



PEPITO
IST-2001-33234
PEer-to-Peer Implementation and TheOry

Deliverable no: D2.3

Report on the distributed directory service

REPORT VERSION: first

REPORT PREPARATION DATE: 2004.12.31

CLASSIFICATION: Public

DELIVERABLE NO: D2.3 DUE DATE: Month 36 DELIVERY DATE: Month 36

PROJECT START DATE: 2002.01.01 PROJECT DURATION: 36 months

RESPONSIBLE PARTNER: KTH

PARTICIPATING PARTNERS: KTH,SICS

PROJECT COORDINATOR: Swedish Institute of Computer Science AB

PROJECT PARTNERS: EPFL Lausanne, INRIA Paris, KTH Stockholm, UCL Louvain, University of Cambridge UK



**Project funded by the European Community under the
'Information Society Technologies' Programme (1998–
2002)**

Project Number: IST-2001-33234
Project Acronym: PEPITO
Title: PEer-to-Peer Implementation and TheOry
Deliverable No: D2.3
Report on the distributed directory service
Due date: project month 36

Responsible Partner: KTH
Participating Partners: SICS, KTH

13th January 2005

Contents

1	Introduction	2
1.1	Overview	2
1.2	Outline	3
2	Related work	3
2.1	Structured overlay networks	3
2.2	Storage systems	3
2.3	File Systems	3
3	Keso	4
3.1	Semantics	4
3.2	Assumptions	4
3.3	File system measurements	5
3.4	Overview	5
3.5	Directories	6
3.6	Files	7
3.7	File versioning and conflict resolution	7
3.7.1	File versions	7
3.7.2	File deletion	8
3.7.3	Network Partitioning and Disconnected Mode	8
3.8	Storing data	8
3.8.1	Replication	8
3.8.2	Caching	8
3.8.3	Acknowledgments	9
3.9	Dealing with storage space	9
3.9.1	Quota	9
3.9.2	Reclaiming storage	9
3.10	Security	10
3.10.1	Access control	10
3.10.2	Data privacy	11
3.10.3	Tamper protection	12
3.10.4	Storing of files - description	13
3.10.5	Reading data	13
3.10.6	Groups	14
4	Implementation	14
5	Summary	14
5.1	Future work	15

1 Introduction

This report describes Keso, a read/write peer-to-peer file system built on top of the DKS overlay network. Keso is intended for large scale usage and it is completely decentralized - there are no dedicated servers. Instead Keso is intended to run on ordinary work stations. It is designed in order to provide all the features expected from a real world distributed file system such as AFS, for example security, access control, location independence and administrative delegation. Keso is also created in order to be self organizing in the sense that no human intervention should be necessary in order to recover from failures or assure that data is layered out evenly over the distributed nodes.

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 four 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.

There are thus several reasons to choose to design a new distributed file system on peer-to-peer technologies. Some of the most prominent ones are scalability, fault tolerance and the ability to make use of hitherto unused resources. Several projects have therefore been focusing on developing peer-to-peer file systems and storage systems. The main contribution from Keso is that it shows how to design a read/write peer-to-peer file system that provides security and access control on top of a distributed hash table.

1.1 Overview

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 datablock 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 datablocks 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.

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 performs all operations in time logarithmic to the number of nodes in 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.

1.2 Outline

The rest of this report regards the design and implementation of Keso. The first Section 2 related work is referred. In Section 3 the design of Keso is presented, including semantics, how data is stored and security. Section 4 gives a brief overview of the implementation. The last Section 5 contains a summary, conclusions and future work.

2 Related work

2.1 Structured overlay networks

Keso is built on top of the Distributed Hash Table (DHT) provided by the DKS overlay network. There are several other flavors of structured peer-to-peer overlay networks, such as Chord [17] and Pastry [11]

Chord [17] is developed at the Massachusetts Institute of Technology. Chord provides support for just one operation: given a key, it maps the key onto a node and scales logarithmically with the number of Chord nodes. DKS has a ring structure that is similar to that of Chord.

Pastry[11] is a location and routing scheme that uses a prefix based routing algorithm. Each node keeps a routing-table with entries that has identifiers that share an increasing number of digits. While constructing these routing-tables network locality is taken into account.

2.2 Storage systems

On top of these overlay networks several different storage systems have been developed [3] [12].

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. It is built on top of the DHash block storage system and the Chord[17] overlay network. CFS splits files into datablocks and uses content hashes as keys in a way that has influenced the Keso file system.

Past [12] is a whole file storage system that uses the Pastry location and routing scheme.

2.3 File Systems

Ivy [8] is a read-write peer-to-peer file system intended for small scale usage. It is built on top of the DHash block storage system also used in CFS. It is based on the idea that each participant keeps its own log and all file system operations performed by that node is stored into the log. In order to get a view of the current state of the file system a participating node extracts the information from all logs that it trusts.

Farsite[1] is a federated peer-to-peer file system intended for large company and university environments. Groups of nodes form a distributed directory server that implements Byzantine failure protocols. It has an encryption scheme for data privacy and access control that is similar to that of Keso.

Elephant [6] 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).

AFS [7] is a widely used distributed file system. All the file system data is stored on file servers and the clients keep locally cached copies of the files. In order to perform the required operations the clients have to prove to the file server that they have the right to do so. In order to keep the local caches up to date the file server notifies the clients that keep local copies of a file before allowing updates.

Coda[14] is a continuation of the AFS project and it introduced the MiniCache[16]. Most of the client side functionality is placed in a user-level process for the reasons that such processes are easier to develop and debug. A small kernel module interacts with the virtual file system switch in the kernel and it also keeps a small cache for performance reasons.

The idea with the Coda MiniCache has been reused in Arla [18], a free AFS client developed mainly by students at the Royal Institute of Technology. Keso uses the kernel module from Arla.

3 Keso

3.1 Semantics

The Keso semantics is somewhat similar to the AFS semantics. The main difference is that Keso is a versioning file system, that creates new, immutable files for every file update. There are several reasons for this scheme. The main one is that the most common reason for user data being lost is user mistakes [13]. Using a versioning file system makes it easy for users to get back and retrieve the lost data. Other reasons for making Keso versioning includes that it is thus easy to implement atomic file system operations and that conflicts due to network partitioning easily can be resolved by using a heuristics in order to decide what version to be used.

Some of the semantical similarities between Keso and AFS include global namespace, location independence and the absence of locking.

3.2 Assumptions

Keso is intended for large scale usage, for instance companies and universities. It is also intended for being 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.

The organization that uses the file system is expected to have the means to install necessary certificates in a secure manner and that only machines that are expected to behave nicely should be let in. This way machine availability and reliability is expected to be higher than on the Internet, but lower than for traditional servers. The participating nodes cannot however be trusted in the same way as traditional file servers, so Keso should still perform the requested services even though a configurable minority of the clients misbehave.

Since Keso is used by an organization most of the machines are expected to be on the same LAN with high throughput and low latency.

Keso is not intended for databases and scientific or other applications that require high performance I/O. Databases requires locking, which is not a part of the Keso design.

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 do, however, expect our system to perform reasonably well.

3.3 File system measurements

We did a minor study of the environment at the IT-department (IT-enheten) at The Royal Institute of Technology in Stockholm, Sweden. There have been quite a large number of measurements on file sizes and file usage in file systems [?][1]. The main goal of this study was to find out how much data in the distributed file system that was redundant, such as the same files with different names, and how much storage capacity that was unused on the work stations. As a secondary effect we also collected statistics on average file sizes and the characteristics of the files that contained the largest amounts of data.

In the measurements a simple scheme for detecting redundant data blocks was used. All files (directories and symbolic links were ignored) in the AFS cell were split into 2Bk large blocks and the SHA-1 hash [10] was calculated for all these blocks. Afterward all the hashes were compared and the redundancy percentage calculated. This shows that with a simple scheme like this 24% of the datablocks would be considered as redundant.

The total amount of data in the AFS cell was 400GB and the unused storage on the workstations was 1.75Tb. In other words there was enough storage capacity on the workstations to remove the file servers without adding extra storage space to the system.

Most of the files are very small, which is completely in line with previous results [?]. However, most of the data is stored in a few, very large files. These files are often of the type that is rarely modified, for instance ISO-images and mp3-movies. In other words, most of the file system operations are related to small files that make up a quite small percentage of the amount of data stored in the system. It is thus reasonable to optimize the file system for small files.

Previous studies [?] has shown that distributed file system accesses shows locality in both time and space. In other words the file system needs to have a support for having fast, local caches of the files accessed. Coda [14] and Arla [7] [18] have a method for achieving this with local file caches that are installed into the kernel of the client node.

3.4 Overview

Keso is built on top of the DHT provided by the DKS system. Each data item to be stored into the DHT is assigned a key and there is at every given moment only one node that is responsible for each key. In Keso the DHT is used for storing two different types of data, files and directories. Files are split into datablocks and inodes and the inodes contain a block list and the files metadata. Both datablocks and inodes are stored as content hash blocks, in other words a hash of their contents is used as a key. Directories on the other hand are assigned a key at the creation of the directory and that key remains the same throughout the directories lifetime, even though the contents may change.

Each participating node can be said to have three different roles. The most important part for the system as whole is that it must participate in the message routing at the DKS layer. The second role is the part of the Keso protocol that other nodes are depending on. This consists of storing and replying to requests for content hash blocks and directories. For content hash blocks, inodes and data blocks, this is fairly straight forward. If a request for storing a datablock arrives at a node the node must store this datablock after it has verified that the request is issued by a legitimate user of the system and that the key matches the contents. For directories this is more complicated. The request must not only be issued by a legitimate user, that user must also be permitted to make the required operation. Finally, a call back mechanism similar to that of AFS is used in Keso, so all nodes that have a local cache of the directory must be notified.

The last role for a Keso node is interacting with the user. This includes fetching files and direc-

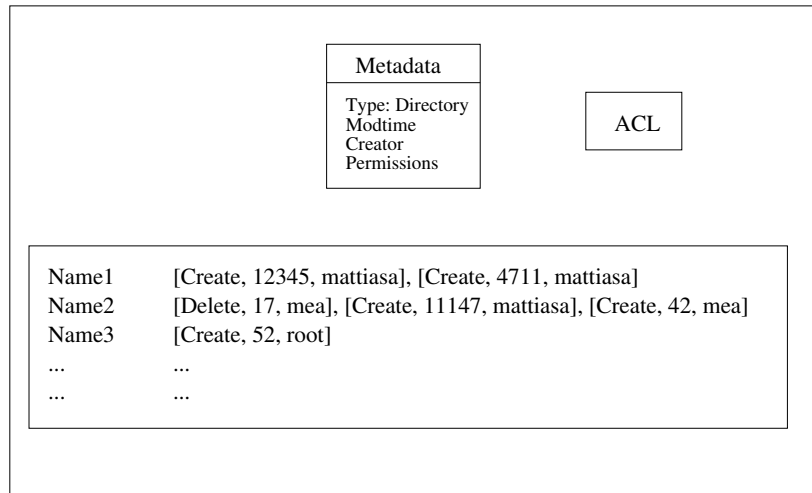


Figure 1: Layout of directory. It includes metadata, the list of file versions and a signature

ories that the user wishes to read, keeping a local cache and interacting with the operating system in an appropriate way.

3.5 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 1.

The directory metadata is similar to that of files, with modification time, which user 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 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 delete entry in the source directory, decrypting and reencrypting the inode and adding a new create entry containing the new inode number in the new source directory.

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.

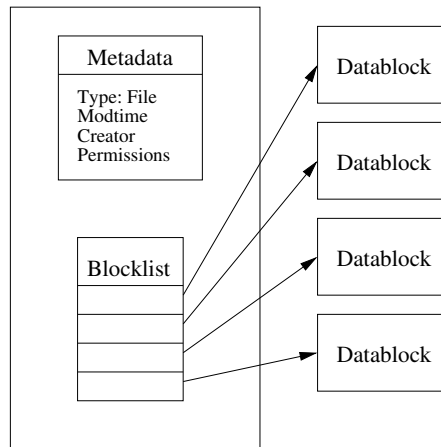


Figure 2: An inode, with metadata and blocklist

3.6 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 2.

The metadata contains data about the file itself, such as the time this version of the file was created, the list of UNIX permission bits 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.

3.7 File versioning and conflict resolution

3.7.1 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.

3.7.2 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.

Storage space is currently not reclaimed in Keso. A few strategies for how this can be achieved is discussed in section 3.9.2.

3.7.3 Network Partitioning and Disconnected Mode

After a network partition the version lists for each file are merged. If there are conflicts a 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. This is not handled in the current implementation of Keso.

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.

3.8 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 from Ivy [8] and CFS [3].

3.8.1 Replication

Data stored at a node is replicated by the DKS layer. The scheme is 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.

3.8.2 Caching

There are two different types of caching that could be used in the Keso system. Client node caching means that a node working with a file or directory will keep a private copy of that file or directory. Block caching can occur at the DKS layer, letting nodes cache data items with identifiers that are not within the range of that node.

Client node caching is used in the Keso implementation and a call back mechanism is used for directories. When a node fetches a directory it is added to a call back list on the server node. The server node then tells the node with a local cached copy of that directory if the directory is updated within a certain time frame.

Block caching is uncomplicated for datablocks and inodes, but more complex for directories. 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, since they have the same location but change their content. This makes it necessary to invalidate the cache copies as the directory changes. No block caching is implemented in the Keso system.

3.8.3 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 should be signed with the nodes private keys.

3.9 Dealing with storage space

3.9.1 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 storage 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.

3.9.2 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 occasionally, 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 [13] 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 and decremented each time a block is deleted.
- ▷ directories occasionally 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.

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 1: Keso rights

- ▷ 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.

3.10 Security

Security in Keso is based on three primitives:

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

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 [15]

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.

No parts of the security scheme described below are implemented.

3.10.1 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 1.

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

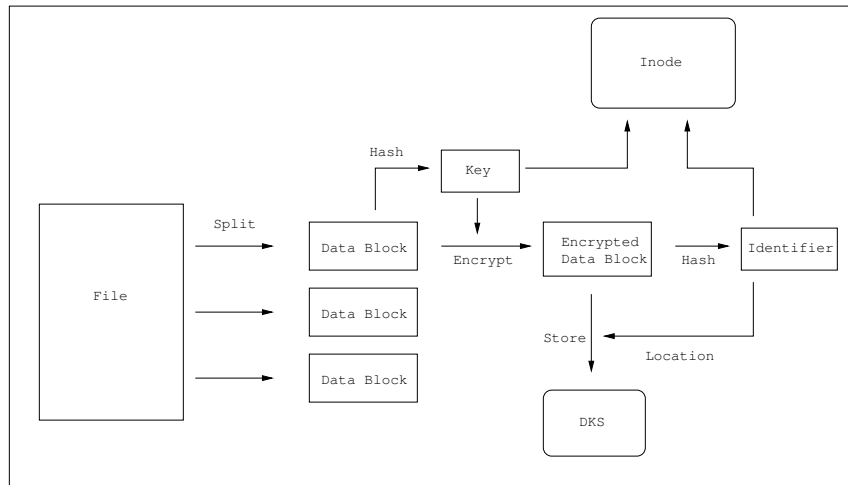


Figure 3: The scheme for storing a file.

knowledge about the private keys associated with the public keys in the ACL will be able to decrypt and read the actual data and make updates to the directory that can be verified as correct.

3.10.2 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 [1] [4] and an overview of it is shown in Figure 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 [9]. Another hash is then computed of the encrypted block, and the block is saved into the DHT using this hash as the key.

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.

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 [5]. 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.

3.10.3 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 through 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 through 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 writer's 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 is 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 asking all the replicas for the latest version and comparing the replies. Both of these strategies are however expensive.

When a directory is created it is important to make sure that it is possible for a user to verify that the directory is indeed the directory that the user is looking for. In order to achieve this the newly created directory should be signed together with the name of the directory and the identifier of the parent directory by the creator/owner of the directory.

The last strategy will become expensive as soon as a directory starts to contain several files and file versions. It should thus be possible to verify that a version of a directory contains correct information without having to verify that the complete chain is correct.

If the initial signature is included in the signature of a new entry it is possible to verify that this is indeed a version of files created for this directory by the user that has made the change. Verifying the complete chain can this way be left for something to do for server nodes and when a user wants to go back in time and view older versions.

3.10.4 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  $H$ 
do
  calculate signature
  send KESO_ADDVERSION(name,inode,signature) to node responsible for directory
  await acknowledgement
  if update_failed
    fetch latest version of directory
  until update succeeds

```

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.

3.10.5 Reading data

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

```
translate name to inode using directory
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
```

3.10.6 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.

4 Implementation

The Keso implementation has complete read/write support and supports all basic file system operations. No parts related to security, fault tolerance and replication is however implemented. It is written in C++ and consists of three separate modules that correspond to the three different roles a node in the Keso system has. DKS routing is handled by a DKS module implemented in C++.

The LocalStore handles requests for datablocks and directories. Finally, the Keso layer handles the user interaction. It depends on a kernel module called NNPFs for interaction with the VFS on the local machine. NNPFs is a part of the Arla project [18] and makes it possible to implement the file system logic in an userland daemon. NNPFs keeps a local cache of the file installed in the kernel, thus significantly speeding up several of the file system operations. [14]

5 Summary

In this paper Keso, a read/write peer-to-peer file system, was presented. Keso is indented for large scale usage in for example university or large companies. The usage of the distributed hash table provided from the DKS system provides Keso with an efficient basis for routing and block storage.

File data and inodes are stored as content hash blocks in order to make it easy to verify data integrity. Files are immutable and every time a file is updated a new version is created. Directories are treated entirely different and they keep the structure of the file system and they contain a list of all file names and the corresponding inodes. A directory is not stored with a content hash - it keeps the same key during the whole lifetime of that directory.

Security and access control is provided by a combination of symmetric and asymmetric cryptography. datablocks are encrypted with their own contents. Inodes are encrypted with the encryption key belonging to the directory that the file is stored in. The encryption key for the directory is in turn encrypted with the public keys of all users and groups that are supposed to be allowed to access the file. All updates to a directory is signed by the user that did the update, so that the contents of the directory can be verified.

5.1 Future work

In the current design the contents of directories is open for anyone to read. This should be corrected.

The implementation has to be completed in order for a performance test to be possible.

The way that directory contents is protected from being exchanged for the contents of another directory should be analyzed in order to ensure that such attack is impossible. Over all, a formal security analysis should be done.

Quota and similar problems should be solved.

The idea of introducing a Byzantine-fault-tolerant protocol involving all the replicas of a data item should be explored.

References

- [1] 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.
- [2] 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.
- [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] 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
- [5] J. R. Douceur, A. Adya; J. Benaloh; W. J. Bolosky; G. Yuval, "A Secure Directory Service based on Exclusive Encryption", 18th ACSAC, Dec 2002.
- [6] 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).
- [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] Athicha Muthitachoen, 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.
- [9] National Institute of Standards and Technology. Advanced Encryption Standard. Technical report, FIPS 197-1, Washington, D.C., November 2001.
- [10] National Institute of Standards and Technology. Secure Hash Standard. Technical report, FIPS 180-2, Washington, D.C., August 2002
- [11] 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.
- [12] A. Rowstron and P. Druschel. "Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility", ACM SOSP, October 2001.
- [13] 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).
- [14] Satyanarayanan, M., Kistler, J.J., Siegel, E.H., Coda: A Resilient Distributed File System IEEE Workshop on Workstation Operating Systems, Nov. 1987, Cambridge, MA
- [15] B. Schneier, Applied Cryptography, Wiley, New York (1996).

- [16] David C. Steere, James J. Kistler and M. Satyanarayanan, Efficient User-Level File Cache Management on the Sun Vnode Interface Proceedings of the 1990 Summer USENIX Conference June 1990, Anaheim, CA
- [17] 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.
- [18] A. Westerlund, J. Danielsson. Arla - A free AFS Client. In Proceedings of the USENIX Technical Conference 1998. USENIX, June 1998.