

Keso

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

Johanna Svenningsson



Keso

- ◆ What?
 - ◆ A read/write peer-to-peer file system
- ◆ Why?
 - ◆ Fault tolerance
 - ◆ Scalability
 - ◆ Make use of unused resources

Outline

- ◆ Overview
- ◆ Related work
- ◆ Measurements from the IT department
- ◆ Fundamentals
- ◆ Security in Keso
- ◆ Accessing data
- ◆ Implementation
- ◆ Conclusion

Overview

What is Keso?

- ◆ Keso is a distributed file system built on a peer-to-peer infrastructure.
 - ◆ Completely decentralized
 - ◆ Scalable
 - ◆ Secure
 - ◆ Self-organizing
 - ◆ DHT
 - ◆ Versioning
 - ◆ AFS as an usage model

Overview

How Keso works

- ◆ Runs on workstations
- ◆ Files split into blocks and distributed over the participating nodes
- ◆ Uses a combination of symmetric and asymmetric encryption
- ◆ Data blocks and directories are replicated to f nodes to provide redundancy
- ◆ Built on top of the DKS overlay network

Related work

Storage systems:

- ◆ CFS
- ◆ Past
- ◆ OceanStore

Related work

File systems:

- ◆ Ivy
- ◆ FarSite
- ◆ Mammoth

Related work

What makes Keso interesting?

- Security and access control on top of a DHT.
- Theoretically scalable.

Measurements

Measurements taken at the IT-department (IT-Enheden) at KTH. There are approximately 500 users and the system consists of 104 workstation. 400GB data is stored in the AFS cell.

Measurements:

- ◆ File sizes
- ◆ Usage of workstations
- ◆ Redundant data

Measurements

File sizes

- ◆ 10^7 files
- ◆ Average size: 40kb
- ◆ Median size: 2kb

Most of the files are very small and most of the data is stored in a few, very large files. The large files are mainly ISO images.

Measurements

Usage of workstations:

Results:

- ◆ 50% of local hard drives unused on workstations
- ◆ 1.75 Tb free disk space

4.3 times as much free disk on workstations as was stored in the distributed file system!

Measurements

Redundant data

- ◆ Files split into 2kb blocks
- ◆ SHA-1 hash calculated for all blocks
- ◆ All hashes compared

With this method 24% of the data in the AFS cell was unnecessarily redundant, mainly the same file stored with different names.

Measurements

Conclusions

- ◆ There is a lot of unused storage capacity to be taken advantage of.
- ◆ Even a simple scheme for avoiding to unnecessarily store redundant data will have a large effect.
- ◆ Most of the file updates are related to very small files and most of the data is in a few, very large files that are rarely modified. It is thus reasonable to optimize the file system for this.

Overview

Semantics

- ◆ Immutable files
- ◆ Global namespace
- ◆ Location independence
- ◆ No locking

Overview

Assumptions

- ◆ A file system belongs to an organization
- ◆ Academic and corporate environments
 - ◆ Hi bandwidth, low latency
 - ◆ Machine availability higher than on the internet, lower than traditional servers
- ◆ Not for databases
- ◆ Not for scientific calculations or other applications in need of high performance I/O

Keso

Overview

- ◆ Built on top of DKS
- ◆ Old versions of files kept in the file system
- ◆ Data encrypted using its content in a way that avoid storing unnecessarily redundant data

Keso

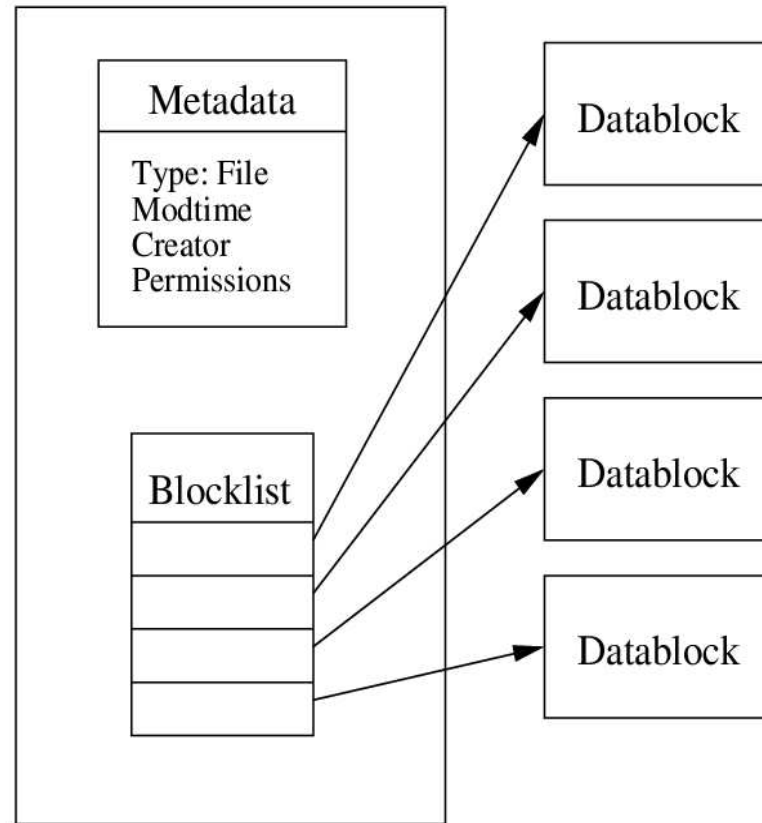
Overview

- ◆ File data and nodes
- ◆ Directories
- ◆ A node that is responsible for a directory is responsible for all updates concerning files in that directory.

Keso

Overview of files

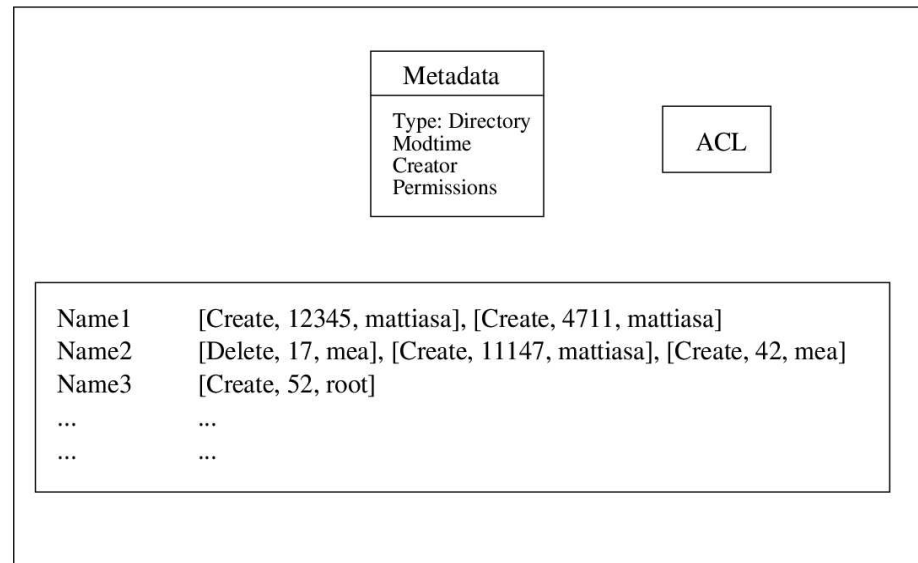
- Data is split into blocks of equal size
- Blocks are referenced from a block list in the inode
- Both blocks and inodes are stored in DKS using a hash of their contents
- All files which contain the same data reference the same blocks



Keso

Overview directories

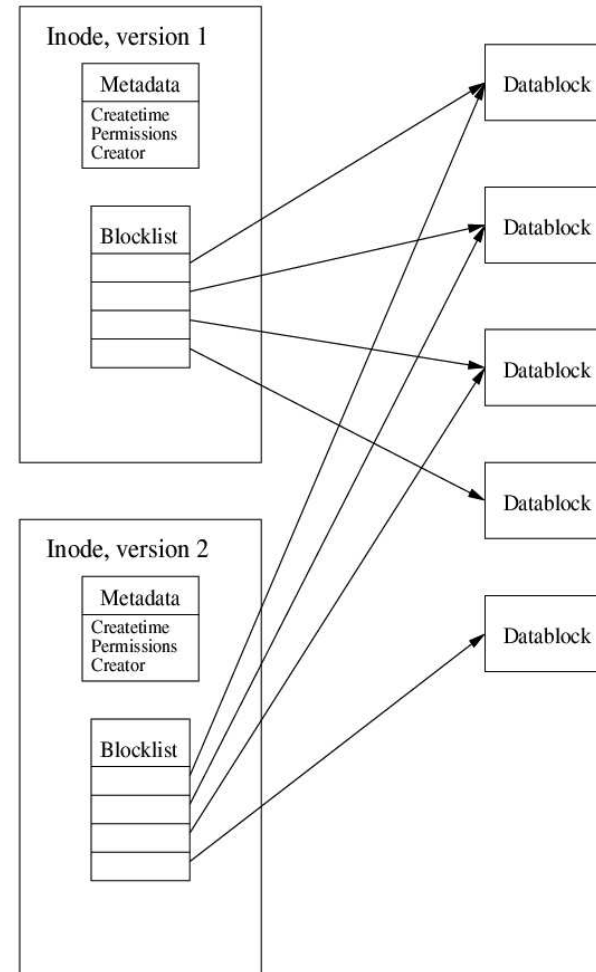
- Acts as a name/inode lookup service
- The identifier of a directory never change
- Should be generated from users key, dirname/parent/other such that the directory cannot be exchanged



Keso

Versioning

- All versions of files are kept
- Users can go back through a file's history
- Directories contain a list of file versions
- Only blocks which are changed must be stored additionally



Security in Keso

- Access control
- Data privacy
- Tamper protection

Security in Keso

Access control

- ◆ PKI – each user and node has a public/private key pair
- ◆ Each directory has a symmetric key used for protecting data in that directory
- ◆ The symmetric key for a directory is encrypted with the public keys of all users and groups permitted to access files in that directory

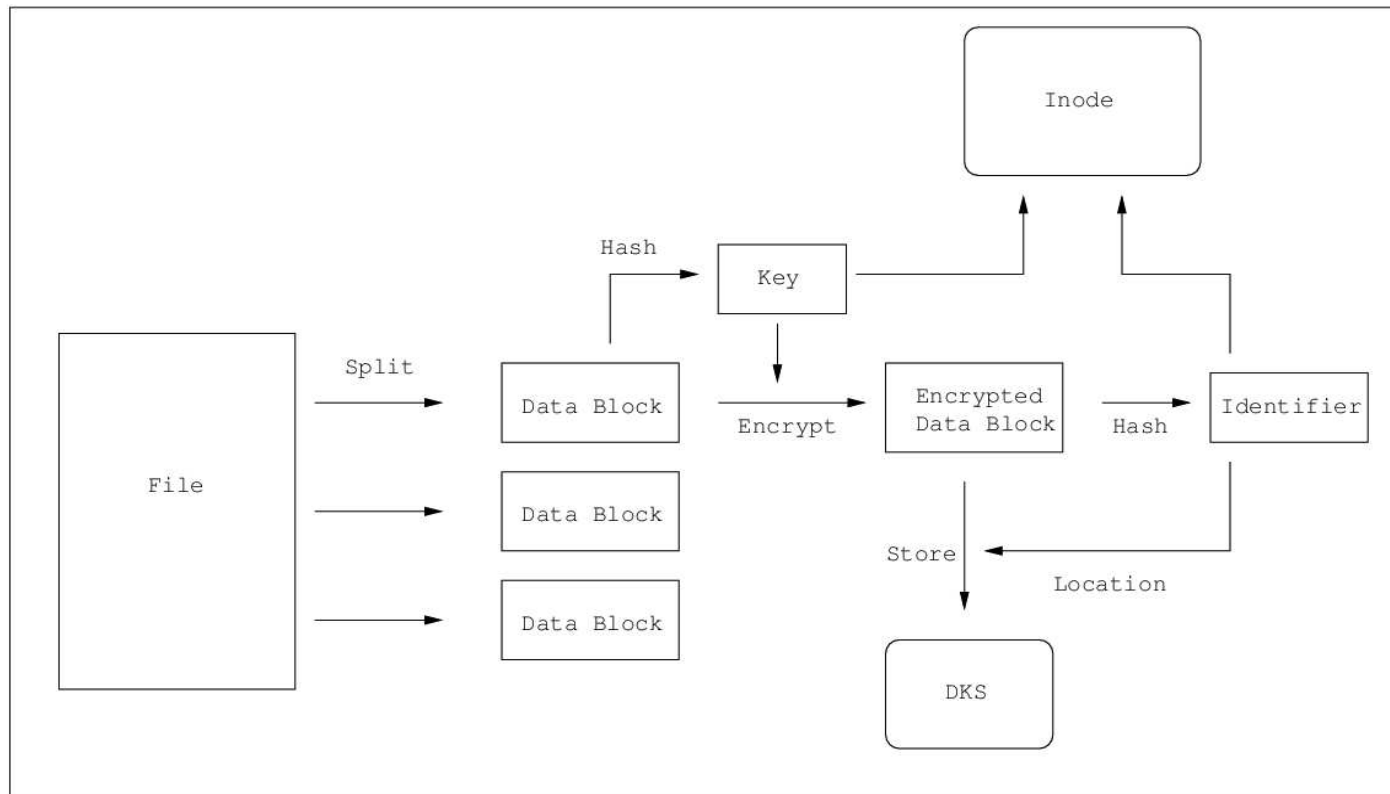
Security in Keso

Data privacy

- ◆ Each file is encrypted using its own content hash
- ◆ The encrypted block is stored in DKS using the content hash of the cipher text
- ◆ Both the hash of the clear text and cipher text blocks are stored in the inode
- ◆ The inode is finally encrypted with the symmetric directory key.

Security in Keso

Data privacy



Security in Keso

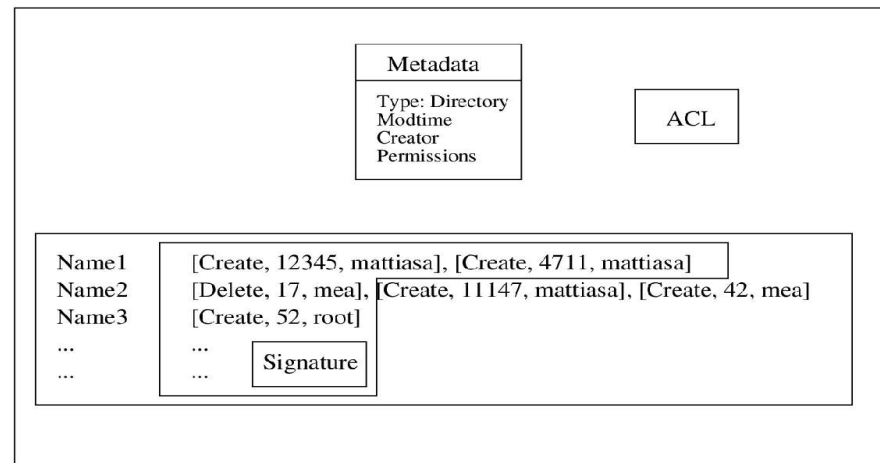
Data privacy

- ◆ Advantages
 - ◆ Multiple blocks with the same content are only stored once
- ◆ Disadvantages
 - ◆ The existence of a block with a specific content can be verified
 - ◆ Possibly expensive

Security in Keso

Tamper protection

- Data blocks and inodes are stored using the hashes of their contents
- When changes are committed to the directory, the entire latest version and the change is signed
- This makes sure that changes can be tracked through time.



Keso

Storing data

- ◆ Data is replicated on a number of nodes using the replication scheme of DKS
- ◆ When nodes store data they send acknowledgments to the "client" node. The "client" node waits until enough nodes have acknowledged that they have saved the data.

Caching

- ◆ In the DHT
 - ◆ Uncomplicated for content hash blocks
 - ◆ Difficult for directories
- ◆ On the client node
 - ◆ Necessary for both files and directories
 - ◆ Call backs!

Call backs

- An idea from AFS for keeping local caches up to date
- When a directory is fetched from a node the node responsible for that key is added to the call-backs list
- If the directory is updated within a given period the node will tell all other nodes that have fetched this directory
- Currently contains both the directory identifier and the name of the file/dir that has been modified

Accessing data

Mounting Keso

- The root directory has a well known key
- Directory corresponding to this key is fetched
- Node added to the callback-list on the node responsible for the root directory
- Directory cached locally and installed into the kernel
- All nodes in the system wants to be notified if the root directory is updated. Same in all DFS and rare, but can become expensive in Keso if no broadcast mechanism exists.

Accessing data

Reading a file

- The directory containing the file is fetched and the fetching node is added to the call backs list.
- The identifier of the files' inode is looked up from the directory
- The inode is fetched
- The required data blocks are fetched

Accessing data

Writing to a file

- All data blocks are encrypted and stored
- The inode is stored and encrypted
- An update request together with a signature is sent to the node that is responsible for the directory
- If the update request is correct the responsible node updates the directory
- Call backs are broken to all nodes that have fetched the directory. The file name is sent along with these callbacks.

Implementation

- ◆ Three separate modules

- ◆ DKS

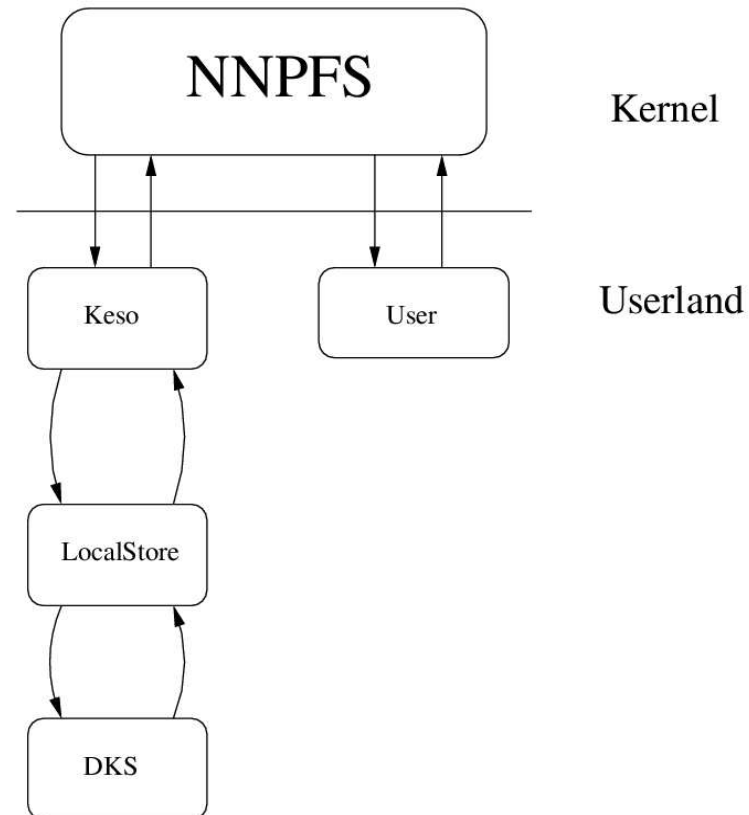
- Communication

- ◆ LocalStore

- Storing data

- ◆ Keso

- Knowledge about the file system structure



Implementation

Made in C++

- Supports all basic file system operations – read, write, delete, mkdir, rmdir...
- No access control
- DKS implementation does not provide replication and detection of failures

Conclusion

Main achievements

- ◆ Design and implementation of a read/write file system on top of a DHT avoiding operations that are $O(\text{Nodes})$.
- ◆ Provide access control, data privacy and tamper protection while avoiding unnecessarily storing redundant data.
- ◆ Collected statistics which show that our design is reasonable in the real world.