maidsafe: A new networking paradigm

David Irvine*
MaidSafe.net, 72 Templehill, Troon, South Ayrshire, Scotland, UK. KA10 6BE.
*david.irvine@maidsafe.net

First published April 2009.

Abstract—**This paper presents a new way of networking and data handling globally. This data centric network is likely to revolutionise the IT industry in a very positive fashion. For the first time you will be able to approach any computer and like saying 'you be mine', log in, and it is as though it becomes your computer, with all your data, desktop etc. but without leaving any trace upon logout.**

Index Terms—**security, freedom, privacy, DHT, encryption**

CONTENTS

LIST OF TABLES

LIST OF ALGORITHMS

NOMENCLATURE

ANMAID Anonymous MAID keypair. This keypair is never used for anything other than sign the MAID packet and delete it again if necessary. The private key is never transmitted or used in any other situation. This is a safe keypair in that it's used with extreme caution and therefor extremely unlikely to be stolen or otherwise comprimised.

ANMPID Anonymous MPID keypair. This keypair is never used for anything other than to sign the MPID packet and delete it again if necessary. The private key is never transmitted or used in any other situation. This is a safe keypair in that it's used with extreme caution and therefor extremely unlikely to be stolen or otherwise comprimised.

ANTMID Anonymous TMID keypair. This keypair is never used for anything other than sign the TMID packet and delete it again if necessary. The private key is never transmitted or used in any other situation. This is a safe keypair in that it's used with extreme caution and therefor extremely unlikely to be stolen or otherwise comprimised.

DNS Domain Name System, used to convert IP addresses (such as 88.222.43.12) into memorable names such as maidsafe.net, google.com etc. and vice versa

ID IDentification or IDentifier. In maidsafe these terms are used to describe several indentifiers, including anaonymous, system level and public.

MAID [maidsafe Anonymous IDentification] An anonymous <key,value> pair used to store data on the network privately and also used to control the PD Vault.

MID A packet used to identify a unique user upon login, derived from their username and PIN (personal identification number).

MMID [Moving maidsafe IDentification] A version 2 MID packet which continually alters its name to further mask the user's ID.

MPID [maidsafe Public IDentification]. A special packet which creates a user identity that is public. This packet is not linked with any other in the system, i.e. there is no link between this and login or data storage and retrieval. It can be considered to be a publicly viewable name (e.g. printed on a business card) that gives no clues to the user's system credentials.

MSID [maidsafe Shared IDentification]. A private share data map held locally and updated on the network via messages.

MSMID [Moving Surrogate maidsafe IDentification] A version 2 SMID which continually alters its name, making guessing a username pin combination statistically very improbable, and with today's technology, mathematically impossible.

PKI Public Key Infrastructure, typically using RSA to create a key pair. The public key is given to anyone and used to encrypt information that is only decrypted by the private key. The public key can confirm the private key of the ID has signed data. A server is usually employed to tie the ID with the public key, however, this is where errors can happen as keys may be replaced on the server, subverting the whole mechanism.

PMID [Proxy maidsafe IDentification]. A packet named with the hash of the vault's public key, signed by the owners MAID and which is linked to the MAID. By linking the vault and the client, it allows the client to store data directly by signing requests with this PMID key. The PMID keypair is created by the client.

PPMID [Purchasable Proxy Maidsafe IDentification]. A packet used to maintain a list of disk space which may be allocated by maidsafe to others either at a cost or by donation, etc. This is used in the pdzone registration process.

RSA Rivest Shamir Adleman asymmetric encryption

SMID [Surrogate maidsafe IDentification] A surrogate MID packet used to ensure MID integrity during an update (write after update).

TMID [Temporary maidsafe IDentification]. An ever changing chunk (at every session update) which holds the data map of the data atlas (i.e. a database of all stored files).

I. INTRODUCTION

NETWORKShave grown in popularity over recent years. This phenomenon has now spread to all walks of life with the advent of the Internet. An issue with today's systems is that they require centralised controls and data structures. This paper describes a method of distributing data in a controlled, non-owned grid. The main elements of the system are:

- anonymous or self authentication

- perpetual data
- globally distributed Public Key Infrastructure (PKI)[1]
- self encryption
- environmentally friendly computing
- enhanced ID theft prevention
- data corruption prevention

maidsafe can be considered a platform rather than a single application and on this platform there is the ability for many applications to exist. The significant difference from traditional networks and particularly storage area networks is that in a maidsafe network people are self policed, network controlled, self created and completely unknown. There are publicly known IDs that can be created by users, however, these are separated from their login or data management IDs used elsewhere in the network.

The initial response to the findings of this system are varied and range from impossible all the way to this is amazing. A point to note is that the systems represented by this paper are at present very computationally intensive in areas, require significant bandwidth and may appear to be unsustainable, however initial testing proves this not to be the case substantively, although it should also be noted we are in a state of Moore's Law growth in CPU and possibly faster than this with broadband bandwidth and hard drive capacities. This area of growth has been exponential over the last decade and the author is very aware that the further down a period of exponential growth then each passing day brings significantly more CPU, bandwidth and disk space than the previous day did. As we are nearing the end (if there can be one) of these particular exponential growth curves the daily increase *tends towards* infinity.

Asked recently as part of a paper submission if the maidsafe network was sentient, that author replied, it self heals, grows, can segment, calculates what to do at any point in time in any area by the actions of many nodes current status and is capable of protecting itself, this is a question for the reader to decide upon.

A. The Issues Addressed by this Paper

- Privacy of network individuals (and maintaining net neutrality)
- Anonymity of browsing and using digital resources
- Security of data (both retention and theft prevention)

In Addition

- ID theft prevention
- Multi node processing
- Hardware problem detection ....

These are the key issues this paper addresses and it does so as an alternative to that offered by existing design. Confidence in IT should be close to 100% and this project may assist in that shift of thinking for users of technology.

B. Conventions Used

Client : This is the user interaction component of the system and performs most of the synchronous encryption, hashing and key pair generation for asynchronous encryption techniques, as well as presenting a user interface to the person using the technology.

Vault : These are simple computer processes that serve chunks, i.e. they serve data and do not allow or require login from users. The vault process is separate from the client processes and may reside on another machine.

H = Hash function such as SHA or MD5 etc. In this case SHA.

AES = Advanced Encryption System (256) - symmetrical encryption (maidsafe uses cfb mode cypher)

RSA = Rivest Shamir Adleman (4096) - asymmetric encryption

PBKDF2 = Password-Based Key Derivation Function (version2)

k = private key (used to decrypt public key encrypted data or to sign data)

p = public key (used to encrypt)

$\bigoplus$= XOR operation

netputV = put a Value on the network (Kademlia)

netputD = put Data on the network (maidsafe)

netgetV = get Value from network (Kadmelia)

netgetD = get Data from network (maidsafe)

C. Initial Requirements of System

There have been many advances over the last 20 years which provide many of the facilities required for this system. Namely, the Internet, Distributed Hash Tables (from the p2p revolution), the widespread use of computers even to the extent of the excellent (One Laptop Per Child) OLPC program and finally the rapid adoption and growth of available bandwidth, which appears to now be capable of Gigabit and Terabit per second in fibre optic links.

These initial requirements for the new network already being in place, all that is needed is a different way to design the whole thing, as though we were presented with a situation where computers, broadband, DHTs etc. were all just newly available. The question is glaring; Would we have built what we see now? The answer is likely a resounding no! So in summary we have:

1) Processing devices - mostly used less than 5% of capability most of the time
2) Storage devices with an average data element of 22Gb and approx 47% free space
3) Most of these devices connected to a network
4) Over a billion users and over 700,000,000 connect to the network daily
5) Many of these devices are on 100% of the time or a significant portion of the time and this is growing (with the advent of digital video recorders and other computerised home appliances such as sky, tivo etc.)
6) Devices that are not 100% connected can be largely addressed by today's p2p technologies (such as Kademlia routing[2]) to a greater extent than previously known. This paper adds to this concept with the inclusion of a ranking system.

Given the above parameters it would seem incredible not to design some system that made more effective use of the free resources we already have! This paper focuses on these resources as the initial building block of the system and chooses to ignore centralised or otherwise controlled systems.

The references used throughout this paper do not provide links to the source. A Internet search will find these documents in their current location. This is intentional as dead links tend to stop the user from continuing to find the cited document. Some references are from the IEEE, requiring an account.

D. Methodology

By taking a small proportion of every hard drive's free space and adding this together we would have enormous storage. In addition, the ability to duplicate data across many locations would give the ability to ensure data can be monitored and when necessary self repaired (by making further copies on disk part (node or computer) failure etc.).

Linking these computers in a way in which everyone benefits and does not necessarily have to pay any price would be seen as not only acceptable but as a significant benefit. To allow people to add data, whatever it may be, and be assured that their data is secure and perpetual for as long as they want it is not possible today with any reality (as we see from the data loss stories so prevalent). It is amazing that this proposed system has taken so long to evolve.

By using some of this space to store authentication records created by the users themselves we could allow self authentication[1]. This allows security to be not enhanced but altered significantly.

> It is inconcievable that in today's world the implemetation of stronger signatures, passwords and even biometric authentication is seen as a security enhancement, when in fact it is arguably the opposite! Today authentication records are transmitted back to a server or a cluster of servers (regardless it is a centralised corporate controlled environement), so imagine in this case you use biometric authentication i.e. a system reads your fingerprint, then sends it to an authentication system. If this fingerprint is stolen from that server then the thief not only has your password to that system but actually has your fingerprint. It is easy to change a password, but not your finger, so the question has to be asked again; is that really more secure? With maidsafe, nothing is transmitted back to a central resource, nobody knows you have an account, or who you are, so where can a thief steal your ID?

maidsafe or any other software system known today cannot defeat keyboard readers (or any device which intercepts keystrokes or other human input device). maidsafe takes the view that anything digitised is able to be copied, therefor it makes sense to limit these situations.

By allowing self authentication, people would be free of corporate controls or risks regarding their data protection. This could be achieved by users placing random data only they know the name of and password protecting that data for later re-validation.

People may prefer to be in control of their own digital assets and not to leave this to corporations.

Interestingly in this design, any disk space you donate or are required to provide to the network *will not store your data*, rather, the encrypted chunks of other people's files will reside on your space, and equally they will store the encrypted parts making up your files.

In the maidsafe network, the network will self-heal and data will move around the network to reduce bandwidth usage, remove defunct hardware from the network and also to reduce any load on individual nodes.

One of the most important parts of this network is that a truly distributed network will be created, enabling people to validate who they are and undeniably who they are communicating with.

To understand this more effectively it is best to think of the network as follows

This project	*Traditional*	Use
maidsafe layer	web/email/ftp etc.	Do things
Kademlia	DNS	Find things
Internet	Internet	Transport

Therefor it is noted maidsafe does not require ftp, web, email etc. and in fact can replace these technologies relatively easily with a faster and more secure network. In addition, the DNS system that has suffered in recent times with capacity issues, phishing (theft and fraud by pretending to be another site - using name mangeling or DNS poisining techniques) and other attacks is also not required. maidsafe also allows data to move freely and cache when in use. This means that busy sites or sites under attack actually get replicated and become faster rather than slower, or failing altogether.

II. Existing Technology Employed

A. The Network Layer

1) Transport Layer Communications: maidsafe employs a transport that is UDP (User Datagram Protocol) based, although as data transfer at the MAID layer requires reliability a TCP (Transmission Control Protocol) emulation layer is employed. This allows unreliable communications and firewall penetration (via hole punching, UPNP, NAT-PMP etc.) at the basic network layer (UDP) and also for small kadmelia messages. As the maidsafe layer requests reliability in sending data a TCP emulation layer is required. This was initially an air-hook implementation but more recently greater success has been with UDT[13].

2) Distributed Hash Table (DHT): The DHT is used in maidsafe technology as a key/*value* pair storage area i.e. we store a hash key and associate *values* with it. The intelligence of the Kademlia[2] DHT is that it distributes these key *value* pairs throughout a network in a manner that's very fast to look up (using an XOR mechanism). So we can store apple:*fruit* as a key and value and we could add several values to the key like apple:*core* etc. so the key apple would yield values *fruit* and *core*. All we do is store a key and value – we do not need to understand where it is; it is just stored. We can also check

[1] The ability for users to create their own encrypted user identifiers and have the network look after these in a manner that's reliable and guaranteed. In this way there is no authoritative controls over any person.

a key exists i.e. when we want to create an ID we can check for uniqueness by querying the DHT if required.

3) DHT Tuning: Kademlia, like most DHTs, can be tuned to affect performance; namely reliance to churn over bandwidth used and local processing power (although this is relatively small). A good example of this is cited in section 4.2 of [10] and also well researched in [3]. These papers should be read in order to understand the workings of Kademlia (alongside [2] of course). A particularly good example of trade-offs in tuning networks was written by Jun Xu [11] and efficient selection of Kademlia in this case to give $O \log_{2^b} n$ distance for the routing table as described by Jun yields the most efficient routing table size in the trade-off between bandwidth and efficiency.

B. Filesystem Layer

The filesystem layer is presented here as the user interface, although in reality the system has 2 user interfaces; the fuse interface and a graphical user interface to provide the login screen and amongst others the messaging screen, vault management, contacts etc.

1) FUSE (File System In UserspacE): The filesystem presented to the user has to be one which is controlled by the maidsafe client process and populated from the users' own datamap, whilst at the same time only appearing to have all the users data actually there (otherwise download all data all the time limits the system). For this purpose, a filesystem in userspace was required. The chosen system is slightly different for each tier1 operating system Linux has FUSE, MAC OSX has MACFUSE and Windows has Dokan.

All of these have a very similar API (Application Programming Interface) and allow files to be presented to the user by supplying directory structures and certain flags, such as file size, name, type, attributes etc.

These systems are all kernel level components and as such allow us to present a full filesystem to the user as a mounted drive (similar to a usb drive) where users can browse the drive like any other drive on their computer. This device is a software device and as such we can allow an infinite amount of data to be present on it. Any application accessing the data believes this is a normal filesystem and treats it as such. Underneath this, the maidsafe client software is retrieving bits of data and reconstituting pieces or whole files on request and at the same time may be chunking encrypting and storing data saved to this drive, all invisibly to the user.

C. MAID layer

1) Data Serialisation, Transfer and RPC system: In the maidsafe layer data is continually being serialised and sent across the network and messages containing requests and responses are being transferred. To accomplish this there are many serialisation options available. The option chosen here is 'protocol buffers', which is an open sourced project from Google (many Google components are used by maidsafe, including coding standards, test framework etc.). These allow data to be serialised based on templates known as proto files which are compiled into native c++ code which the programmer can use to read and write serialised data in a very efficient manner.

XML was considered, but rejected, as the overhead was too high for a system where the data end points can be so tightly controlled. Interconnection to the maidsafe API would not likely include data transfer issues and therefor the data serialisation would be considered a private matter to the system and not required to be exposed at the API level, otherwise XML may have some language agnostic capabilities making it more appealing.

III. System Building Blocks

A. Buffer Packet

To enable faster moving values than a standard Kademlia key could update or keep in synchronisation, a special Kademlia type is created called a buffer packet. This is a system where a key can accept many short TTL values. The recipients of these values read them then confirm they are new using the hash of the value being checked for uniqueness. This requires the TTL on the data to be at least 20% shorter than the remembered hashes list the recipient has to create and monitor. As an indicative step a TTL of 180seconds is recommended with a remembered list time of 50 seconds to ensure consistency. This allows what would seem like faster reads on the network than is currently possible. Here Kademlia delete is not used but rather TTL and data expiration is used.

XXXXXXXXXXX expand and finalise this

B. Generic BUFFER Packet

An important building block for the system depends on what we call buffer packets. These packets can be data chunks, stored on the vaults in a data form and referenced from the DHT key,value pair. For the purpose of simplicity here we depict the buffer packet as a value in the <key, value> pair.

We have known keys in the DHT key value, pair such as PMID, MPID, MAID etc. which hold differing information such as public keys etc at the moment. Knowing these keys means we can know of the existence of buffer keys by some hashing (i.e. to send to a buffer packet for DAVID would be H(DAVIDBUFFER)) - now we have the hash of the PMID for example as a key H(PMID) which we can easily find given a known PMID. If we want a buffer packet, we append the word BUFFER to the PMID and hash it. We can then find this key easily with H(PMID+BUFFER), which is true of all known IDs we have on the network.

The content of these keys (i.e. the values) are defined as follows for a particular owner of the buffer packet with relation to the senders to this buffer packet.

- $Sign(IDs\, allowed\, or\, ALL)_{[PMID]}$ (This signature allows the owner (PMID) to delete or remove content from the value. It also identifies which IDs can store information here)
- $Sign(RSAencrypt(message)_{[PMID]})_{[senders\ ID]}$ (signed with senders ID and encrypted with owners public key)
- $Sign(RSAencrypt(message)_{[PMID]})_{[senders\ ID]}$
- $Sign(RSAencrypt(message)_{[PMID]})_{[senders\ ID]}$

- ...May be repeated !

If this buffer becomes too full then an actual data chunk is created and the values of where the data chunk is stored is added here. The data chunk has the same parameters as this value and acts in the same way. This data chunk will be created by the next person wishing to save a message and the value added as the last entry on the list. In PD we can limit this buffer to 500Kb or some other figure decided, preferably by the network.

A special entry in the buffer packet $RSAsign(OnlineIP : PORT : RVIP : RVPORT)_{[PMID]}$ may be set, which tells other nodes trying to communicate that the end node is actually online at the relevant IP and UDP Port (the RV IP and port are described later). In this case, the censoring or filtering of messages is handled at the node itself, which will silently drop all non-authorised messages. This can be enhanced by creating a share-like scenario, where a key pair is created and the user is passed the public key to decrypt this location, or even a password is created for nodes to decrypt this information if synchronous encryption is decided to be a better option here.

C. *Chunk Info Holder (CIH)*

The CIH contains 2 peices of logic: the watchlist and the reference packet.

1) Watchlist: The watch list is a list of 4 IDs that are all self signed and works as follows:

- A client requests that the network store a chunk by selecting 4 nodes from his routing table with the appropriate rank

The IDs are the PMID IDs, and are not linked to the user ID or public ID; they are only used to save data and request delete of data chunks. This list may eventually be encompassed in the rank packet described later.

2) Chunk References: As a vault stores a chunk, it updates the chunk reference packet with a message signed by its PMID. This packet takes the form

Chunk Name (Key)
$PMID1 - NAME_{[PMID]}$
$PMID2 - NAME_{[PMID]}$
....

The key holders are the k closest nodes to the key as is standard in Kadmelia networks. They must know this is a chunk reference packet and follow these rules:

1) Allow deletion of a value by the signing authority of the value (in this case the value is a PMID (512 bit hash)) and the signature is the PMID signed by the PMID private key. The deletion instruction would have to be signed by the PMID private key. This can be thought of as self deletion, i.e. it is the PMID deleting itself from the reference.
2) Allow deletion of a value on lost chunk - by the request of any of the other chunk holders signed by their PMID. In this case the holder will confirm with the 'to be deleted' PMID that the vault has in fact lost the chunk (or stopped storing it). This is achieved via a RPC. If the deletion request is valid, the reference is removed and

Figure 1. Add to Watch List

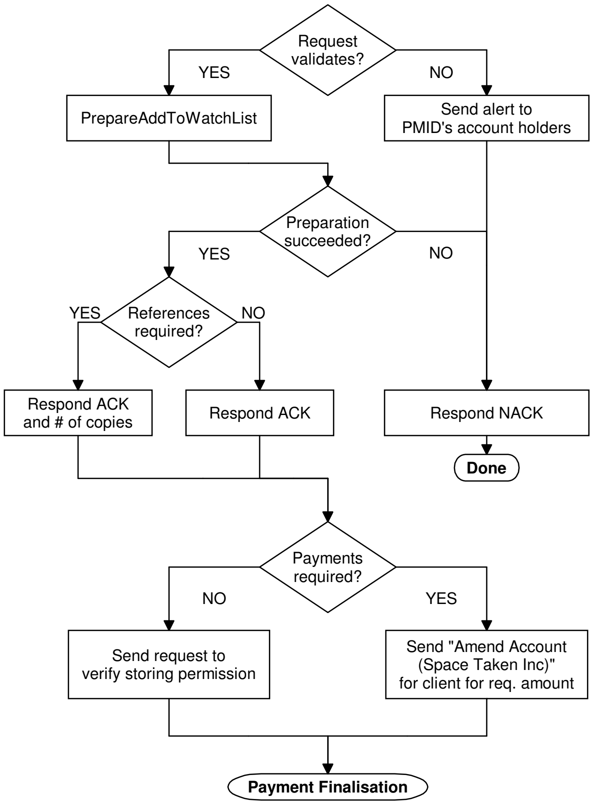

Request validates?
YES
NO
PrepareAddToWatchList
Send alert to PMID's account holders
Preparation succeeded?
YES
NO
References required?
YES
NO
Respond ACK and # of copies
Respond ACK
Respond NACK
Done
Payments required?
NO
YES
Send request to verify storing permission
Send "Amend Account (Space Taken Inc)" for client for req. amount
Payment Finalisation

Figure 2. Remove from Watch List

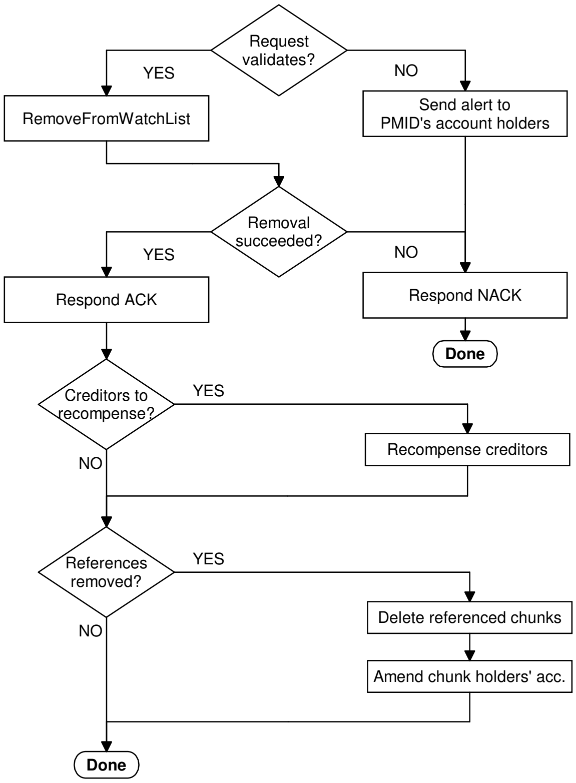
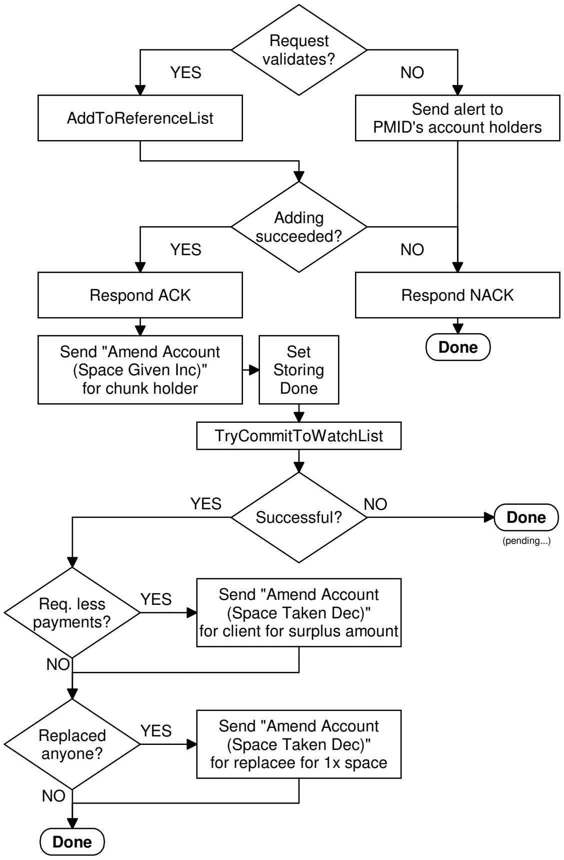


Figure 3. Add to Reference List


Request validates?
YES
NO
AddToReferenceList
Send alert to PMID's account holders
Adding succeeded?
YES
NO
Respond ACK
Respond NACK
Done
Send "Amend Account (Space Given Inc)" for chunk holder
Set Storing Done
TryCommitToWatchList
Successful?
YES
NO
Done
(pending...)
Req. less payments?
YES
Send "Amend Account (Space Taken Dec)" for client for surplus amount
NO
Replaced anyone?
YES
Send "Amend Account (Space Taken Dec)" for replacee for 1x space
NO
Done

the existing chunk holder is messaged (to their buffer packet - or direct), declaring that there is a lost chunk.[2].

3) For both 1 & 2 above, the deleted PMID rank packet (seeV-F) is decremented by a value of the number of kilobytes of the chunk.
4) In the case of a PMID being added to this reference, the PMID add reference message is stored in the cache directory.

D. Create a Generic ID (e.g.. MAID, ANMID, ANTMID etc.)

MAID, ANMID, ANMAID, ANTMID, ANMPID are examples of this computation (note *MPID* and PMID are NOT !)

1) $RSAgenkeypair_{()} \equiv (k_{[]}, p_{[]})$
2) $RSAsign(k)_{([p])}) \equiv P_s$(signed so it can be deleted by owner)
3) $SHA(P_s) \equiv ID$ (ensures content must meet this criteria)

Algorithm 1 system ID=(SHA512Hash(Public_Key+Signature [Signed_by_some[priv]]))

4) $netgetV(ID) \equiv NIL$ (To ensure uniqueness, or back to 1)
5) $netputD_{[ID]}(p_s)$ (ID packet, stored)
6) Store k and ID in the private key chain $\hat{P}$ of the DA for later use

E. Public Key Infrastructure (PKI)

To check an ID that has apparently signed a request.

1) $netgetD(ID)$
2) Read content (public key)
3) Confirm signature (using standard PKI rules, check privately signed data with the public key of the id used)

The issue is that unless the public keys are protected (in that they cannot be altered) such a system would fail, as any cracker could essentially replace the public key and then pretend to be the ID. In maidsafe this is made impossible using protection mechanisms from deleting MPID packets as shown in section VI-H. maidsafe also provides anti fraud mechanisms by having all users copy all public keys from MPID they communicate with. Thereby fraudsters would create system alerts by trying to replace this key. This gives the owner an opportunity to re-save the key onto the maidsafe network. (anti Sybil attacks!)

This gives us a distributed validated and updated PKI infrastructure with one proviso: we require that these chunks are not tampered with, and that is where we use SHA. The SHA of many ID keys is the chunk name and also the ID associated with the chunk. With ranking and perpetual data this ensures the chunk is valid as any chunk on the maidsafe network. These anonymous chunks can sign non-anonymous chunks, and then only they can delete them (signing is built into chunk name - cannot be retrospective, i.e. you could not (given today's technology) create a key pair that creates the same resulting hash name, so fraud would be computationally near impossible). This is described In Algorithm 1

As we have well formed chunks and a validity check, we can be assured the public key we get is the key that was saved in that chunk and has not been tampered with. The MPID and similar keys are very small and stored on trustworthy machines (or high-ranking machines as described in section V-F) in perpetuity.

This is a self-regulating and self-managing global PKI infrastructure.

F. System Packet Structure

Every system chunk is signed (RSA 4096) with a private key and is named using differing conventions. These are :

MID $Sign(\mathcal{MID}content)_{[ANMID]}$
Content == random number signed by ANMID
ID == H(H(U) + H(P))

[2] It should be noted here, this is a case where a signed data bit can be deleted, by someone who is not the signatory. This is only allowed by reference holders and confirmed by every reference holder. i.e. when an update is done here then the reference holder republishes the value to the K closest nodes immediately. This republish MUST contain the signed deletion instruction in the case of point 1 or it must contain the deletion request in point 2, in this case each holder confirms again the deletion is in fact true, otherwise republish again. Republish does not mean any node updates the rank this is only done once by the original reference holder.

SMID $Sign(\mathcal{SMID}content)_{[ANSMID]}$
Content == random number signed by ANSMID
ID == H(H(U) + H(P) + 1).
MAID $Sign(\mathcal{MAID}content)_{[ANMAID]}$
Content == Public Key signed by ANMAID
ID == H(content)
TMID $Sign(\mathcal{TMID}content)_{[ANTMID]}$
Content == (data map of data atlas) encrypted using users password signed by ANTMID
ID == H(content)
MPID $RSAsign_{[\mathcal{ANMPID}]}(\mathcal{MPID}content)$
Content == Public Key signed by ANMPID
ID == H(public name)
PMID $RSAsign_{[\mathcal{MAID}]}(\mathcal{PMID}content)$
Content == Public key signed by MAID
ID == H(content)

MSID - (private share data map held locally and updated on the network via messages)

PPMID - Purchasable Proxy maidsafe ID - Used to maintain a list of disk space which may be allocated by maidsafe to others either at a cost or donation etc. This is used during the pdzone registration process.

To maintain separation of the packets from any linked information they are all signed by an anon ID and these IDs are named ANMID, ANMAID etc. with the exception of the PMID which is signed by the MAID. The reason for this is the MAID and PMID are linked and it is this signature that links them. The one way signature (MAID->PMID) is a measure of security in case of vault failure or intrusion possibilities. The MAID security measures are stored securely on the network but the PMID private key is contained at the vault location. The system can unauthorise a vault via the MAID ID if there are any security issues at the vault. This is a fail-safe measure to ensure integrity of data as the PMID chunk on the network can be deleted and a new one created on failure of any kind on the vault.

In future there will be the followng two additional system packets.

MMID : $RSAsign_{[\mathcal{ANMID}]}(\mathcal{MID}content)$, ID = hash ((username + pin + Date))

MSMID $RSAsign_{[\mathcal{ANSMID}]}(\mathcal{SMID}content)$ = This is the movable surrogate MID packet used to ensure MID integrity during an update (write after update). Again this ID is the H(U + P +1 + date) hashed.

IV. Distributed Hash Table

A. Overview

This section assumes the reader has read the Kademlia paper referenced in this section as repeating the details is not of value here.

In a standard hash table you could consider a simple two column database where column 1 is a hash and column 2 is the value. This is in fact how a hash table works. Normally the hash is a calculable value which produces a key and with this key the data or value is retrieved from the hash table.

Example 1. If we have some fruits, say apple, orange and pears and we stored the number of these items in a hash table then it's done by taking a hash of the name apple, orange or pear and creating a record with the hash as key and the number of the fruit as the value. If later we wish to find out some information about the apples we hash apple and look up the value against this key (in this case number left in store).

maidsafe, however, does something different (as you may expect) and actually stores content in many cases which is the value and this value hashes to the key! This seems backwards as if you have the value already, why hash it to look up the value? Well in maidsafe we don't have the value we have only the key(as normal) but we know the value we get will hash to the key. This is where we use the hash technique to our advantage. We know the value we retrieve will hash to the key but we don't know what the value is (this is empirical in hash techniques, one way mathematical algorithm to produce fixed length hashes).

The reason for this is twofold; initially the value is generally a large chunk of data we cannot hold ourselves as it would be too large (consider we may have several million keys per user, adding to several exabytes of data if the values were all local). Secondly the security of a distributed system is paramount, as you do not have access to the machines the data is on you require a mechanism to ensure that the data you get back is valid and not tampered with, maidsafe does many different checks on this, but there is a final user check available which ensures the value hashes to the key!

So maidsafe uses Kademlia to store keys and values as well as routing information, but uses an overlay network (maid layer) to ensure data robustness (explained in Vaults section V) and integrity, as described and also further detailed in Vaults section V.

B. Distributed

The power of distributed hash tables over regular hash tables is found in the ability to split the database over multiple locations. These locations are in themselves hash keys, usually in the same address space as the values (but this is not a requirement). The distinguishing factor here is what request for the data is made (in the case of Kademlia a FIND_NODE is an address look-up and FIND_VALUE is a data (or value) look-up).

To find a value in Kadmelia, the search congregates on the nodes closer and closer to the value being searched (like a find node) and when a node with the value (which may be very far away from the value) has the value it replies to the FIND_VALUE with the actual value. In opposition to a FIND_NODE where merely closer and closer nodes are returned (there is no value in this case) until some node has no details closer than requested. This is then the closest node unless others returned in the alpha replies are closer. In the case of find_node the answer is closest node whereas find_value is the actual value being searched.

C. Kademlia

maidsafe utilises Kademlia[2] as the Distributed Hash Table (DHT) of choice. This was considered alongside Chord, Chord2, Tapestry and many others. Kademlia was chosen as it presented the most random method of distribution of nodes and using the XOR searching criteria allowed very fast network resolution. This is clearly identified and described in the paper[2].

Standard Kademlia, though, has some limitations and an example of some limitations are:

- Susceptibility to node failures causing bad routing information to be propagated for a longer time than necessary
- Reliance on a balanced routing table match the balanced binary tree. Standard Kademlia does not put any favour on closer nodes being more important (examination of the binary tree routing algorithm will quickly show the closest nodes are more important than the very sparse further away (XOR distance)) nodes. Kademlia does recognise the closer more important nodes in its standard, as it does not assist these nodes in any way.
- Potential 'race conditions' in republishing values between the k-closest nodes to any particular value. This is a situation where a value is republished and all k nodes then decide to republish in a given interval. It is obvious that they will all republish at interval n after a republish is seen. This means all nodes may in fact be crossing over each others republish times, causing the race.

These first three potential issues are addressed and analyzed in a paper[3]. These additions have been recognised and improved upon by maidsafe in creating perpetualdata. Further issues with Kademlia and DHTs in general is the ability to amend (or delete) values after publishing. This is due to the inherent distributed nature of the values (as the name suggests). As values are distributed in a pseudo random manner (mathematical predictability becomes an issue due to the large amount of variables (routes, nodes, node names, locations etc.) and random human input).

Random here is used to indicate not computationally recognisable with today's resources.

Searches on Kademlia networks are very fast and iterative (*not recursive*). This means that each node will ask some nodes a question, get answers (hopefully) and from the answers construct a new node list to ask. If this were recursive, then a failed node would halt the process; as this is iterative then failed nodes simply timeout whilst the search continues (this is the β (required replies to continue) requirement, from the α (parallel level) searches). As Kademlia uses a binary tree and XOR searching the search time per iteration is $\mathcal{O} \ln n$ (where $\mathcal{O}$ is network latency and a factor of XOR distance between node IDs (holes in address space) and n is number of nodes, not number of possible nodes). This figure can be improved upon by the additions shown so far. Further examination of the $\mathcal{O}$ constant may lead to further improvements in search completion times by not only using decreasing distance but also decreasing distance with path protection (multiple routes protected) by examination of XOR distance between parts of the network, particularly where there is binary imbalance in early days.

D. Value Handling

In Kademlia there are several methods of ensuring values are consistent and protected at the same time. The initial value to be considered is K and this should be chosen to be the number of random nodes that can be allowed to go offline in a Refresh (Refresh) time. Also there is a republish time (Repub) which is the initial store of the value and sets a time to live (TTL) value on the data.

1) Caching: In Kademlia, values are cached distant from their natural location of the K closest nodes. The TTL is used in this instance to calculate how long a value can live and the further away from the K closest determines the TTL value that each value has. This value is chosen as the reciprocal of the distance from K of the value (so 2 steps away the TTL is halved and 4 steps the TTL is quartered etc.).

This can present issues when the refresh time (usually 60 mins but configurable) is due. In this case the node must check it is in the K closest and if so refresh (i.e. send a store_value) the K closest nodes. If a node is not in the K closest then this store RPC should not be sent, unless all of the K closest nodes do not have the value and the TTL has not expired, in this case only a node with a value which is not in the k closest group will republish (the true TTL so reverse of inverse reciprocal) to the K closest. In maidsafe, β refresh is used to vary in an evenly distributed way the 60 minute intervals to reduce potential for race conditions in republishing values. If a value is deleted the closest nodes with the value (k closest) should be updated with a -1 value and this should not be cached, this prevents cached copies actually republishing values that are due to be removed anyway (i.e. ignoring or overriding the delete instruction)

2) Time To Live Values: The TTL can be different for any different type of file, from seconds to a figure meaning do not delete ever (i.e. -1). For fast changing data the TTL should be small, and for unchanging data (like digital keys) the TTL should be very large. This value is only reset by the storing node unless another algorithm for that value type is known by the Kademlia node i.e. in the case of vault validity check, it will be by one of the identified Kadmelia IDs watching the chunk, so the value will hold the IDs allowed to republish the data (also the ID will be allowed to delete himself from the list but not the others). Using this mechanism, maidsafe uses a 24hr period for reference pointers and a 60 day period for signature packets, which will also republish these Kademlia values in case of error at every validity check. This is done simply as a speed up, as Kademlia values are retrieved much faster than maidsafe data.

E. Delete or modify values

1) Description: As previously stated, modifying a value in a distributed hash table is notoriously difficult. This is endemic in the very nature of the pseudo random distribution of data in such a network (the DHT strength is one of the largest weaknesses here). If a single value is altered via any method, the value itself may exist in many places and it's finding these places that's the problem. We have already seen the distribution as random, therefor this implies you would be required to search every node in the network (not only closest) for copies and make sure they are also altered. To make this even more difficult, another node may also be altering the value at the same time, thereby causing chaos in the synchronising of data on the network, eventually leading to total collapse.

This would imply that DHTs are only good read only, however, in the case of maidsafe and pd, this negative attribute is turned into a positive and essential aspect. As noted in the section 1, a PKI network actually requires non changing of identities that are linked to a public key. In this case this problem described above is fantastic and we're glad we have such an issue. Added to this fact, even trying to alter the content of a packet of data that hashes to the key is an illegal instruction in maidsafe, therefor there's a double benefit for us.

The only changeable data stored on the Kadmelia network as far as a maidsafe implementation is concerned is data deemed as *"Multi value one of which should be correct*". By this it means a value should have several results and as such one of the results should be correct, but not necessarily all of them. This seems weird but is in fact very simple, basically every value will hash to a key or will contain multiple values (maidsafe uses signatures in addition to ensure safety through validation) and of each value a correct answer should exist in one value.

Example 2. maidsafe stores pointers to data chunks (data hashes to a key again) as Kademlia values. The number of pointers is usually four (4). As this is a truly distributed network maidsafe expects nodes to be off or damaged or somehow unreachable and expects this on fairly regular terms. On retrieving a value, each pointer is tried to see if not only the node is there, but also that the node has the data (see Vaults, as data moves around an awful lot). In this way a dead or incorrect value is allowed and catered for.

To add to this, maidsafe makes use of a couple of features of the upgrades to Kademlia explained earlier. One such feature is force-K, to ensure close nodes are given a high priority, using the maximisation of near neighbour knowledge. A small addition to enhance this aspect is to flag all data cached throughout the network as cached and not a value within the k-closest nodes to the value. In this manner, values retrieved can be tried for validity in the knowledge that although unlikely, it may be stale and if so, continue the search for the data going to the k-closest nodes to ask for fresh data.

To get to the k-closest nodes, we require that nodes do not return anything but closer nodes. On finding no closer being returned would indicate we have got one of the k-closest nodes. This is where we alter the workings of Kademlia and ask the closest node once more for the k-closest nodes to the value were looking for. With the force-k addition this would likely include the k-closest nodes (this can be further enhanced if necessary by asking all nodes returned the same question again and sort the answer by XOR closeness). Assuming we have the closest nodes we can do many things but initially we can assume we have the valid freshest data. This should return pointers (from the above example) that are valid.

Nodes with flagged cache values will check using this process that they are not in the k-closest nodes and timeout (exponentially) the cache value.

2) Locks: To achieve reliability of data where multiple copies may exist and be altered from multiple locations is a problem that requires addressing, before changing data. Using the above scenario a node intending to alter a piece of data tries to get all the k-closest nodes via an iterative search. On success each node is sent a lock request which it answers with (Acknowledge) ACK or (Negative Acknowledge) NACK. Each node can pass this request to its known k-closest nodes. On receipt of k ACKs, the node then alters the value. On receiving an altered value the holding nodes release the lock. Failure to receive a value update in 10 seconds should auto release the lock.

Two locks in collision (which can still happen) are compared for rank and the highest rank gets the lock, the lower of the two gets the NACK.

V. VAULTS

A. Overview

The vaults are a very capable, fast and efficient massively distributed storage mechanism but *not* merely a distributed filesystem as in Andrews or CFS. Instead, each vault is a conceptual storage mechanism that can be presented on any filesystem whether a local system such as FAT, FAT32, NTFS, EXTx, resier etc. or a network file system such as NFS, Andrews, SMBFS etc. although debateably there would be no requirement for network file systems in a maidsafe network.

The vault is primarily the storage component of the network, but is also partially responsible for signature validation and decryption. It is a service or daemon controlled by either an individual client or several clients and makes use of the following layers:

maidsafe

Transport

- Transport Layer (Layer 1) is responsible for providing the network routing, NAT traversal and network churn management.
- maidsafe (MAID) Layer (Layer 2) is responsible for implementing most of what follows in this paper, including all algorithms relating to self healing (additional churn control), ranking, perpetual data, self encryption and most importantly self or anonymous authentication.

To allow fairness in the system of vaults a measurement has to be made of the vaults' capability to store and provide access to data. This measurement should take into account at a minimum the vaults online time, bandwidth, processing speed, free CPU availability, and drive capacity. This may seem like a lot to take into account and measure with any degree of accuracy, so a system that would allow every aspect of a vault to be measured had to be developed. This is known as 'dynamic ranking', see V-F and is the fairest mechanism to 'measure' a vaults capability.

In nature this mechanism is made clear for us to see, the strength of an organism is how best it survives, the most effective hunters obtain the most nourishment, the most efficient plants grow to gather the most sunshine and so on. The lesson is the more efficient a system, the stronger and more effective it is.

Vaults are organised to mimic this philosophy in computing terms, which means a system that tolerates (to varying degrees) inefficiency as well as rewarding efficiency (as in nature). Vaults 'raison d'être' is to store and allow retrieval of data; the more data it can store and provide the more efficient it is. The ability to store data should then be based in a logical manner, which as it turns out is a very effective process.

The vault process is separate from the client processes and may reside on another machine.

B. Network

1) NAT traversal: Vaults require a publicly accessible UDP port for the network to function. This is made difficult today as most computers are behind firewalls. There is currently no method of firewall penetration that is guaranteed to work in every case. There are several attempts such as STUN, TURN, ICE, UPNP, PNP-NAT and many others, but on their own there is no guaranteed method with the exception of (debatable) TURN where every node is connected to an external node and relays all traffic through it. This is not an acceptable situation for large data transfers though as the load on the TURN server would be significant.

There is a mechanism called UDP hole punching which shows success rates of around 83-92%. maidsafe utilises standard UDP hole punching using known accessible addresses to punch the hole through. The nodes that are seen as 'direct' connect are:

1) Actual direct connect
2) UPNP / PNP-NAT connected
3) Full-cone NAT
4) Manual port forwarded connections

Each of the above can act as the hole punch go between in the maidsafe network giving us at least 80-90% penetration. The other 10% or so are regarded as not vault compatible and users would not be able to run a vault from that location (it would go into beacon mode as described V-B2).

2) Port usage: Vaults will default to use a random port in the range 5001-65535 (user definable UDP ports) in most cases the exceptions are.

User defined, this is where a user decides to manually port forward a router port to the vault (via the client RPCs).

The other port will be a standard fall back port *5483* and this port will broadcast the vaults presence on start-up to the local network (via a UDP broadcast). Any ports responding will do so with their KADID (which is actually the PMID) and will go into the vaults' routing table as the internal IP:port of that PMID. The vault will only ever pass this to other internally connected vaults. To any external request this information is ignored and the PMID external contact details are used.

3) Network Segmentation: In the case of network failure / segmentation all vaults will periodically broadcast a bootstrap (FIND_NODE RPC) on this port to locate other nodes. When other nodes are found, the internal contact details PLUS the normal external contact details are provided, if available. This allows the networks to reconfigure into smaller networks and hopefully find the parent network again when the network error has been sorted. In large catastrophes like continents going offline (possible) this allows maidsafe to continue to operate and it is hoped the data would be 100% available very quickly on such an outage.

4) Listening socket: Vaults will start a listening socket for incoming messages and deal with them appropriately. The issues to be aware of here are:

1) Restarting socket/vault in event of failure (achieved by having the fallback socket ping this socket every 10 seconds)
2) Closing bad connections - i.e. attempted buffer overflows etc.
3) Dropping bad messages without harming the processes
4) Maintenance of a session length black list of IP:PORT combinations that are faults and to be dropped. *This may extend later for network wide blacklists.*

C. Vault Control Remote Procedure Request

MAID == MAID of vault owner

PMID==PMID of vault owner

PMID1==PMID of a friend or other vault

See Table I for a list of vault RPCs that are controlled by the owning client.

See Table II for a list of vault RPCs that are used by other vaults and clients.

D. vault Registration

1) Local Vault Registration: This is the situation where a person is registering themselves on the system that actually will run their vault. A user's vault doesn't necessarily need to run on their computer or the same computer as the client, only one which they have access to and is online most of the time with some available disk space. On successful registration the user can then log into any maidsafe client on any machine and access their data. In the case of a family situation then perhaps all the family will create their user ID on a single machine that's always on, but then choose to use the client part on only their own machine.

1) A vault starts and waits for a client to connect with it to provide a PMID key pair.
   a) Connection on 127.0.0.1 port 5483 (beacon port)
2) vault ID (PMID) is generated for a vault by a client.
3) For a locally started vault: the client creates a PMID key pair and a PMID system, where PMID structure is $Sign(PMID[pub-key])_{[MAID]}$
4) This PMID is saved to the network $netputV_{PMID-packet}$
5) The vault will then be contacted via an RPC message (RPC SET_OWN) by the local client.
6) The vault takes the keys (and possibly a start port (RPC SET_PORT) for situations where users will port forward rather than have maidsafe hole punch or otherwise get through a firewall) and creates a config file which it will read on startup. The config file contains at a minimum an optional PORT and required PMID private and PMID public key, MAID public key, signature of PMID pub key with MAID private key and node ID (which is hash of PMID_pub_key signed by MAID).

Request	Response	Purpose
$Sign(PMIDkeypair)_{[MAID]}$	$NACK\Vert ACK$	SET_OWN
$RSAenc(Sign(PORT\ number)_{[MAID]})_{PMID}$	$NACK\Vert ACK$	SET_PORT
$RSAenc(Sign(PMID1keypair/space)_{[MAID]})_{PMID}$	$NACK\Vert ACK$	ADD_ANOTHER_PMID
$RSAenc(Sign(Space\ Avail)_{[MAID]})_{PMID}$	$RSAenc\ x\ Mb_{[MAID]}$	FREE_SPACE
$RSAenc(Sign(All\ Space\ Avail)_{[MAID]})_{PMID}$	$RSAenc\ List\ dir/space_{[MAID]}$	FREE_SPACE_ALL
$RSAenc(Sign(All\ Drives)_{[MAID]})_{PMID}$	$RSAenc\ List\ dir/space_{[MAID]}$	LIST_STORES
$RSAenc(Sign(List\ dir/space)_{[MAID]})_{PMID}$	$NACK\Vert ACK$	SET_STORES
$RSAenc(Sign(DeletePMID)_{[MAID]})_{PMID}$	$NACK\Vert ACK$	DELETE_ALL
$RSAenc(Sign(DeletePMIDX)_{[MAID]})_{PMID}$	$RSASign(Delete)_{[MAID]}$	DELETE_A_PMID

Table I
VAULT CONTROL RPCS

Request	*Response*	*Purpose*
$RSAenc(Sign(SAVE\ name\ size)_{[PMID1]})_{PMID}$	$NACK\Vert ACK$	SAVE_DATA
$RSAenc(Give\ name)_{PMID}$	chunk	GET_DATA
$RSAenc(Sign(name + RND\ DATA)_{[PMID1]})_{PMID}$	$RSAenc(Sign(hash)_{[PMID]})_{PMID1}$	VALIDITY_CHECK
$RSAenc(Sign(+ + name)_{[PMID1]})_{rankPMID}$	$NACK\Vert ACK$	INC_RANK
$RSAenc(Sign(- - name)_{[PMID1]})_{rankPMID}$	$NACK\Vert ACK$	DEC_RANK
$RSAenc(Sign(- - -name))_{rankPMID}$	$NACK\Vert ACK$	REP_DEC_RANK
$RSAenc(Sign(addReference)_{PMID})_{referencePMID}$	$NACK\Vert ACK$	ADD_REF
$RSAenc(Sign(delReference)_{PMID})_{referencePMID}$	$NACK\Vert ACK$	DEL_REF
$RSAenc(Sign(addmessage)_{allowed\ PMID})_{PMID}$	$NACK\Vert ACK$	ADD_MESSAGE
$RSAenc(Sign(read\ all\ message)_{PMID})_{PMID}$	$RSAenc(Sign(msgs))_{PMID}$	READ_MESSAGES
$RSAenc(Sign(delmessage)_{allowed_{PMID}})_{PMID}$	$NACK\Vert ACK$	DEL_MESSAGE
$RSAenc(Sign(is\ online)_{allowed_{PMID}})_{PMID}$	$RSAenc(Sign(IP : port))_{allowedPMID}$	IS_ONLINE

Table II
VAULT GENERAL RPCS

7) The vault then starts its main port and goes through the normal bootstrap process.

2) Remote Vault Registration: TBD

E. Vault Startup

1) A vault starts up as a daemon process and reads its configuration file, or waits on ownership in registration mode, see V-D1.
2) From the file it takes the hash of the signed public PMID key, which is used as the vault ID (==PMID)
3) The vault now carries out a network bootstrap to determine its connectivity status (behind firewall etc..). This is done from a list of potential network bootstrap nodes who are directly connected[3]’ and can act as rendezvous nodes if needbe.
4) The vault now carries out a Kademlia bootstrap (FIND_NODE==PMID). This is done from a cache of previously known Kademlia nodes (sorted by rank - highest first)
5) The vault then does a check on each chunk to determine what chunks are still under its control (i.e. it is in the reference holders value for the chunk).
6) The vault then moves the lost chunks to its cache directory

F. Rank (or fairness meter !)

For this system to operate effectively the vaults have to be able to provide a mechanism that allows the good to grow, the weak to fail and all the options in between. This is accomplished using special system packets called rank packets which rely on a the ability to delete single signed values from a chunk containing a list of values. This particular chunk is called a rank chunk.

So a rank chunk is H(MPID + "RANK") (the actual word rank).

This packet is held as a Kademlia key value in the k-closest nodes as standard in Kademlia. As in III-C2, the node has to know this is a rank packet and follow some rules regarding it.

In this document rank is described in a fashion that is basically a count of how much data any PMID has on the network at any one time and this is incremented or decremented to allow a node to recover space, i.e. when a user deletes a file, the client part of the system will instruct the vault to delete the equivelent number of chunks that make up the file. This deletion means the vault moves some chunks to a cache dir, but only after instructing the network for a reduce rank message. Failure to instruct a reduce rank message will in fact cuase the network to reduce the rank by a factor of 2 or 3 times the actual amount of data the vault deletes.

1) Rank Decrement:

1) On receipt of an decrement rank message, the signature is checked to confirm that this is a signature of the

[3] In this scenario directly connected means has a publicly accessible IP/PORT that’s directly accessible from any node without requiring rendezvous or similar.

reference chunk holder, the remove request notice is passed as a part of this signed message and the signature of this remove notice is also checked.

2) The rank is decremented by the size of the chunk in kilobytes and the validation packet described above is passed to the k-closest nodes to do likewise.
3) Any note to decrement a rank is checked by all nodes. In case of disagreement this is a majority rule situation where the majority will likely be the actual rank sorted due to churn effects of the network.

2) Rank Increment: In this case the actual PMID to be incremented is responsible for acquiring the increment notice. Again the number to increment by is the size of the chunk in kilobytes. This is done using the following process:

1) Store request used to store the chunk initially is wrapped in the increment request sent to the reference holder by the PMID. The reference holder stores this request in his cache folder.
2) The RANK holder contacts the reference holder to confirm this addition to the reference packet took place. This is requested by the PMID storing the chunk.
3) The reference holder checks his cache dir for the relevant add reference PMID to the value, confirms this back to the rank holder.
4) The reference holder then republishes the new chunk reference with the k-closest nodes and the copy of the increment request.
5) The reference holder deletes the relevant add reference file (to stop multiple adds).
6) The RANK holder now republishes the updated rank with the copy of the increment request

G. Find Rank

How do we store / find this rank etc.

The rank is part of the contact tuple for each PMID on the network, transmitted and saved via routing tables. The update rank can be obtained at any time by checking the rank packet. This rank should be returned with every FIND_NODE RPC. A FIND_RANK RPC will allow the Kademlia network to be queried for all nodes close to the RANK packet and the average of the returned values is considered that nodes' rank. This RPC may be performed prior to a STORE_DATA RPC to ensure the rank is correct and up-to-date, when required.

The client can decide to carry out a rank check (get the H(node ID + "rank")) if less than alpha returns contain the same rank for that node.

H. Save Chunk Location

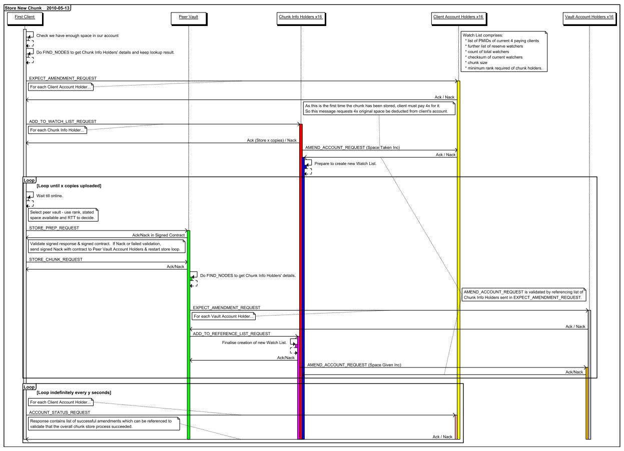
To store a bit of data we firstly need to determine which nodes to store the data on. We need to know the RTT for nodes in our routing tables and we calculate the locations based on rank / RTT e.g.

1. Our rank is 1123

2. We search for nodes with a rank just short of this or equal (from our routing table)

3. We get the distribution of these nodes by call - closest - furthest - then the two at ~33% and ~67%.

4. Do a swap chunk


Store New Chunk 2010-05-13
First Client
Peer Vault
Chunk Info Holders x16
Client Account Holders x16
Vault Account Holders x16
Check we have enough space in our account
Do FIND_NODES to get Chunk Info Holders' details and keep lookup result.
Watch List comprises:
* list of PMIDs of current 4 paying clients
* further list of reserve watchers
* count of total watchers
* checksum of current watchers
* chunk size
* minimum rank required of chunk holders.
EXPECT_AMENDMENT_REQUEST
For each Client Account Holder...
Ack / Nack
As this is the first time the chunk has been stored, client must pay 4x for it.
So this message requests 4x original space be deducted from client's account.
ADD_TO_WATCH_LIST_REQUEST
For each Chunk Info Holder...
Ack (Store x copies) / Nack
AMEND_ACCOUNT_REQUEST (Space Taken Inc)
Ack / Nack
Prepare to create new Watch List.
Loop
[Loop until x copies uploaded]
Wait till online.
Select peer vault - use rank, stated space available and RTT to decide.
STORE_PREP_REQUEST
Ack/Nack in Signed Contract
Validate signed response & signed contract. If Nack or failed validation, send signed Nack with contract to Peer Vault Account Holders & restart store loop.
STORE_CHUNK_REQUEST
Ack/Nack
Do FIND_NODES to get Chunk Info Holders' details.
AMEND_ACCOUNT_REQUEST is validated by referencing list of Chunk Info Holders sent in EXPECT_AMENDMENT_REQUEST.
EXPECT_AMENDMENT_REQUEST
For each Vault Account Holder...
Ack / Nack
ADD_TO_REFERENCE_LIST_REQUEST
Finalise creation of new Watch List.
Ack/Nack
AMEND_ACCOUNT_REQUEST (Space Given Inc)
Ack/Nack
Loop
[Loop indefinitely every y seconds]
For each Client Account Holder...
ACCOUNT_STATUS_REQUEST
Response contains list of successful amendments which can be referenced to validate that the overall chunk store process succeeded.
Ack / Nack

I. Swap Chunk

Swap Chunk example

Swapping chunks is the method used to store data. You will already have candidates with good geo-location and rank. The offer to store a chunk of the same size is made to them (they should be happy your rank is probably better than theirs).

Recipient

1. Checks your rank

2. If you have any lost chunk references - they will request you contact the holder of a lost chunk and get another copy and have them sign a stored receipt in your rank packet.

3. They agree to the swap. The chunk they swap will be of a same or less size - but their choice of which chunk is decided by a few factors a: size (obviously) b: Length of time they have held the chunk (use oldest first) / OR a chunk which shows your RTT is further away than their partners for this chunk.

You

1. On swap agreement - you send your chunk the first (closest partner) and the nodes to store on.

2. Each node in turn replies with a stored OK - you remove from your queue (this can be carried out much faster by relaying all data from start through partners to last partner - single transmission). Each node keeps a copy of the data swapped - i.e. swapped chunk ABC for DEF with node 243 etc. (see cache lost chunks) See later about how

J. Cache Chunk

When searching for a chunk (akin to find value in Kademlia), the client tells the last node that did not have the chunk to take a copy of this data. The last node gets the data and consequently find value will return this chunk to the searcher.

Cache times - or how long do we store a cache chunk ?

Option 1: As in Kademlia we store the cache chunk for a time which is inversely proportional to the XOR distance from the original.

Option 2: Store in a FIFO directory - so don't delete till FIFO is full.

DISCUSS !!

K. Cache Lost Chunk

In the special case of lost chunks we need to make copies. How to do this fairly remains a point of internal debate. It would appear there may be a flaw in the design as a hacker may just request all his chunks are lost copies, thereby bypassing any payment.

So there are a couple of rules to stop that and enhance the system:

1. Chunk lost store request must be co signed by original storage partners.

2. Cache lost chunks stored in a separate locations from normal cache (only to differentiate) but follows same rules about giving the data.

3. Cache chunk is stored along with the ID of who stored it.

4. A validity check on these chunks is requested to force the partners to again check the node that lost the chunk - if all ok - this chunk is deleted (or moved to normal cache dir and follow same rules as a normal cache chunk)

5. If there is no chance to have this taken back the cache chunk holder is deemed to now be in control of this chunk and the partners sign a receipt for having stored a chunk (DISCUSS)

6. In the case of 5, the originator will have his swap chunk swapped back for no charge by the partners (he has not honored his deal by losing this chunk) - so when he is next online he can take the chunk back or lose it! If the chunk has more people looking at it (i.e. buffer packet shows more people looking at it) then the originator is penalized with a lost chunk notice on his rank packet.

L. Validity Check

To ensure data validity is constant on the network, validity checks are performed at regular intervals. For each chunk a vault holds it will carry out a validity check from time to time on the chunk. Chunks' health checks (or validity checks) are required to ensure data is actually in existance and also not been tampered with or damaged.

The validity check is as previous design except the timing of such. Validity checks can be initialized in several ways

1) A chunk holder is asked for a chunk (original chunk holder only). [done after chunk passed]
2) Any chunk swapped will mean the partners will do a validity check to ensure the new location is ok.
3) On startup - any chunk noted as lost in your rank packet - should initiate a check on that chunk.
4) At any time a corrupted chunk in the rank packet should initiate a check on all chunks you have (otherwise you will find it hard to save data - or the client attached to the vault will). This will make the corrupted one go to bottom of the list faster.

Each of the above is done in conjunction with a validity check for the chunk with longest time between a validity check! Any validity check for any chunk will mean you always check the longest unchecked chunk.

1) Implementation: This checking process is between partner nodes who all share a common chunk. i.e. a chunk copied 4 times will have 4 validity checking nodes. To allow the network to check the data without transmitting the actual data another method is described below. It should be noted that ALL data chunks' IDs are a hash of the content, therefore there is another check available for end user client node to validate the data they receive. The partner node validity check is described below:

1) Cn -> $r = rnd(chars)$, $res \equiv H(c1 + r)$
2) Cn takes chunk name (C1) and r and passes to S2 and S3 and requests they perform the same operation and pass the answer (result) back
3) If any node has false answer which differ from the other 2 then it has dirty data
4) The other nodes raise alerts (signed) and copy a new chunk to a new location
5) All other nodes storing data on Sn are now alerted to check
6) Each node will publish the chunk referenceIII-C2 if it has been lost for any reason.

VI. Clients

The maidsafe network can be considered as a three layer stack design as follows:

Filesystem/ GUI

maidsafe

Transport

The layers can be thought of as:

- Network Layer (Layer 1) is responsible for providing the network access.
- MAID Layer (Layer 2) is the storage area as far as a client can tell.
- FileSystem Layer (Layer 3) is the area of the system responsible for presenting the user with a filesystem from which to work with data. Initially this will be a FUSE implementation (Filesystem In User Space). For embedded systems where FUSE type is not available the file interface may be controlled via a traditional explorer type windows in the application itself rather than as another filesystem accessible from any native application.

A. Self Authentication

A user is authorised on the the network with 3 unique pieces of information: A username (U), a pin number (P), and a password (W). We use a method of hashing called Secure Hash Algorithm (SHA) in this case. SHA produces a code for given data, any small change in the data produces significantly different code. It is non reversible and consistent (given consistent data).

B. Create a User Identity

1) $H(H(U) + H(P)) \equiv \mathcal{MID}$
   $H(H(U) + H(P) + 1) \equiv \mathcal{SMID}$
2) $netgetV(\mathcal{MID}) \equiv NIL$
   && $netgetV(\mathcal{SMID}) \equiv NIL$ (to ensure uniqueness)

3) $d \equiv rnd(),\ R \equiv AESenc_{PBKDF2(U,P)}(d)$

4) Create *ANMID* as in III-D
5) $RSAsign_{(k)_{ANMID}}(R) \equiv R_s$
6) $netputD_{[MID]}(R_s)$ [*MID* packet]

7) Create *ANSMID* as in III-D
8) $RSAsign_{(k)_{ANSMID}}(R) \equiv Q_s$
9) $netputD_{[SMID]}(Q_s)$ [*SMID* packet]

10) Create *ANMAID* as in III-D
11) $RSAsign_{(k)_{[ANMAID]}}(p) \equiv x$ (sign public key with private key)
12) $netputD_{[\mathcal{MAID}]}(x)$ [*MAID* packet] - used later created on registration only.
13) $n \equiv H(H(U) + H(P) + H(d))$
14) $TMID \equiv AESenc_{[PKDBF2(W,rid)]}(DM\ of\ DA)$

15) Create ANTMID as in III-D
16) $\mathcal{TMID}_s \equiv RSAsign_{(k)_{[\mathcal{ANTMID}]]}}(\mathcal{TMID})$ (signed by *MAID3ID* to allow delete and separate from *MAID*)
17) $netputD(\mathcal{TMID}_s)$ [*TMID* PACKET]

C. Login to System

1) $H(H(U) + H(P)) \equiv \mathcal{MID}$
$H(H(U) + H(P) + 1) \equiv \mathcal{SMID}$

2) $netgetD(\mathcal{MID}) \equiv R_s$ on fail
$netgetD(\mathcal{SMID}) \equiv R_s$
3) $AESdec_{PBKDF2(U,P)}(R) \equiv d$
4) $n \equiv H(H(U) + H(P) + H(d))$(ensure unique)
5) $netgetD(n) \equiv \mathcal{TMID}_s$
6) $AESdec_{(PDBKDF2(W,rid))}(\mathcal{TMID}_s) \equiv \mathcal{TMID}$ Thus we have the Data Atlas

D. Log Out (or Update)

1) $d \equiv rnd(),\ R \equiv AESenc_{PBKDF2(U,P)}(d)$
2) $RSAsign_{(k)_{[ANsMID]}}(R) \equiv R_s$
3) $netputD_{[\mathcal{SMID}]}(R_s)$

4) $n \equiv H(H(U) + H(P) + H(d))$
5) $\mathcal{TMID}\equiv$
$AESenc_{[PDBKDF2(w,rid)]}(\mathcal{MAID}+k + DA)$
6) $\mathcal{TMID}_s \equiv RSAsign_{(k)_{[ANTMID]}}(\mathcal{TMID})$
7) $netput(\mathcal{TMID}_s)$
8) $RSAsign_{(k)_{[ANMID]}}(R) \equiv R_s$
9) $netputD_{[\mathcal{MID}]}(R_s)$

E. Self Encryption

$chunk_n(x) \equiv \overrightarrow{x}$n-chunking operation (with padding) $x_1 + x_2 + \ldots \equiv pad_n(x), |X_i \equiv n, |$
$concat(x_1, x_2 \ldots) \equiv \vec{x}$ chunking operation
$rot_n(\overrightarrow{x}) \equiv \overrightarrow{\tilde{x}}$rotating operation (shift left cyclic (put last chunk in first position))

1) $f_c \equiv file\,content,\ f_m \equiv file\,metadata,\ hm \equiv H(f_m)$
2) $\overrightarrow{c} \equiv chunk_n(f_c) \equiv c_1c_2 \ldots c_{m-1}c_m$ where r is fixed ref chunk same size as c with size n
3) $\vec{h} \equiv H(\vec{c}) =$
$H(c_1), H(c_2) \ldots H(c_{m-1})H(cm),\ rot(\vec{h}) \equiv$
$\vec{q},\ hf \equiv H(concat(\vec{q}))$
4) $r = hf + hf\,till\,r = n$
5) $\vec{e} \equiv (c_1 \bigoplus c_2)(c_2 \bigoplus c_3) \ldots (r \bigoplus c_1) \equiv \overrightarrow{c} \bigoplus rot(\overrightarrow{c})$
6) $\overrightarrow{f} \equiv$
$AESenc_{q1}(e_1)AESenc_{q2}(e_2) \ldots AESenc_{qm}(e_m) \equiv$
$AESenc\overrightarrow{q}(\overrightarrow{c} \bigotimes \overrightarrow{\tilde{c}}) \equiv$
$AESenc_{rot(}\overrightarrow{h}(chunk_n(fc) \bigoplus \widetilde{chunk}_n(fc))$
7) $\overrightarrow{f} \equiv selfenc_n(fc) \equiv$
$AESenc_{rot}(\overrightarrow{q})(chunk_n(fc) \bigoplus \widetilde{chunk}_n(fc)$

F. Data Maps

For a particular file
$D_{hf} \equiv \vec{h}X\vec{f} \equiv \{\hat{h}_1 : f_{1,}h_2 : f_2, \ldots\}$
Metadata data maps (may be multiple)

$$\hat{M}^1_{fm} \equiv \{1 : hf\} + \vec{hm}^1 \times$$
$$fm^1 \equiv \{1 : hf, hm^1_1 \vec{:} fm^1_1,\ hm^1_2 : fm^1_2, \ldots\}$$

$\ldots$

$$\hat{M}^p_{fm} \equiv \{1 : hf\} + \vec{hm}^p \times$$
$$fm^p \equiv \{1 : hf, hm^p_1 \vec{:} fm^p_1,\ hm^p_2 : fm^p_2, \ldots\}$$

Then
$D_n\hat{DF} \equiv \hat{D_{hf}} + \sum_{i=1}^{p} \hat{M^i_{fm}} \equiv \{1 : hf,\ hm^1_1 : fm^1_1, hm^1_2 : fm^1_2, \ldots hm^p_1 : fm^p_1, \ldots\}$This is the data map for this file
serialise data map into scalar (string) $concat(\vec{DF}) \equiv DF$

G. Data Atlas

Construct the data map for each file in the system in this way and create the serialised Data Atlas
$DF \equiv ser(\sum_{files} \hat{DF^i})$
Add data structure (key pairs etc. for all *ID*s (*MAID, MID, PMID* etc)
$\hat{P} \equiv \{ID_1 : k_1 : p_1 : \ldots ID_n : k_n : p_n\}$
$\hat{DF} \equiv DF + ser(\hat{P})$
Now Data Atlas is formed
$\hat{DA} \equiv h_{DA} \vec{\times} selfenc(\hat{DF}),\ h_{DA} \equiv \vec{chunk}_n(\hat{DF})$
$DA \equiv ser(\hat{DA})$
The Data Map (i.e. $DM = selfenc(DA)$) of the Data Atlas is what is stored to the TMID and encrypted with the user's password (W). Data Atlases are represented to the user as a file system with the file name and path containing the file hash which maps to the Data Atlas to retrieve the file content.

H. Public ID MPID

This is a different packet from others as its name must be the hash of the users chosen public name. Unlike other packets MPID public IDs should be stored with any node that communicates with it. Only allowing delete from these nodes with a revocation certificate [LATER]

1) $t \equiv userinput$, $\mathcal{MPID} \equiv H(t)$, $netget(\mathcal{MPID}) \equiv NIL$ (to ensure uniqueness)
2) $netget(\mathcal{MPID} + BUFFER) \equiv NIL$ (to ensure uniqueness of the buffer packet for sending messages later)
3) $RSAgenkeypair(MPID) \equiv (k, p)$
4) Create [ANMPID] from section III-D
5) $RSA_{sign_{(k)_{[ANMPID]}}}(p) \equiv p_s$ (signed to ensure only owner can delete)
6) $netput_{[\mathcal{MPID}]}(p_s)$ (*MPID* packet)
7) $netput_{[\mathcal{MPIDBUFFER}]}()$ (*MPIDBUFFER* packet)
8) Store k and *MPID* in the private key chain $\hat{P}$ of the DA for later use

The MPID buffer packet may contain a list of IDs allowed to post messages there. This is described in the messenger section. VI-L

I. Store Data on Network

S(1-3) = Storage node : Cn = client node (vault - identified by a PMID). We should be aware as a client saves chunks these are passed to his or her vault to process the following: (also see VI-K2)

1) Cn -> locates S1, S2, S3 to store a chunk c1, this requires finding out the correct rank as in **??** and also the correct timezones or whatever system is used to sub divide the network.
2) S1 replies OK you need to store a chunk (c2) from me to. This chunk is of the same size.
3) S1 gives the Sa2 and Sa3 nodes relating to C2 Cn as the replacement for him in the checking process. S1 updates the DHT by replacing values for key C2 to include Sa2 Sa3 and Cn being the new partners of this chunk.
4) Cn Sa2 and Sa3 are now all sharing C2.
5) S1 accepts Cn (C1) chunk.
6) Cn passes C1 to S2 and S3 giving them signed statements from Sa2 and ‘ $RSAsign_{(k)_{[PMID(Sa3)]}}(''CnhasstoredC2forpaymentofstoreC1'')$ [confirms Sa2 and Sa3 are happy with the swap]
7) S2 and S3 confirm C2 saved and OK, by checking DHT for C2 Values.
8) S2 and S3 accept the chunk copy and begin the checking process.
9) If Cn fails tests within 72 hours (this will be calculated based on number of chunks and network tuning) then the contract is nullified and S1 takes back the C2 checking process with S2 and S3. Cn then has to renegotiate a new location on another rank for C1. TODO add rank

J. Client Node Churn Resolution

As vaults from rank 0 to X are expected to be online for a determinable percentage of time (or else they are demoted / promoted) then their effect on churn is somewhat determinable. However, client nodes are not, and so they can have a detrimental effect on churn. To alleviate this, maidsafe issues client nodes with network addresses which have predeterminable information.

As maidsafe alters Kademlia to use a 512 bit number rather than a SHA-1 160 bit number as a network address, there is some leeway to play with. The solution is

- All client node’s addresses comprise an initial 352 (512-160) bits which are all hyphens
- All client node’s last 160 bits of address will be a standard SHA-1 hash 160 bits long.

To implement this, we have is_unique calls in the system to ensure we do not have collisions of data on the network, including creating IDs with key pairs etc. (at the system level). To ensure we never try and store a “- - - -” (352 bits long) address part we add this pattern to the is_unique() check. If this fails we assume a collision and create a new key pair etc. to hash for an ID (as in III-D) or alter the number of chunks in a data file. As this is a system fundamental then we should never have any client storing key value pairs of significance and their churn is much less damaging to the system.

K. Get Data from Network

To retrieve network data it’s a method of search for chunk and retrieve. It can be broken into small pieces as follows:

1) Get chunk name from DA (or in case of MID from H(U+P))
2) Find nearest node to this name.
3) Retrieve the value for the key of the name; this will provide at least 3 locations of duplicate chunks.
4) Ping each location.
5) As soon as first answer then this is fastest node, request the download from this location.

1) Delete Data from Network: The act of deleting data is again relatively simple, although some caveats are in place. Here we distinguish between data chunks and system chunks.

Data chunks are chunks on the network where the name (which is also the key in the DHT key value pair) is equivalent to the hash of the chunk content. This provides a safety check for chunk validity. System chunks on the other hand may not always meet this criteria, although all system chunks are signed by a PKI private key.

The process is as follows.

1) Vault gets a signed request to delete a chunk.
2) Vault checks this chunk by attempting to extract a signature.
3) If the signature matches the requesting signature the chunk is deleted.

4) The vault passes this request on to his peers who are also saving the chunk. It is up to the peers to delete the chunk, this vault's contract of saving is signed off and complete on receipt of this signed request.
5) If the signature does not match, the request is silently dropped, and a system alert is generated [ALERT_ATTEMPTED_UNAUTHORISED_DELETE].
6) If there is no signature then the content is hashed and compared with name, to ensure integrity.
7) If it is a valid data chunk the requester ID is removed from the [WATCHINGchunkname] list. (this list is an arbitrary figure long - say 10 entries, see VI-K2).
8) If data chunk and no IDs are left in the list then the chunk is marked for delete in 60 days.

2) Chunk Lifetimes and Duplicate Removal: As data chunks have a delete time of 60 days all clients MUST check their chunks every 45 days at most. When either a client checks his chunks OR on initial backup then the following holds true:

1) vault runs the store data on network process as in VI-I
2) If the chunks are already there OR this is a check on chunks.
3) The vault checks chunk availability (validity is done elsewhere).
4) Then vault checks the [WATCHINGchunkname] database to ensure either, there are at least 10 vaults listed or else it adds its own signed name to this list ($RSAsign_{[MAID]}(MAID - chunkname)$ (where MAID is MAID ID)
5)

L. Messenger System (Version 1.0)

The initial messaging system will not be real time but will use a buffer packet as described in III-B. This chunk is the hash of MPID+BUFFER and contains all user system messages.

- $RSAsign_{c1[privkey]}(RSAenc_{[c2pub\,key]}(\hat{DF})) \rightarrow C_2[MPIDBUFFER]$
- Where C1 and C2 are MPID nodes on the network sharing data.

M. Private Data Sharing

[rely on share⊂DA and a subset of $\subset \hat{Df}$]

1) $sharename \equiv Ns$
2) $DA \equiv unser(DA) \equiv \hat{DA}$
3) Select data maps as created in section VI-F
4) Select data Metadata from section VI-F
5) Select rights for each file and the share itself (addfiles/deletefiles/updatefiles/addusers/delusers/updateusers)
6) Concatenate
7) Serialise into share called Ns
8) Send sys message to public name you are sharing with (in a buffer message as in III-B)
9) Any share actions and all participants are messaged (file lock / updates - everything)

Therefore

(all communications via a buffer packet for each user)

1) $DA \equiv unser(DA) \equiv \hat{DA}$
2) Assume C1 is client 1 (MPID$_1$) C2 is (MPID$_2$)...
3) So $C1[DA] \equiv unser(DA) \equiv \hat{DA}$
4) $MSID_1 \subset DA$ ([defines share as a subset of DA)
5) $MSID_1 \equiv sharename$
6) $MSID_1 \rightarrow RSAsignC_{c1}(MSID_1)$
7) $C_2 \rightarrow C_1 RSAsign_{C2}(acept_{share}(C_2)$Share accepted
8) $C_1 \rightarrow MSID_1 \rightarrow C_2$ (share transferred (including data maps and Metadata maps)
9) If C$_1 OR], C_c add/delete/edit\, a\, file\, then\, C_2\, C_1 sends\, message\, to\, c$

message format

- $Sign_{[C_1]}(\hat{DF}) \rightarrow C_2(lock\, etc)$

VII. Additional Benefits

A. Duplicate Prevention

As we saw in section VI-E the data in this proposal is encrypted using information contained within the data. It is the data map that holds the keys and sources of any file or other piece of digital data.

It has been suggested in tests done at a local level that this could achieve a reduction in hard drive space required to store all the world's data. Tests by EMC on a local duplicate removal system (in a single LAN) showed saving 10 to 25 times the disk space used by data. maidsafe imagine this would mean a reduction in disk space required today - or a reclamation of disk space of somewhere between 60 - 90%, although only real world testing will provide the definitive answer. A recent study by datamonitor puts savings from 4 /1 to 89 /1 with an average at enterprise level of 20 /1 i.e. 20Gb can be backed up on 1 Gb space, therefore 95% savings.

B. Mobility

Log in to your data from any maidsafe enabled device from any Internet location.

C. Data-Loss Prevention

[search for known chunks]

D. Virus Resistance

[dirty data]

E. Web Site Logins with No Password

[PD signs requests using public or anon IDs]

VIII. Future Trends

The future of this type of technology is a bright one. A self managed digital ID which can spawn as many IDs as the system or the user decides it should (SHA512 hashes allow for $2x10^{154}$ results which equates to more hashes than atoms in the visible universe).

With a digital ID people could do many interesting things such as:

A. Perpetual Coin

A cryptographically secure digital ID with which any human can carry out interaction on the Internet as an anonymous but fully-authenticated entity. Current practice requires authentication by a third party. Logging in to PayPal is an example. With a maidsafe System, a member can log in from any networked computer in the world. The login process takes place on the local computer and involves only the user and the software. No password is ever transmitted. This ID can be used for anonymous self-authenticated voting, for file sharing and for commercial exchange.

Digital coin, like the gold coin of old, is a thing. It is a modular unit of software that can:

1) Be subdivided or combined, and be appropriately redenominated to any decimal place;
2) Remember and display the contractual agreement from which it originated;
3) Perform internal calculations according to pre-set formula, and,
4) Autonomously access the Internet for variables to input into said formula.

Digital coin's usefulness as a trading medium is unprecedented and unsurpassed. Digital coin can be instantly transferred via the Internet but never copied. Each unit of digital coin will have a unique identifier and be digitally signed by its owner while in that owner's possession. No one else will be able to spend it. This digital signature will be completely erased when the next owner's digital signature is imprinted, maintaining anonymity. Parties to transactions may be anonymous, using a Digital ID, verified by self-authentication. At all times the total amount of digital coin in existence will be finite, easily determined and publicly available.

[some maths]

B. Digitally Validated Voting

The ability to vote digitally and securely is inherent now with this design, but a major new situation is possible and this is vote validation.

[some maths]

C. Solid State Stand Alone Chunk nodes

Low power, 'plug and play' devices with little processing (vaults do not encrypt or decrypt) and attached to a network could expand the network's data storage capabilities automatically. To add or remove data storage it would be a matter of plugging in these devices.

D. Read Only Operating System

[benefits - no firewall no virus]

E. USB Computer

IX. Results

Results of testing so far are:

Time to self encrypt files (1Mb = 0.2secs) - table to follow

X. Conclusions

The Kademlia protocol will require to be updated as time goes by, this has already been tested[3] and should be included in the version 1.0 release of such a system.

References

[1] as described by Van Jacobson in this link below, August 30, 2006 HTTP://video.Google.com/videoplay?docid=-6972678839686672840

[2] PETAR MAYMOUNKOV AND DAVID MAZI'RESE Kademlia: A Peer-to-peer Information System Based on the XOR Metric {petar,dm}@cs.nyu.edu http://Kademlia.scs.cs.nyu.edu

[3] ANDREAS BINZENH¨ FER AND HOLGER SCHNABEL. Improving the Performance and Robustness of Kademlia-based Overlay Networks o University of W¨ rzburg, Institute of Computer Science u Chair of Distributed Systems, W¨ rzburg, Germany u Email: binzenhoefer@informatik.uni-wuerzburg.de

[4] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In Proc. ACM SIGCOMM (San Diego, 2001).

[5] ROWSTRON, A., AND DRUSCHEL, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In Proceedings of the 18th ACM Symposium on Operating Systems Principles (Oct. 2001).

[6] ZHAO, B., KUBIATOWICZ, J., AND JOSEPH, A. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, Apr. 2001.

[7] FRANK DABEK, M. FRANS KAASHOEK, DAVID KARGER, ROBERT MORRIS, ION STOICA. Wide-area cooperative storage with CFS MIT Laboratory for Computer Science chord@lcs.mit.edu http://pdos.lcs.mit.edu/chord/

[8] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. Freenet: A distributed anonymous information storage and retrieval system. In Proceedings of the Workshop on Design Issues in Anonymity and Unobservability (July 2000), pp. 46–66.

[9] KARGER, D., LEHMAN, E., LEIGHTON, T., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In Proceedings of the 29th Annual ACM Symposium on Theory of Computing (May 1997), pp. 654–663.

[10] JINYANG LI, JEREMY STRIBLING, THOMER M. GIL, ROBERT MORRIS, M. FRANS KAASHOEK. Comparing the performance of distributed hash tables under churn: MIT Computer Science and Artificial Intelligence Laboratory {jinyang, strib, thomer, rtm, kaashoek}@csail.mit.edu

[11] Jun Xu. On the fundamental tradeoffs between routing table size and network diameter in peer-to-peer networks. In Proceedings of the IEEE Infocom, March 2003.

[12] Philip Hawkes , Michael Paddon , and Gregory G. Rose, On Corrective Patterns for the SHA-2 Family, Qualcomm Australia, Level 3, 230 Victoria Rd, Gladesville, NSW 2111, Australia

[13] Yunhong Gu, Robert L Grossman, UDT: UDP-based data transfer for high-speed wide area networks

A Scottish Engineer and innovator who has spent the last 12 years researching ways to make computers function in a better way. Listed as sole inventor on 11 patent submissions and designer of one of the worlds largest private networks. He rarely stops working on algorithms and problems looking for better ways. He also enjoys sailing and the company of imaginative people.