch inode
retrieve key from directory
decrypt inode
foreach block i from set of blocks the user wants to read from
fetch block using CH_i
verify ciphertext using CH_i
decrypt block using H_i
verify cleartext using H_i
return data
```

5.6 Dealing with storage space

5.6.1 Quota

An administrator of a distributed file system expects to be able to place limitations on the amount of storage space used in a certain part of the file system or the amount of space used by a certain user or group of users. This is a much harder problem in a completely decentralized file system such as Keso than in a traditional file system, since no node has complete knowledge of all the data stored by a single user.

One way to handle quotas is to let each node limit the amount of data stored by a certain user. The limit for each node can for instance be a portion of the total amount of storage space available to that user relative to the portion of the total address space assigned to this node.

The benefit of this system is that it is fully decentralized. There is no need for communication when data is inserted in the system. The disadvantage is that it is only a local quota. A user may be hindered to store a file even though there is quota space available, just because the user had bad luck and filled up the quota on a certain node.

Another approach is to keep a quota object in the system for each entity where the quota is to be restricted. This object is kept as a normal object in the DHT and is consulted each time an entry is added or removed. The main benefit of this is that the quota now is a global quota imposed on the whole system. The disadvantage is that it is more complex and it adds the need to consult other nodes when doing file system operations.

5.6.2 Reclaiming storage

When a file has been marked as deleted the file content is still stored in the Keso system. Old file versions should probably be purged occassionally, along with data which is no longer in use. Some work on determining which versions to purge has been done with the Elephant file system [17] and the main idea is to keep landmark versions - in other words versions that are considered interesting. These could for instance be the last version from a rapid series of updates.

Removal of the blocks in Keso provides a few difficulties. Blocks could be referenced from several different inodes and creating a list of the blocks that currently should be in the system requires knowledge of the complete state of the system at the moment, which is difficult to acquire.

We see four different strategies that could be used:

- all blocks have a reference counter that is incremented each time a block is stored. A cleaner process then walks through the file system and decrements the reference counters for blocks belonging to versions that become deleted.
- directories ocassionally decide which versions to keep and send update requests for all blocks belonging to the files that are supposed to be kept. This would be similar to the CFS system [3], except for the system issuing the update-requests automatically.
- calculate global state and delete blocks no longer part of the system. This is a very expensive operation.
- not purging data at all. Since storage space is reused this might not be as expensive as it seems, but this would have to be evaluated.

Chapter 6

Implementation

The implementation of Keso consists of three separate modules; DKS, the LocalStore and Keso.

The DKS part implements the DKS overlay network as described in Chapter 4. The LocalStore is responsible for handling insert- and lookup-requests similar to the DHash (Section 3.3.1) in CFS[3] and Ivy[4]. The LocalStore also contains logic to verify the update-requests for directories. Keso is built on top of these two modules and implements the actual file system and the overall design is thus very similar to the Ivy[4] and CFS[3] systems.

Keso currently support file creation, modification and retrieval. There is currently no kernel support, but it is prepared for being implemented using NNPFS from the Arla[28] project.

6.1 DKS

DKS is designed in an object oriented and very modular fashion. The individual modules are described below and their relations are shown in figure 6.1. Each DKS node is implemented as a separate object which can be instantiated and used by an application. The API used when interfacing with the DKS object is shown in table 6.1. Our implementation of DKS does not at the moment support the notion of virtual servers. The SHA-1 hash of the nodes IP-address is currently used as its identifier.

Our DKS implementation currently supports node join and leave, item lookup and insertion. The implementation also uses the correction on use mechanism defined

Creation of a new DKS object	DKS(N, k, f , nodeid, LocalStore, ipaddress of this node, ipaddress of a remote node);
Data insertion	void put(Identifier , Datum &);
Address lookup (blocks until a reply has been recieved)	incu::SockaddrIn getAddr(Identifier &);
Data lookup (blocks until a reply has been recieved)	Datum *getData(Identifier &);
Stoping the DKS system	void shutDown();
Printing status of the DKS node	void show();

Table 6.1: The DKS API

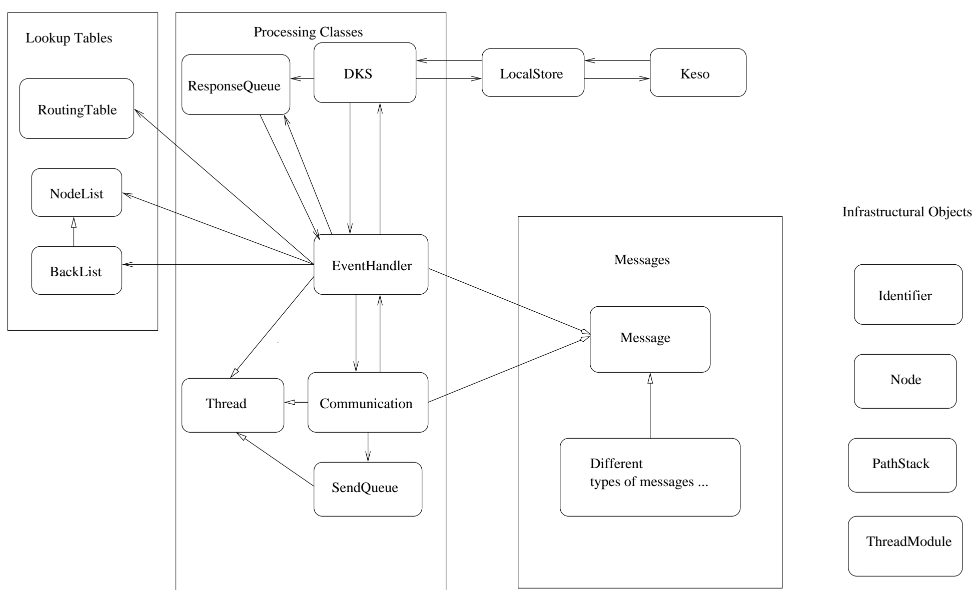

Figure 6.1: The main modules in the DKS implementation.

for DKS. The implementation currently does not support replication of data across multiple nodes. We also do not detect if a node fails. This is something which was not a part of the version of the DKS specification which we have used for this project.

6.1.1 Message marshalling

Message marshalling and unmarshalling has been solved via the use of iostream methods implemented in all classes that are supposed to be sent on the network. It was quite a cumbersome solution, but we decided to implement all marshalling ourselves since we could not find any suitable marshalling modules for C++.

6.1.2 The EventHandler class

The EventHandler class is the heart of the DKS system and handles all the logic related to message passing. It mainly contains the different event handlers that correspond to the different types of DKS messages.

6.1.3 The Communication class

The communication class is responsible for sending and receiving messages and it uses an object oriented communication package called Incu [33] which has been written by Rasmus Kaj. Currently it opens new TCP connections for all messages, which is non-optimal since it might result in messages arriving at their destination out of order (DKS assumes that the communication channel will deliver messages in the order that they were sent).

6.1.4 The RoutingTable and the Node/BackList classes

These classes implement the front lists, back lists and the routing table in DKS. They contain the necessary methods for adapting to new information and find suitable nodes for the message routing. They also contain methods for marshalling and unmarshalling, since both routing tables and node lists are sent on the network.

Begin an operation. Returns the identifier of an operation.	int sendOperation(Operation *op)
Retrieves the result of an operation	Operation * getReply(int operation)
Transfer all items between start and stop to node destination	void transfer_state(Identifier start, Identifier stop, Identifier destination)
Initialize the localstore.	void init()
Get the number of items in the localstore.	int size()

Table 6.2: The LocalStore API

6.1.5 Threads and other helper classes

C++ does not have any suitable built-in support for object oriented threads. For this reason, we have implemented our own threads package. It is based upon pthreads and contains a thread-class, semaphores and mutexes.

6.2 Localstore

The LocalStore module implements the distributed storage. It is responsible for keeping the data which the local node is responsible for and to transfer it to other nodes when necessary. It interacts with DKS through the DKS API in table 6.1.

The LocalStore module has no knowledge about the internal structure of the file system. All this knowledge is in the Keso module, and calls are made from LocalStore to Keso when specific knowledge is required. One example of such a case is when files are added to directories. In order to assure consistency of the directory, this is done on the storing node.

6.3 Keso

Keso implements the file system logic. It uses the distributed hash table functionality of DKS as its storage, and is unaware of the actual location of the data.

User interaction with Keso is implemented as a command line interface. There is no interaction with the kernel in the current implementation. This is not a huge drawback, since the design is prepared for the kernel.

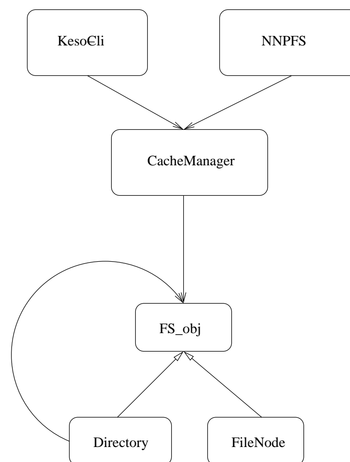

Figure 6.2: The main modules in the Keso implementation.

6.3.1 Interaction between Keso and LocalStore

Keso instantiates a LocalStore object and tunnels all its request through the LocalStore object down into DKS.

When Keso wants to perform an operation it will create an Operation message which contains all the necessary information to complete the operation. This is then handed to LocalStore which returns an Operation ID to Keso. This enables Keso to do other work while LocalStore completes the operation. Keso may for instance insert requests for several different pieces of data before starting to retrieve the first piece. This ability is important when retrieving files, since it makes it possible to prefetch data.

The current implementation of Keso includes most of the basic file system design. It is possible to setup a network of nodes. A file system structure with directories and files can be setup on top of this network. None of the security and access control parts are currently implemented.

6.3.2 Keso code

Keso consists of three main modules. This is shown in figure 6.2 The file system object handles implementation of the specifics for each type of file system object. Currently, only files and directories are supported. We have defined a base class which is called FS_obj which contains all the basic properties of file system objects, which is inherited by each specific file system object class.

A file is a list of data blocks identified by their content hashes, some metadata such as the owner of the file, the creation date and the last date the file was modified. The file also includes a signature of the person who wrote the file to the file system. It is the responsibility of the file object to handle splitting of files into blocks, computing the hashes, maintaining the blocklist and metadata, and all other file related operations.

A directory is a mapping service from names to lists of file system objects. Each list contains all the versions of a file in the file system. The directory object implements all the directory operations, such as insertions, deletions and updates of files and subdirectories.

The cache-manager is responsible for implementing the file system semantics against the upper layer, such as the kernel or the command line interface. It tunnels most of the operations to the underlying file and directory object.

Chapter 7

Statistics

During the work with this masters thesis we have collected a number of statistics from the distributed file system at the IT-Department (IT-enheten) at KTH. These statistics include the amount of data available on the local harddrives on workstations, distribution of files with respect to file sizes and the amount of data used grouped by file sizes. These statistics are presented below and a summary is given in Table 7.1.

7.1 Description of the system

The system where we have made these measurements is an AFS distributed file system at the IT-Department (IT-enheten) at KTH. The system has about 5000 users and there is a total of around 400GB of data stored. About 200GB of this data is stored in users home directories. There are around 10^7 files stored in the file system.

The user base is mainly technical students in electrical engineering and researchers in various technical areas such as mathematics, electrical engineering and computer science. The file system has been in production use since 1992.

7.2 Free space

We began by measuring the amount of free space available on the harddrives of workstations in the computer system. We measured this over a total of 104 workstations. The machines had a total of 3.5 TB of harddrive space and 1.75 TB of this was unused. In other words, 50% of the harddrive space of local machines was unused.

These results are quite similar to other investigations [25].

7.3 File sizes

We measured the sizes of all files stored in the file system. The average file size is 40 kilobytes and the median file size is 2 kilobytes.

First, we show the number of files in relation to the size of the files. This is shown in figure 7.1. This figure shows that the vast majority of files are relatively small, while there are few files of larger sizes.

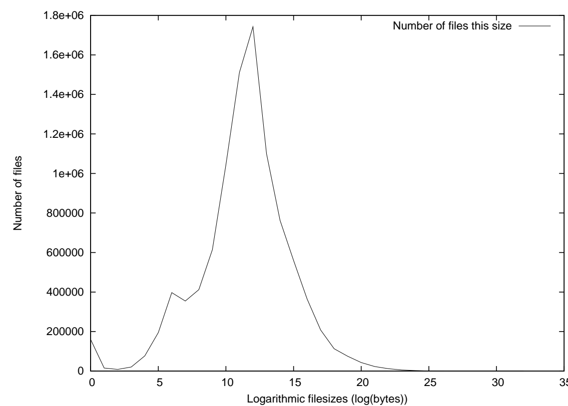

Figure 7.1: Distribution of files according to file sizes

Figure 7.2 shows the amount of harddrive space used for files within each size range. The quirk around 2^{30} bytes is because a number of cd-images of software distributions are stored in the file system, and the maximum size of a cd-image is 650MB.

The combination of these graphs show that most of the data in the file system is stored in a few large files, while the majority of the files are rather small. This is also reflected in the mean and average file sizes.

Finally, figure 7.3 shows the integral of the first graph. This shows the number of files which are as small or smaller than a certain size. This shows that somewhere around 2^{17} to 2^{19}, the graph levels out and we have covered most of the files. Keso is intended to split files into blocks of equal size. A choice of 256k blocks means that 88% of the files will be stored in one single file.

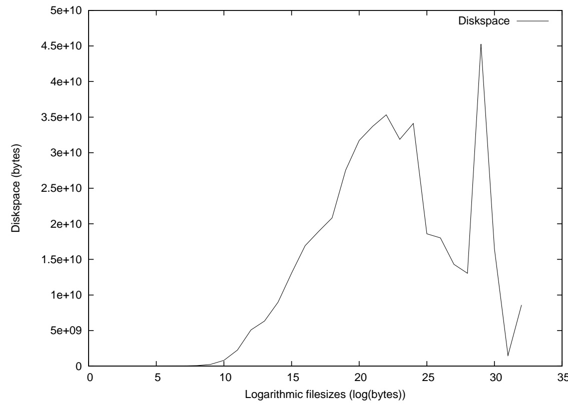

Figure 7.2: Amount of data stored in files of different sizes

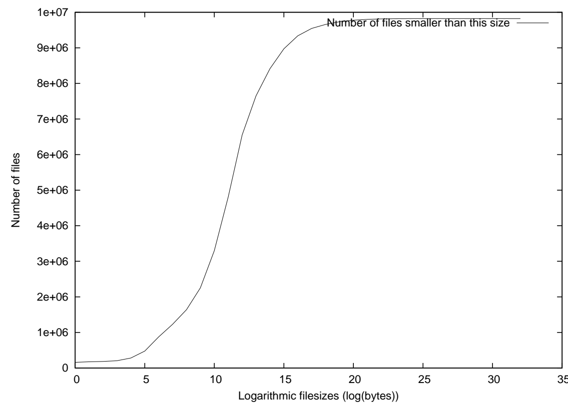

Figure 7.3: Number of files which are smaller than a certain size

7.4 Hashes of datablocks

We have computed SHA1 [26] hashes of each 256K block from each file in the file system. We have then compared these hashes and looked for duplicates.

We then compared the actual data in theses duplicate blocks and found no colliding hashes. This was expected since SHA1 is a collision resistant hash function and there are no known collisions. For this reason we can consider two file system blocks equal if they hash to the same value.

There were a total of $1.1 * 10^7$ blocks in the file system. $6.2 * 10^6$ of these were unique. $1.2 * 10^6$ of the hashes had at least one duplicate. The total amount of data in the file system was 473GB at the time. The amount of total data in unique

Number of users	approx. 5000
Total amount of read/write data	approx. 400 GB
Amount of data in user home directories	approx. 200GB
Number of files	approx. 10^7
Average file size	40Kb
Median file size	2Kb
Percent redundant data	24%
Number of workstations	104
Total disk space on workstations	3,5 TB
Free disk space on workstations	1.75 TB
Percentage of unused disk space on workstations	50%

Table 7.1: A summary of the measurements

blocks were 361GB. This means that 24% of all the data stored is redundant data. An earlier investigation [25] has shown that about 50% harddrive space is redundant. The difference between our investigation and theirs is that they measured redundant data on local harddrives. This means that their measurements includes the local operating system, and possibly local copies of the same files which would only be one copy in a distributed file system.

7.5 Errors

The measurement of file sizes was conducted over one week during the fall of 2003. This was a period of quite low activity in the computer system since students had their examination period. Most of the measurements were also done during the weekend when the staff were off duty. Changes to the file system during the snapshot period may have upset the statistics somewhat. This measurement is fairly resilient to errors due to file system changes since it only measures each file by itself, and not groups of files. It is therefore safe to presume that the measurements were not affected by changes to the file system.

The block hashing measurement was conducted over a four month period between December and March. This means that there were quite a lot of changes to the file system during this time. This was particularly apparent when we tried to compare the actual file system blocks of colliding hashes. Around 2% of the underlying file system blocks had been altered since their hashes had been computed. This has probably not affected the final result to any large extent.

Chapter 8

Conclusions

8.1 Main achievements

We have presented the design of Keso, a distributed and completely decentralized file system based on the peer-to-peer overlay network DKS. In this thesis we have shown how to design a scalable and fault-tolerant read/write file system on top of the distributed hash table service. We have also shown how to provide access control and security in such a system while avoiding to unnecessarily store redundant data.

The statistics that we have collected from the IT department at KTH indicates that large amounts of resources are wasted in traditional environments with file servers and workstations. It is therefore reasonable to assume that large organizations would benefit from a file system based on the ideas from Keso.

8.1.1 The project

We began this project by reading research papers related to this area. This included both papers relating to peer-to-peer structures in general and papers on distributed file systems with a focus on peer-to-peer file systems.

Thereafter we made a study of the existing distributed file system at the IT-Department at KTH. This made us realize that as much as 24% of all data in the distributed file system was duplicated data. We also discovered that 50% of the disk space of the workstations at the department was unused. This amounted to a total of 1.7 TB data, or three times as much as the total amount of data stored in the distributed file system.

We took these insights to the design phase and concluded that two of the main design goals of Keso should be that it should avoid storing unnecessarily redundant data and that it should make use of the hard drives of local workstations. After the design phase we started implementing the overlay infrastructure DKS and finally we made a partial implementation of the file system.

8.1.2 Design

The Keso file system is designed make use of unutilized resources, be highly scalable, self-organizing and fault-tolerant. It is also well suited for a real world environment, with support for access control and data privacy.

Keso is built using an underlying peer-to-peer infrastructure. A peer-to-peer system is potentially much more resilient to system malfunction than centralized computer systems. The computers are distributed over a large physical area and data is distributed and replicated over these computers. Thus a number of computers have to fail in order for the data to become unavailable. This makes Keso very resilient to failures. The use of a peer-to-peer infrastructure also provides for high scalability and self-organization.

Since we wanted to avoid unnecessarily storing redundant data we have designed Keso to only store each unique file system block once. This is made possible since we identify disk blocks by their contents, and then reference these disk blocks from each file with the same contents.

Rethinking security in order to trust users instead of traditional servers made goals such as access control and data privacy possible. Versioning has shown to serve two quite different purposes. Keeping old versions in the system is both a way of providing a good service to the user and a way of handling concurrent updates without locking.

The main drawback is that a completely decentralized system becomes quite complex, especially when you must handle the possibility of cooperating malicious participants.

8.1.3 Implementation

The implementation of Keso is a partial proof-of-concept implementation. It includes all basic file system functionality except for kernel interaction. Most of the special features, such as security and access control, version handling and replication has not been implemented. The implementation has been done in C++.

The structured overlay network DKS has been implemented from the specification

draft we received at the beginning of the project. The specifications did not include replication or handling of failed nodes, and we have therefore not implemented these parts of DKS.

8.2 Evaluation

The evaluation done is mainly a theoretical analysis of whether Keso meets its design objectives or not. When the Keso implementation is more complete, a full performance analysis should be done as well.

8.3 Future work

A complete implementation is needed in order to understand how the system will behave under real usage. The current security model depends heavily on cryptography and might be quite expensive. In order to be comparable to other distributed file systems, the file system must interact with the kernel. This is probably best done via the NNPFS kernel module.

A formal security analysis is needed in order to prove the security of the encryption and signature scheme.

Quota is a difficult problem in decentralized file systems. This needs to be addressed in the future.

The different replication schemes should be evaluated and the corresponding cost for adding a group agreement protocol in order to remove the serialization point should be analyzed.

When the underlying network is partitioned a file may be updated in both partitions. This will create multiple version lists. The issue of merging these lists must be addressed.

Handling of failed nodes and replication has to be added to the DKS implementation.

8.4 Conclusion

We believe that Distributed Hash Tables are a good basis for developing complex, scalable and fault-tolerant decentralized systems. We have shown that a file system built on top of untrusted components can provide reliable and secure services to users and administrators alike. The design of Keso is promising and we believe that future research in the area definitely is motivated.

Bibliography

[1] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications, ACM SIGCOMM 2001, San Deigo, CA, August 2001, pp. 149-160.

[2] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems". IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, pages 329-350, November, 2001.

[3] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica, Wide-area cooperative storage with CFS, ACM SOSP 2001, Banff, October 2001.

[4] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A Read/Write Peer-to-peer File System. To appear in Fifth Symposium on Operating Systems Design and Implementation (OSDI). Boston, MA. December 2002.

[5] The Institute of Electrical and Electronics Engineers, Inc. POSIX: IEEE Standard Portable Operating System Interface for Computer Environments, 1988. IEEE Std 1003.1-1988.

[6] M. Satyanarayanan, "Scalable, Secure, and Highly Available Distributed File Access," IEEE Computer, pp. 9–20, May 1990.

[7] J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal and F. Smith, "Andrew: A Distributed Personal Computing Environment", Communications of the ACM, 29(3) pp184-201, March 1986.

[8] J. G. Steiner, B. C. Neumann and J. I. Schiller, Kerberos : An Authentication Service for Open Network Systems, Proc. Winter USENIX Conf., pp.191-201, 1988.

[9] D. Mazieres. A toolkit for user-level file systems. In Proceedings of the USENIX Technical Conference, pages 261–274. USENIX, June 2001.

[10] K. Fu, M. F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. Computer Systems, 20(1): 1-24, 2002.

[11] I. B. Damgaard. A design principle for hash functions Advances in Cryptology, CRYPTO89, LNCS 435, pp 416-42.

[12] B. Schneier, Applied Cryptography, Wiley, New York (1996).

[13] Luc Onana Alima, Sameh El-Ansary, Per Brand, and Seif Haridi, Dks(n; k; f): A family of low communication, scalable and fault-tolerant infrastructures for p2p applications, Proceedings from International workshop on Global and Peer-ToPeer Computing on large scale distributed systems (Tokyo, Japan), May 2003.

[14] I. Clark, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In Proc. of the Workshop on Design Issues in Anonymity and Unobservability, Berkeley, CA, http://freenetproject.org/freenet.pdf, 2000.

[15] Tanenbaum, A. S. Modern Operating Systems. Prentice- Hall, 1992.

[16] D. Brodsky, J. Pomkoski, S. Gong, A. Brodsky, M. Feeley, N. Hutchinson (2002). Mammoth: A Peer-to-Peer File System. University of British Columbia, Department of Computer Science (TR-2003-11).

[17] Douglas J. Santry, Michael J. Feeley, and Norman C. Hutchinson. Elephant: the file system that never forgets. Hot Topics in Operating Systems (Rio Rico, AZ, 29–30 March 1992.

[18] G. Ganger and Y. Part. Metadata Update Performance in File Systems. In Proc. of USENIX Symposium on Operating System Design and Implementation, pages 49-60, November 1994.

[19] Chow, R., Johnson, T.: Distributed Operating Systems and Algorithms. Addison-Wesley, Reading, MA (1997).

[20] R. Sandber, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network file system. In Proceedings of USENIX Summer Conference, 1985.

[21] Microsoft Corporation. Microsoft Networks SMB File Sharing Protocol (Document Version 6.0p). Redmond, Washington.

[22] Marshall K. Mckusick, Keith Bostic, Michael J. Karles, and John S. Quarterman. "The Design and Implementation of the 4.4 BSD Operating System". Addision-Wesley, 1996.

[23] Tor Klingberg and Raphael Manfredi, draft of the ”The Gnutella 0.6” rfc, http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html, 2002

[24] Ian Clarke, “Freenet’s Next Generation Routing Protocol”, http://freenet.sourceforge.net/index.php?page=ngrouting, 2003

[25] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In SIGMETRICS, 2000.

[26] National Institute of Standards and Technology. Secure Hash Standard. Technical report, FIPS 180-2, Washington, D.C., August 2002.

[27] National Institute of Standards and Technology. Advanced Encryption Standard. Technical report, FIPS 197-1, Washington, D.C., November 2001.

[28] A. Westerlund, J. Danielsson. Arla - A free AFS Client. In Proceedings of the USENIX Technical Conference 1998. USENIX, June 1998.

[29] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI), Boston, MA, Dec. 2002. USENIX.

[30] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In Proceedings of the Third Symposium on Operating Systems Design and Implementation, pages 173–186, New Orleans, Louisiana, February 1999. USENIX Association.

[31] J.R Douceur, A. Adya, W.J. Bolosky, D. Simon, M. Theimer, “Reclaiming Space from Duplicate Files in a Serverless Distributed File System,” in ICDCS, 2002

[32] J. R. Douceur, A. Adya; J. Benaloh; W. J. Bolosky; G. Yuval, “A Secure Directory Service based on Exclusive Encryption”, 18th ACSAC, Dec 2002.

[33] Rasmus Kaj. The incu library. http://www.stacken.kth.se/ kaj/incu-0.7.tar.gz

[34] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, Ion Stoica Load Balancing in Structured P2P Systems. Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS’03)

[35] F. Cristian, "Understanding Fault-Tolerant Systems", Communications of the ACM, Vol. 34, No. 2, Feb 1991, pp. 56-78.