MaidSafe Distributed File System

David Irvine, email: david.irvine@maidsafe.net, LifeStuff: David

maidsafe.net limited (registered in Scotland Sc 297540)

September, 2010 ***Abstract*—Distributed file systems require servers or control nodes. Access to a file system is a security issue that can apparently only be controlled by some kind of authority, and this is always a potential point of failure. These file systems also require an indexing mechanism. This paper presents a distributed file system without centralised control or indexing. This file system also utilises a distributed locking mechanism to ensure data integrity in the case of multi-user access to any file.**

***Index Terms*—security, freedom, privacy, file systems**

Contents

I. Introduction

FILE systems are a relatively new and slow-changing part of computing. There are differences between operating systems (and even versions of operating systems) on how they handle access to data via a file system. This has proved to be problematic over the years and has led to many short-term fixes. Systems such as CIFS, Andrews, SMB and many others appear to be an answer, but for many reasons (occasionally political) they lose ground again. It is the intention of this paper to present a universal file system that implements a minimum set of features that will operate cross platform.

This system will represent itself to a user as a native file system on any platform, and as such requires low level drivers and code to be installed alongside any application using it.

A significant advance in distributed locking is employed which allows shared directories to be easily set up and maintained.

In addition, there is a solution to the problem of location of data, or path sizes, which is limited on every operating system. This paper presents a mechanism to overcome this and allow for an almost infinite number of levels of directory structure to be implemented.

A. Conventions Used

There is scope for confusion when using the term “key”, as sometimes it refers to a cryptographic key, and at other times it is in respect to the key of a DHT “key, value” pair. In order to avoid confusion, cryptographic private and public keys will be referred to as $\mathsf{K}_{\mathsf{priv}}$ and $\mathsf{K}_{\mathsf{pub}}$ respectively, and DHT keys simply as keys.

- Node $\equiv$ a network resource which is a process, sometimes referred to as a vault in other papers. This is the computer program that maintains the network and on its own is not very special. It is in collaboration that this Node becomes part of a very complex, sophisticated and efficient network.
- H $\equiv$ Hash function such as SHA, MD5, etc.
- $\mathsf{XXX}_{\mathsf{priv}}$, $\mathsf{XXX}_{\mathsf{pub}}$ $\equiv$ Private and public keys respectively of cryptographic key pair named XXX
- $\mathsf{SymEnc}_{[\mathsf{PASS}]}(\mathsf{Data})$ $\equiv$ Symmetrically encrypt Data using PASS
- $\mathsf{Sig}_{[\mathsf{K}_{\mathsf{priv}}]}(\mathsf{Data})$ $\equiv$ Create asymmetric signature of Data using $\mathsf{K}_{\mathsf{priv}}$
- $+$ $\equiv$ Concatenation
- $\bigoplus$ $\equiv$ Bitwise Exclusive Or (XOR)
- STORE $\equiv$ Network or other key addressable storage system
- $\mathsf{PutV}_{[\mathsf{Key}]}(\mathsf{Value})$ $\equiv$ Store Value under Key on STORE. This value is signed.

B. Security of Data

In order to achieve data security, nodes requesting a $\mathsf{PutV}_{[\mathsf{Key}]}(\mathsf{Value})$ *must* sign both Value and the request itself. STORE must only allow subsequent changes to Value if the request and replacement Value are similarly signed.

II. Overview

To enable a huge amount of data to be organised into a directory structure (as we are used to) then a new method of data management is needed. In this case, we have created a system of divorcing the structure of any data from a tree in one direction.

In this paper we present a system where a directory structure may be traversed forward from any point and with efficient implementation back to that point, but no further back unless new knowledge of the structure is gained. This has the effect of allowing directory trees to be free forming rather than tied to any root or base level. In this system, a user can have a tree, but it is a free forming tree and not a distributed overall root directory.

III. Distributed Directories

A. General

Every directory has a unique identifier associated with it. Part of this identifier is used as a key under which its encrypted contents[1] are held on STORE. The contents are encrypted (and hence can only be decrypted) by using the identifiers of the directory and the directory's parent. In this way, a given directory's contents (and hence files and subdirectories) can be recursively decrypted to yield all the levels of subdirectories it may contain. However, without knowledge of the identifier of the directory's parent, the parent cannot be decrypted.

B. Creation Process

The process to create a subdirectory (Child) of a directory (Parent which has identifier ParentID) is as follows:

1) Find a random key (ChildKey) which is unused on STORE
2) Derive the Child's identifier (ChildID) from ChildKey (e.g. by appending random data)
3) Add a new entry[2] which includes ChildID in Parent to represent Child
4) Encrypt Parent and Child (yielding ParentEnc and ChildEnc respectively) as described below
5) $\mathsf{PutV}_{[\mathsf{ParentKey}]}(\mathsf{ParentEnc})$
6) $\mathsf{PutV}_{[\mathsf{ChildKey}]}(\mathsf{ChildEnc})$

C. Encryption of Directory Entries

This uses a process very similar to that described in *Self Encrypting Data* [1]. The process is as follows:

1) Self-encrypt the Child directory as any other file in [1]. This yields a datamap (ChildDM)
2) Where $\mathsf{H}(\mathsf{ChildID} + \mathsf{ParentID})$ is named Obf, create a data chunk (ObfChunk) which is the same size as ChildDM by repeatedly rehashing Obf and appending the result, i.e. $\mathsf{Obf} + \mathsf{H}(\mathsf{Obf}) + \mathsf{H}(\mathsf{H}(\mathsf{Obf})) + ...$
3) Create an obfuscated datamap, $\mathsf{ObfDM} = \mathsf{ChildDM} \bigoplus \mathsf{ObfChunk}$
4) Create a symmetric encryption passphrase, $\mathsf{Pass} = \mathsf{H}(\mathsf{ParentID} + \mathsf{ChildID})$
5) Finally, create the encrypted datamap, $\mathsf{EncDM} = \mathsf{SymEnc}_{[\mathsf{Pass}]}(\mathsf{ObfDM})$

D. Advantages of Distributed Directories

- We can have an almost infinite traversal of directories with no limit on depth
- To share data, all that is required is access to a decrypted directory[3] and then all the directories below that are automatically shared
- Directories following a distributed paradigm are more suited for mass distribution

E. Data Locks

If the STORE is a file system or database, then data locks are a standard feature. Here we discuss locking in a Distributed Hash Table (DHT) which is problematic. This section assumes a DHT of a similar capability as *MaidSafe Distributed Hash Table [2].*

To ensure writes to data are atomic, we require a locking mechanism that is solid and allows the network to recover from stale locks, which tend to be an issue. In a MaidSafe DHT this can be achieved quite simply and is efficient due to the speed of the network, via managed connections.

To write data, a node requires to request a lock from the $\mathcal{K}$ closest nodes to the data (this requires that no dead nodes exist in the DHT). On receiving a lock, each node will confer with the other nodes; if all accept the lock, then it is in place. If there is any collision of two separate requests for a lock, both are rejected. In this case the requesting nodes will back-off for a random time and try again.

On receiving a lock, a node will read the data again to confirm it is the same version that has been updated and will then update the value.

There should be a system wide lock duration *constant* in place that will remove any locks that have gone stale (as they will).

Lock requests will be signed by the sender to allow the $\mathcal{K}$ recipients to confirm the permissions of the requester. If the signature validation fails, the requests are silently dropped.

F. Private Shared Directory Structures

To enable private shared directory structures, two issues need to be addressed; what to use in place of ParentID for the shared root directory, and how to allow permitted peers access to the decrypted shared root directory.

[1]"contents" here does *not* mean the files listed in the directory, but rather the structure used to represent the list (e.g. a database)

[2]The entry may also contain all required metadata of Child

[3]or an encrypted directory along with the keys required for decryption

1) Creating a Shared Root Directory: Creating a shared root directory is as per III-B above with the exception of the encryption element. The node creating the shared root directory cannot allow peers to know the $\mathsf{ParentID}$ for the directory (for security reasons). A replacement is derived, and the encryption phase is carried out as per III-C. Creating and storing the replacement for $\mathsf{ParentID}$ is done as follows[4]:

1) Create a key pair $\mathsf{ShareOwn_{priv}}$ and $\mathsf{ShareOwn_{pub}}$
2) Create an ID for the share owner, $\mathsf{ShareOwnID} = \mathsf{H}(\mathsf{ShareOwn_{pub}} + \mathsf{Sig}_{[\mathsf{ShareOwn_{priv}}]}(\mathsf{ShareOwn_{pub}}))$
3) $\mathsf{PutV}_{[\mathsf{ShareOwnID}]}(\mathsf{ShareOwn_{pub}} + \mathsf{Sig}_{[\mathsf{ShareOwn_{priv}}]}(\mathsf{ShareOwn_{pub}}))$
4) Create a key pair $\mathsf{Share_{priv}}$ and $\mathsf{Share_{pub}}$
5) Create an ID for the share, $\mathsf{ShareID} = \mathsf{H}(\mathsf{Share_{pub}} + \mathsf{Sig}_{[\mathsf{ShareOwn_{priv}}]}(\mathsf{Share_{pub}}))$
6) $\mathsf{PutV}_{[\mathsf{ShareID}]}(\mathsf{Share_{pub}} + \mathsf{Sig}_{[\mathsf{ShareOwn_{priv}}]}(\mathsf{Share_{pub}}))$
7) $\mathsf{H}(\mathsf{Share_{pub}})$ is now used as the replacement for $\mathsf{ParentID}$

Normally a node storing an encrypted datamap EncDM of a private non-shared directory would sign the $\mathsf{PutV}_{[\mathsf{DirectoryKey}]}(\mathsf{EncDM})$ request and data with a cryptographic private key ($\mathsf{K_{priv}}$) known only to itself. A single $\mathsf{K_{priv}}$ could be used for all directories, regardless of whether they are in the same tree or not.

However, in the case of a shared directory, peers must be able to modify it, i.e. they must be able to sign modified datamaps and requests with the original private key. For this reason, the $\mathsf{Share_{priv}}$ is used when storing an encrypted datamap of a shared directory. The same $\mathsf{Share_{priv}}$ is used for all subdirectories of the shared root. However each new shared root directory must have a unique $\mathsf{Share_{priv}}$ to allow peer permissions to be assigned on a “per-shared-directory” basis.

2) Getting Access to a Private Shared Directory: The creator of a private shared root directory allocates permissions to selected peers and sends each a signed, encrypted message on successful creation of the directory. All of the peers receive the share name (human-readable as assigned by the creator), $\mathsf{ShareID}$, $\mathsf{Share_{pub}}$, and $\mathsf{ChildKey}$. This allows retrieval of the encrypted datamap (from STORE held under $\mathsf{ChildKey}$) and its decryption (using $\mathsf{H}(\mathsf{Share_{pub}})$ and $\mathsf{ChildKey}$), i.e. read access to the directory and its subdirectories.

For peers given write access, the message also contains $\mathsf{Share_{priv}}$ to enable them to modify the encrypted datamap held on STORE.

For peers given administrator access, the message also contains $\mathsf{ShareOwn_{priv}}$ to enable them to alter $\mathsf{ShareIDs}$, remove users and delete the share completely from the STORE.

3) Revoking Access to a Private Shared Directory: To revoke a peer's access to a shared directory, an administrator creates a new $\mathsf{ShareID}$ and keypair. Then the administrator locks the root of the share, copies the content to the new root and creates a new $\mathsf{ChildKey}$. A message is sent to all authorised peers (minus the peer having its access revoked) as in the previous section. When a peer receives such a message, it must start re-reading the share from the new root.

The administrator then copies all the existing directory structures (locking each in turn) to the new structure, starting at the new root and deleting the old directories recursively.

A user's system will note this action and if a file is opened, it will wait for the new directory (and datamap) to become available if the current directory is locked, otherwise it is safe to store the file as the recursive 'move' has not reached that point yet.

[4]Steps 1 to 6 describe creating a self-signed identity packet as detailed in *"Peer to Peer" Public Key Infrastructure*[3]

G. *Public Shared Directory Structures*

Each user may also create public shared directory structures; i.e. ones which can be accessed (read-only) by any peer. The process is simpler than the one for private shared directories in that the datamaps need not be encrypted, and there is no need to send a message to a group of peers.

The creator of a public shared directory uses a different cryptographic private key (named $\mathsf{MPID_{priv}}$) to sign the datamap and requests. $\mathsf{MPID_{priv}}$ is not revealed to any peer and can be used for all public shared directories, regardless of whether they are in the same tree or not. This way anyone can see the data but only the creator can edit it, requiring no locks of course.

In order to allow peers to find public shared directories, users should have a publicly-known ID (similar to the concept of an email address). For a node whose public ID is $\mathsf{PublicID}$, the datamap of its root public shared directory should be stored under $\mathsf{H}(\mathsf{PublicID})$.

In this way, any user can publish information and it can be read by any peer who merely knows the user's public ID. Data will be passed around, and browser addons can be created to allow widespread access to data on public shared directories.

H. *Anonymous Shared Directory Structures*

In future, users will also be able to create anonymous shared directory structures. These will be similar to public shared ones, but the keys under which datamaps are stored and any signing keys will not be traceable back to the user concerned.

IV. Conclusions

This paper has introduced a method of storing data in a distributed network in a manner that is addressable, searchable and very scalable. It is apparent that such systems could in fact supplement or replace paradigms used by the existing world wide web for data sharing. It is not difficult to see that applications that make use of massively shared data and data presented on a native format to users is an exciting proposition.

References

[1] David Irvine, Self Encrypting Data, david.irvine@maidsafe.net
[2] David Irvine, MaidSafe Distributed Hash Table, david.irvine@maidsafe.net
[3] David Irvine, "Peer to Peer" Public Key Infrastructure, david.irvine@maidsafe.net

David Irvine is a Scottish Engineer and innovator who has spent the last 12 years researching ways to make computers function in a more efficient manner.

He is an Inventor listed on more than 20 patent submissions and was Designer / Project Manager of one of the World's largest private networks (Saudi Aramco, over $300M). He is an experienced Project Manager and has been involved in start up businesses since 1995 and has provided business consultancy to corporates and SMEs in many sectors.

He has presented technology at Google (Seattle), British Computer Society (Christmas Lecture) and many others.

He has spent many years as a lifeboat Helmsman and is a keen sailor when time permits.