Autonomous Network

David Irvine*, Fraser Hutchison†, Steve Mücklisch‡
MaidSafe.net, 72 Templehill, Troon, South Ayrshire, Scotland, UK. KA10 6BE.
*david.irvine@maidsafe.net, †fraser.hutchison@maidsafe.net, ‡steve.muecklisch@maidsafe.net

First published September 2010.

***Abstract*—Autonomous networks are self-healing, self-managing and most importantly independent of human interference. Such networks will be able to be developed in a way that avoids wasting effort on maintaining even simple mechanisms such as storage, scalability and data retention. Systems like these will quickly extend to providing a method of highly scalable platforms that can accommodate real time transactional logic. A working example of an autonomous network is outlined in this paper.**

***Index Terms*—security, freedom, privacy, authentication, encryption, autonomous**

CONTENTS

I. INTRODUCTION

COMPUTING capability has dramatically increased in recent years, particularly in terms of processing power and available inter-connectivity of devices via the Internet. This has allowed the creation of remarkable technology that would have been considered "space age" or the works of a science fiction writer until comparatively recently. Devices and applications that allow world mapping, video conferencing on the move, instant recording and sharing of nearly any type of information from text to high definition video, the ability to know where you are anywhere in the world and to locate friends, information and tools to help calculate results of questions is all without doubt amazing.

There is a huge downside though; and that is the ability to store such data in secure, accessible and reliable locations without the requirement for human organization. Today's cloud computing paradigm is an attempt to deliver such a system, but does so in a manner that is more hype than fact. Actually, today's cloud computing is the antithesis of actual or true cloud computing and is merely a marketing attempt to persuade us that an autonomous network has been created.

This paper delivers a true platform for cloud computing, that ensures human intervention is forbidden and in fact circumvents any attempts to manipulate data or processes.

A. Prerequisites

Prior to reading this paper, it is highly advisable that the reader fully understand *Self Encrypting Data [1]* and *Peer to Peer Public Key Infrastructure [2].* These papers detail some components that will be used with the system presented in this paper; in fact the Public Key Infrastructure (PKI) described in *Peer to Peer Public Key Infrastructure [2]* is a fundamental requirement of the autonomous network presented here.

B. General Conventions

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.
- $\mathsf{H} \equiv$ Hash function such as SHA, MD5, etc.
- $\mathsf{PBKDF2}_{[\mathsf{Passphrase}][\mathsf{Salt}][\mathsf{IterCount}]} \equiv$ Password-Based Key Derivation Function or similar
- $\mathsf{XXX}_{\mathsf{priv}}$, $\mathsf{XXX}_{\mathsf{pub}} \equiv$ Private and public keys respectively of cryptographic key pair named XXX
- $\mathsf{AsymEnc}_{[\mathsf{K}_{\mathsf{pub}}]}(\mathsf{Data}) \equiv$ Asymmetrically encrypt Data using $\mathsf{K}_{\mathsf{pub}}$
- $\mathsf{AsymDec}_{[\mathsf{K}_{\mathsf{priv}}]}(\mathsf{Data}) \equiv$ Asymmetrically decrypt Data using $\mathsf{K}_{\mathsf{priv}}$
- $\mathsf{Sig}_{[\mathsf{K}_{\mathsf{priv}}]}(\mathsf{Data}) \equiv$ Create asymmetric signature of Data using $\mathsf{K}_{\mathsf{priv}}$
- $+ \equiv$ Concatenation

C. Specific Conventions

1) Network Identities: In *Peer to Peer Public Key Infrastructure* [2], the ability to create cryptographic key pairs that are both secure and mathematically deduced is shown. Several of these cryptographic key pairs with specific roles are introduced here, as well as additional system-specific components:

- DHT: This paper assumes the use of a key addressable network, and in this case a Kademlia Distributed Hash Table (DHT) is assumed. There is no requirement for this to be restrictive in any way. The DHT can be replaced with any key addressable network. It is also assumed there is no issue with Network Address Translation (NAT) and all Nodes can freely communicate. In this paper, it is assumed all cryptographic keys are signed on the system and only the signatory identity may amend or delete a value. There may be rules as to which identity can store certain information in specific locations. Such an implementation can be found at http://code.google.com/p/maidsafe-dht This implements a Kademlia-based network and is described in *MaidSafe Distributed Hash Table* [3]
- $\mathsf{K} \equiv$ The DHT replication factor.
- ANMAID $\equiv$ The (ANonymous Maidsafe Anonymous IDentification) packet is the root of a chain that reaches as far as the PMID (below). This is a pure packet, which means that the identity is created as $\mathsf{H}(\mathsf{ANMAID}_{\mathsf{pub}} + \mathsf{Sig}_{[\mathsf{ANMAID}_{\mathsf{priv}}]}(\mathsf{ANMAID}_{\mathsf{pub}}))$. This identity is never stored on the network, thereby even something encrypted with this public key is not on the network at all. It is assumed that this identity is otherwise maintained securely by a system such as that described in *maidsafe: A new network paradigm* [4]
- MAID $\equiv$ The (Maidsafe Anonymous IDentification) packet has as its identity $\mathsf{H}(\mathsf{MAID}_{\mathsf{pub}} + \mathsf{Sig}_{[\mathsf{ANMAID}_{\mathsf{priv}}]}(\mathsf{MAID}_{\mathsf{pub}}))$. This packet is stored on the network with its identity as the key (as described in *Peer to Peer Public Key Infrastructure* [2]). The MAID can be used by another Node or component (even a person) to act on the network with the same authority which the PMID has. This is an important distinction from many other such systems. The ANMAID is the revocation key for this identity.
- PMID $\equiv$ The (Proxy Maidsafe IDentification) packet has as its identity $\mathsf{H}(\mathsf{PMID}_{\mathsf{pub}} + \mathsf{Sig}_{[\mathsf{MAID}_{\mathsf{priv}}]}(\mathsf{PMID}_{\mathsf{pub}}))$. The $\mathsf{PMID}_{\mathsf{priv}}$ has to be stored on the machine that runs the Node process. This is a potential security risk and therefore the system requires that the PMID identity be restricted in capability as far as possible. The MAID is the revocation key for this identity.

2) Storing and Deleting Encrypted Data Chunks: The process for storing chunks of data is described in more detail at IV-B. However, the storing process essentially involves a group of Nodes brokering a deal between the requesting Node (called the client) and the responding Node (called the vault). The deal is validated by means of a StoreContract which contains an InnerContract which itself contains a SignedSize as detailed below. Deleting a chunk also involves a SignedSize.

- SignedSize is a serialisable data object containing:
  1) ChunkSize in bytes
  2) $\mathsf{Sig}_{[\mathsf{PMID}_{\mathsf{priv}}]}(\mathsf{ChunkSize})$
  3) PMID identity
  4) $\mathsf{PMID}_{\mathsf{pub}}$
  5) $\mathsf{Sig}_{[\mathsf{MAID}_{\mathsf{priv}}]}(\mathsf{PMID}_{\mathsf{pub}})$

  In the case of storing a chunk, the PMID is owned by the client; for deleting a chunk, it is owned by a Chunk Info Holder (covered below).
- InnerContract is serialisable data object containing:
  1) ACK || NACK (agreement to deal)
  2) SignedSize
- StoreContract is a serialisable data object containing:
  1) InnerContract
  2) $\mathsf{Sig}_{[\mathsf{PMID}_{\mathsf{priv}}]}(\mathsf{InnerContract})$
  3) PMID identity
  4) $\mathsf{PMID}_{\mathsf{pub}}$
  5) $\mathsf{Sig}_{[\mathsf{MAID}_{\mathsf{priv}}]}(\mathsf{PMID}_{\mathsf{pub}})$

  In the case of storing or deleting a chunk, the PMID is owned by the vault.

II. Overview

The answer to the current issues as described in **??** is to redesign networks to require no central control and by implication, no servers as we currently know them, whether centralised or distributed. To achieve this there are several important requirements:

1) Encryption of data units to a very high level. This is described in *Self Encrypting Data* [1].

2) A method of validation of Nodes as described in *Peer to Peer Public Key Infrastructure* [2].
3) The ability to randomly select Nodes based on a mathematically distributed algorithm that can identify groups of Nodes to act as independent certification for network actions (and in some cases arbitration).
4) A method of distributed and verifiable measurement of a Node's capability.
5) A system of distribution of data to ensure geographic protection of replicated information.[1]

One improvement, but not requirement, is the application of a ranking system to allow a granular approach to Node capability and therefore *cost* to the network.

The thinking involved in such a system is very similar to the thought process behind Kademlia itself in many ways. It is also apparently simple at first glance, but much more complex when deriving the detail and particularly when considering alterations to logic and the consequences of such.

III. Components

A. Kademlia Component

1) Knode: Each Node on the network has an instance of a Kademlia node (knode) running. Kademlia is mainly used as the means of finding peers, not for actually storing/retrieving data; this is left to another layer that acts as an overlay to the DHT overlay network itself.

2) Kademlia RPCs: Please see Table I on page 4.

B. Overlay Component

1) Node: Nodes are run as a separate process and largely manage themselves. At the moment, when the software is installed and run, a daemon / service runs which starts an "unowned" Node. This listens on the local network for owner control RPCs.

2) Node RPCs: Please see Tables II - VII on pages 4 - 5

3) Roles: Nodes have three major concurrent roles in the network: storing and maintaining encrypted data chunks, maintaining various information about chunks, and maintaining account information about other Nodes (see III-C below). Nodes are referred to as Chunk Holders, Chunk Info Holders or Account Holders respectively depending on which role we wish to emphasize at the time. (Note however that any single Node is likely to be all of these for multiple chunks and peers simultaneously).

C. Accounts

In order to maintain fairness on the network, each Node has an associated account, the name of which can be derived from the Node's PMID identity. The PMID identity should not however be derivable from the account name.

This account details the amount of storage space the Node offers to the network, the space the Node's owner has used on the network, and the space the Node has actually given to the network. The owner uses network space by storing files to the network; the Node gives space by storing and maintaining other users' chunks.

Nodes can only store new data to the network if their account will remain "in credit" (i.e. space offered > space used) after the store operation. Users, regardless of account status, will always be able to get data from then network.

A ranking mechanism which takes these figures (and many other metrics) as inputs will eventually be implemented. Generally, rank will increase with the amount of space offered and given.

D. Chunk Holders

For any chunk there should be at least two copies on the network, ideally four or more, each stored on separate Nodes. A Node which stores such a copy is referred to as a Chunk Holder. It is in a Node's interest to store chunks since its rank metric is increased for doing so. Chunks can be moved from one Node to another; as this happens the two corresponding accounts are debited and credited. Frequently accessed chunks can be cached on Nodes other than the official Chunk Holders, to speed up retrieval along a lookup path.

E. Chunk Info Holders

1) Reference Lists: In order to be able to locate a given chunk, a list of Chunk Holders' identities (referred to as a Reference List) is kept on the network. The Reference List is held and maintained by the K Nodes whose IDs are closest to the name of the chunk (see Figure 1). Each Node in this group is referred to as a Chunk Info Holder.

A Reference List entry comprises the Chunk Holder's Node ID, and the time it was last contacted by the Chunk Info Holder. This allows the Chunk Info Holder to return only active Chunk Holders in response to a GetChunkReferences request, yet keep details of stale Chunk Holders in case they come back online.

2) Watch Lists: The same group also holds a list of peers that are "watching" a chunk (referred to as a Watch List), which means they stored the chunk at some point[2] and are now interested in retaining it on the network (see Figure 2).

Watch Lists are currently limited to 250 entries[3]; once filled, new watchers are only registered by increasing a counter and including their ID in a checksum. A chunk can only be removed from the network once the corresponding Watch List is empty, the counter is zero, and the checksum indicates that all watchers have subsequently removed themselves (see Figure 3). So even if the Watch List is empty and the counter is zero, if something went wrong during the Chunk Info's lifetime (e.g. a Node which was never added to the Watch List requested to be removed from it) the checksum helps detect

[1]Here we assume replication as opposed to forward error correction. This is a debate in computer science and may rage on for a while. In this paper we assume a more binary approach to data safety, it is either secure or not as opposed to possibly secure in the forward error correction model.

[2]If the chunk pre-existed on the network, they may not have actually uploaded the chunk themselves.

[3]This figure is currently arbitrary and will be calculated based on several network parameters as the logic improves.

RPC NAME	REQUEST FIELDS	RESPONSE FIELDS	PURPOSE
Ping	Ping	ACK ǁ NACK	Check peer is connected.
FindValue	Key	SignedValue (repeated)	Get all values stored under Key.
FindNode	Key	NodeContactDetails (repeated)	Find K closest Nodes to Key.
Store	Key, SignedValue, TTL, RequestSignature	ACK ǁ NACK	Store SignedValue under Key for duration of TTL. RequestSignature allows validation of ID of requester.
Delete	Key, SignedValue, RequestSignature	ACK ǁ NACK	Delete SignedValue under Key. RequestSignature allows validation of ID of requester.
Update	Key, OriginalSignedValue, NewSignedValue, TTL, RequestSignature	ACK ǁ NACK	Update OriginalSignedValue under Key with NewSignedValue for duration of TTL. RequestSignature allows validation of ID of requester.
DownList	NodeContactDetails	ACK ǁ NACK	Suggest removing Node from routing table. Confirm Node is disconnected by sending Ping to Node before removal.

Table I
KADEMLIA RPCS

RPC NAME	REQUEST FIELDS	RESPONSE FIELDS	PURPOSE
StorePrep	ChunkName, SignedSize, RequestSignature	StoreContract, ResponseSignature	Make initial agreement between client and vault to store data chunk.
StoreChunk	ChunkName, Data, PMID, PMID_{pub}, $\text{Sig}_{[\text{MAID}_{\text{priv}}]}(\text{PMID}_{\text{pub}})$, RequestSignature	ACK ǁ NACK	Store data chunk. PMID belongs to client.
GetChunk	ChunkName	ACK ǁ NACK, Data	Get data chunk.
CheckChunk	ChunkName	ACK ǁ NACK	Check if the recipient has the chunk.
DeleteChunk	ChunkName,SignedSize, RequestSignature	ACK ǁ NACK	Delete data chunk. RequestSignatureformed using Chunk Info Holder's PMID.
ValidityCheck	ChunkName, RandomData	ACK ǁ NACK, HashContent	Ensure data chunk is uncorrupted.
CacheChunk	ChunkName, Data, PMID, PMID_{pub}, $\text{Sig}_{[\text{MAID}_{\text{priv}}]}(\text{PMID}_{\text{pub}})$, RequestSignature	ACK ǁ NACK	Cache data chunk. PMID belongs to client.

Table II
NODE RPCS (CHUNK MANAGEMENT)

RPC NAME	REQUEST FIELDS	RESPONSE FIELDS	PURPOSE
GetChunkReferences	ChunkName	ACK ǁ NACK, Refs(repeated)	Get Node IDs of holders of data chunk.
AddToWatchList	ChunkName,SignedSize, RequestSignature	ACK ǁ NACK, UploadCount, TotalPayment	Request to be added to the list of watchers for data chunk. RequestSignature formed using client's PMID.
RemoveFromWatchList	ChunkName, PMID, PMID_{pub}, $\text{Sig}_{[\text{MAID}_{\text{priv}}]}(\text{PMID}_{\text{pub}})$, RequestSignature	ACK ǁ NACK	Request to be removed from the list of watchers for data chunk. RequestSignature formed using client's PMID.
AddToReferenceList	ChunkName, StoreContract, RequestSignature	ACK ǁ NACK	Request to be added to the list of Chunk Holders for data chunk. RequestSignature formed using vault's PMID.

Table III
NODE RPCS (CHUNK INFORMATION MANAGEMENT)

RPC NAME	REQUEST FIELDS	RESPONSE FIELDS	PURPOSE
AmendAccount	AmendmentType, AccountPMID,SignedSize, ChunkName(optional)	ACK ǁ NACK	If the AmendmentType is space offered, the request comes from a client and doesn't include a ChunkName. Otherwise, the request comes from a Chunk Info Holder and relates to storing or deleting a chunk.
ExpectAmendment	AmendmentType, ChunkName, PMID, PMID_{pub}, $\text{Sig}_{[\text{MAID}_{\text{priv}}]}(\text{PMID}_{\text{pub}})$, RequestSignature, AmenderPMIDs (repeated)	ACK ǁ NACK	Allows Account Holders to anticipate a forthcoming AmendAccount RPC from each of the K Chunk Info Holders (indicated in AmenderPMIDs).
AccountStatus	AccountPMID, SpaceRequested, PMID_{pub}, $\text{Sig}_{[\text{MAID}_{\text{priv}}]}(\text{PMID}_{\text{pub}})$, RequestSignature,	ACK ǁ NACK, SpaceOffered, SpaceGiven, SpaceTaken, AmendmentResults (optional, repeated)	Get the current status of a Node's Account. If the requester is the Account Owner, a list of all account amendments since the last AccountStatus request was made is returned also.

Table IV
NODE RPCS (ACCOUNT MANAGEMENT)

RPC NAME	REQUEST FIELDS	RESPONSE FIELDS	PURPOSE
GetSyncData	PMID, PMID_{pub}, $\text{Sig}_{[\text{MAID}_{\text{priv}}]}(\text{PMID}_{\text{pub}})$, RequestSignature	ACK ‖ NACK, VaultAccountSet, ChunkInfoMap, VaultBufferPktMap	Used by a Node to retrieve serialised containers of data from close peers which it should also be responsible for holding.
GetAccount	AccountPMID, PMID, PMID_{pub}, $\text{Sig}_{[\text{MAID}_{\text{priv}}]}(\text{PMID}_{\text{pub}})$, RequestSignature	ACK ‖ NACK, VaultAccount	Used by a Node to retrieve an individual account from close peers which it should also be responsible for holding.
GetChunkInfo	ChunkName, PMID, PMID_{pub}, $\text{Sig}_{[\text{MAID}_{\text{priv}}]}(\text{PMID}_{\text{pub}})$, RequestSignature	ACK ‖ NACK, VaultChunkInfo	Used by a Node to retrieve info relating to an individual chunk from close peers which it should also be responsible for holding.
GetBufferPacket	BufferPacketName, PMID, PMID_{pub}, $\text{Sig}_{[\text{MAID}_{\text{priv}}]}(\text{PMID}_{\text{pub}})$, RequestSignature	ACK ‖ NACK, VaultBufferPacket	Used by a Node to retrieve an individual buffer packet from close peers which it should also be responsible for holding.

Table V
NODE RPCS (SYNCHRONISATION OF MANAGEMENT DATA)

RPC NAME	REQUEST FIELDS	RESPONSE FIELDS	PURPOSE
CreateBP	BufferPacketName, Data, PMID, PMID_{pub}, $\text{Sig}_{[\text{MAID}_{\text{priv}}]}(\text{PMID}_{\text{pub}})$, RequestSignature	ACK ‖ NACK	Create a buffer packet.
ModifyBPInfo	BufferPacketName, Data, PMID, PMID_{pub}, $\text{Sig}_{[\text{MAID}_{\text{priv}}]}(\text{PMID}_{\text{pub}})$, RequestSignature	ACK ‖ NACK	Modify a buffer packet's control information (e.g. set permissions).
GetBPMessages	BufferPacketName, PMID, PMID_{pub}, $\text{Sig}_{[\text{MAID}_{\text{priv}}]}(\text{PMID}_{\text{pub}})$, RequestSignature	ACK ‖ NACK, Messages (repeated)	Retrieve a buffer packet's messages.
AddBPMessage	BufferPacketName, Data, PMID, PMID_{pub}, $\text{Sig}_{[\text{MAID}_{\text{priv}}]}(\text{PMID}_{\text{pub}})$, RequestSignature	ACK ‖ NACK	Add a message to a buffer packet.

Table VI
NODE RPCS (BUFFER PACKET MANAGEMENT)

RPC NAME	REQUEST FIELDS	RESPONSE FIELDS	PURPOSE
SetLocalVaultOwned	PMID_{pub}, $\text{PMID}_{\text{priv}}$, $\text{Sig}_{[\text{MAID}_{\text{priv}}]}(\text{PMID}_{\text{pub}})$, SpaceOffered	ACK ‖ NACK	Take ownership of an unowned Node.
LocalVaultOwned	Owned	ACK ‖ NACK	Query a Node's owned status.
VaultStatus	StatusRequest	StatusResponse	Used to poll a vault for its current status.

Table VII
NODE RPCS (MISCELLANEOUS)

it, in which case the chunk would just be kept indefinitely or at least for a very long time (several years).[4]

A Watch List entry comprises the watcher's Node ID and a flag to indicate if the entry can be deleted. The flag allows retention of watchers who have requested their removal from the Watch List, but who are still providing a "payment" for the chunk. This permits these ex-watchers to be proper recompensed eventually (once new watchers add themselves to the Watch List).

Because of this wealth of information, Chunk Info Holders are also responsible for monitoring the number of active Chunk Holders on the network, and triggering chunk validity checks, duplication, repair or removal as required. In future, Chunk Info Holders may also collect various statistics, such as the number of watchers over certain time periods and the amount of requests for references to a chunk.

[4]This is a situation that will very likely be improved as further research should yield a better algorithm for the removal of stale data. Any data that is addressed by the hash of its content will only be deleted through a necessity of reclaiming space.

3) Waiting Lists: Nodes requesting addition to a Watch List are added to a Waiting List until payment for the chunk either succeeds or fails, and if duplicate chunk copies are needed, until the new Chunk Holders add themselves to the Reference List.

F. Account Holders

The K Nodes whose IDs are closest to the name of a Node's account are called Account Holders. Full account information can only be retrieved by the owner of the account, for privacy reasons. Other Nodes are only allowed to confirm whether the owner is authorised to perform a storage operation by checking for enough available space.

IV. MAIN PROCESSES

A. Joining the Network

After successfully joining the network on the Kademlia layer, an account needs to be created for the client's Node in order to specify how much space is to be offered to

Figure 1. Add to Reference List

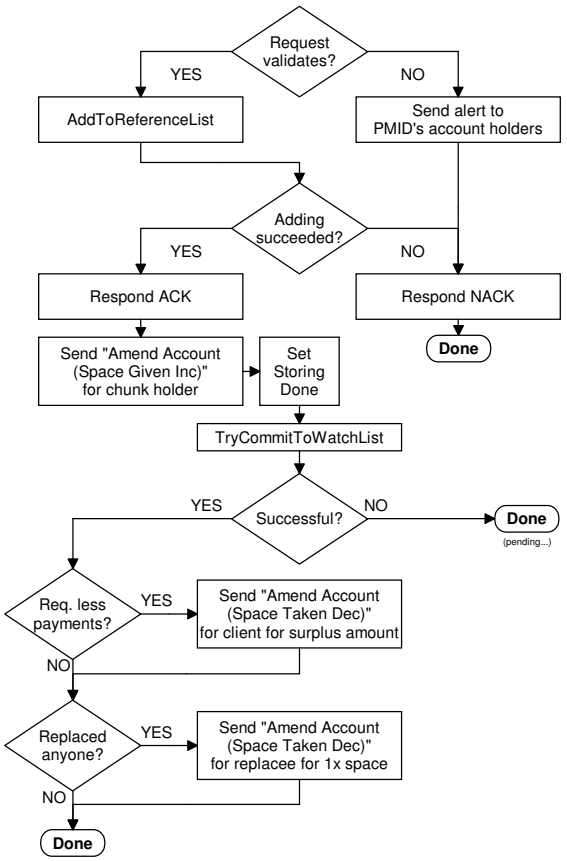


Figure 2. Add to Watch List

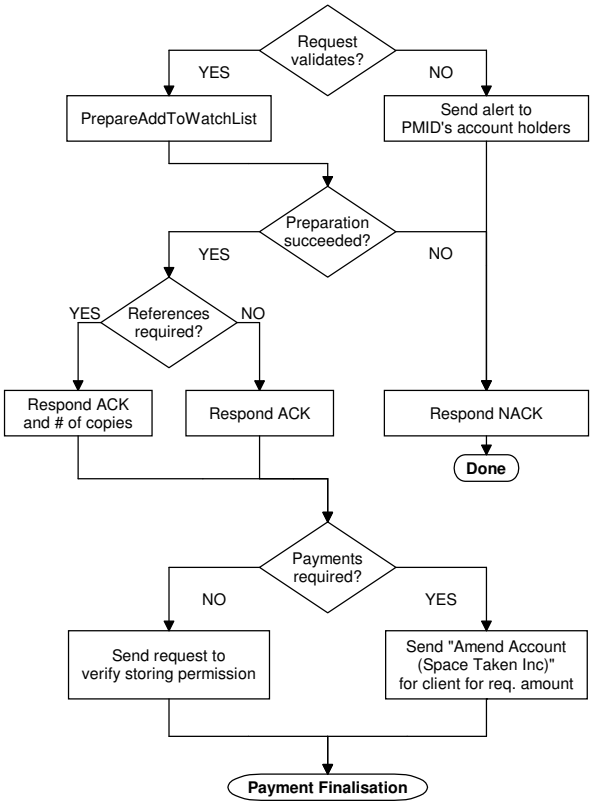
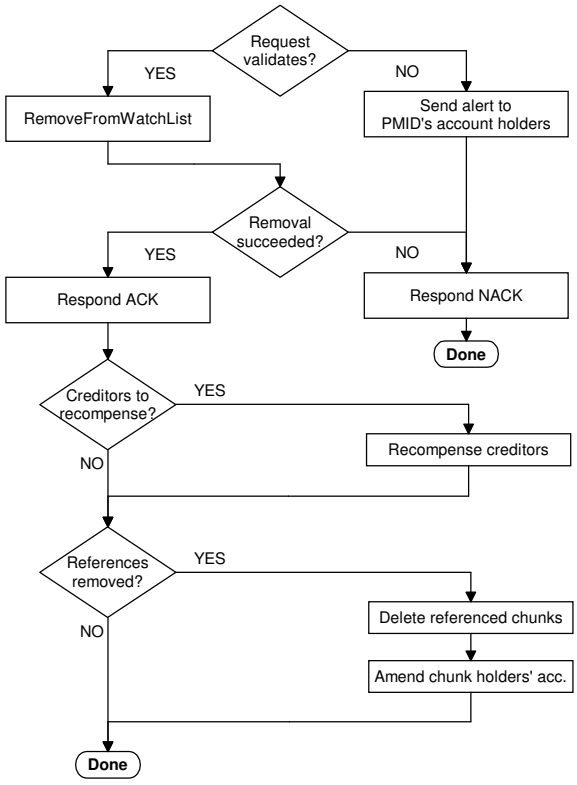


Figure 3. Remove from Watch List



the network. Only once enough of the K Account Holders are aware of this account can further operations demanding payments succeed. Fraudulent account creations are intended to be detected by peer Nodes in future operations, e.g. when storage operations could not be completed.

B. Storing a Chunk

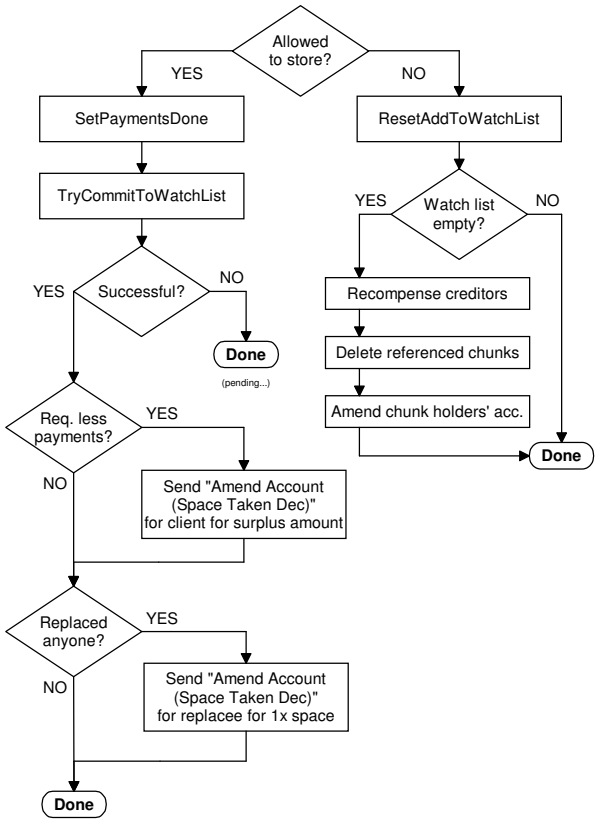
To store a chunk, a client first has to ensure there is enough space in its account (see III-C). It then looks up the K Chunk Info Holders and requests to be added to the Watch List for that chunk.

Once a Chunk Info Holder has received and validated an AddToWatchList request, it adds the ID of the requesting peer to the Waiting List. In response, the client gets informed how many copies of the chunk need to be uploaded. This number depends on how many copies of the chunk are already on the network and is derived from the desired minimum amount of copies (kMinChunkCopies[5]), usually lower to spare the client from excessive uploading.

Each Chunk Info Holder then proceeds to look up the K Account Holders of the client's account, and sends each a request to deduct the required payment (which equals the storage space needed) from the client's account. If the chunk didn't exist on the network, the client is asked for a payment kMinChunkCopies times the size of the chunk, independent of how many chunk copies were requested to be uploaded. If the chunk did pre-exist, then no uploads are required and payment

[5]Currently a system wide constant

Figure 4. Payment Finalisation



depends on the state of the Watch List. If the Watch List is not full, only a single payment is required, otherwise storing is free. If this payment process triggered by the Chunk Info Holders fails, for example because the client's account isn't sufficiently funded, all previous operations are undone.

Once the payment process is complete, the corresponding entry in the aforementioned Waiting List is flagged as having been paid for. If the uploading of chunk copies was stipulated, the entry remains in the Waiting List until the new Chunk Holders have contacted the Chunk Info Holders. Otherwise the entry is removed from the Waiting List.

If the client was told to upload chunk copies, it contacts one Node per requested copy to negotiate a store contract. If the peer Node is able to validate the request and can provide enough physical storage space, the client can then proceed to upload the data chunk. Upon successful receipt of the chunk, the new Chunk Holder(s) look up and contact the Chunk Info Holders, requesting inclusion in the Reference List. This also causes the corresponding entry in the Waiting List to be flagged as having completed the storage requirements. The Chunk Info Holders then look up and request the amendment of each of the new Chunk Holders' accounts to reflect the additional space used by the network.

Once an entry in a Waiting List indicates successful completion of payments (and storage if applicable), the corresponding client can be committed to the Watch List as an actual "watcher". At this point, any superfluous payments that occurred from race conditions will be refunded. This concludes the storing process and the chunk is at this point safely stored on the network.

C. Retrieving a Chunk

To retrieve a chunk, the client has to look up and contact the Chunk Info Holders to acquire the current list of active Chunk Holders. This is done through a Kademlia FindValue, which would return the ID of a peer that holds a cached copy of the chunk, or an empty value requiring the Client to ask for the Chunk Holders directly. The first Chunk Holder to confirm that it has the required chunk is used to retrieve the chunk from.

D. Deleting a Chunk

If clients decide they don't want to keep a file any more, they remove themselves from the Watch Lists of the corresponding chunks. Once a Watch List becomes empty, i.e. no one is watching that particular chunk any more, it can be deleted from the network. This means traversing the current Reference List and calling a remote delete operation on the Chunk Holders, recompensing all the peers that made a payment to keep the minimum number of chunks on the network, as well as decrease the space given value in the (former) Chunk Holders' accounts.

V. Maintaining Network Health

A. Validity Checks by Chunk Info Holders

The Chunk Info Holders will be responsible for triggering validity checks on a chunk. When a check is due, a Chunk Holder will be chosen at random and told to initiate a validity check. This is a relatively simple process; for example, given a chunk named as ABC whose content is $\mathsf{Content}_{\mathsf{ABC}}$, a Chunk Holder will do the following:

1) Get the contact details of the other Chunk Holders for ABC
2) For each Chunk Holder, send a ValidityCheck request (see Table II on page 4) with a piece of random data (different data for each Node) in the RandomData field
3) Calculate $\mathsf{H}(\mathsf{Content}_{\mathsf{ABC}} + \mathsf{RandomData})$ and retain this as Result
4) Each Chunk Holder's reply should contain $\mathsf{H}(\mathsf{Content}_{\mathsf{ABC}} + \mathsf{RandomData})$ in the HashContent field
5) Confirm each Chunk Holders' HashContent field matches the corresponding Result.

Any discrepancies are reported to the Chunk Info Holders and the Node at fault is informed. If this Node is unable to rectify the fault (e.g. by requesting a fresh copy from a good Chunk Holder and successfully passing a subsequent validity check) it is removed from the Reference List for the chunk and its account is amended to reflect less space given. This is a part of the system where rank will be adversely affected when it is put in place. Bad Chunk Holders will lose rank very quickly.

The triggers from the Chunk Info Holder to the Chunk Holder are time-based and will initially start at 2 minutes doubling every time to 20 hours. Any failure will reset the schedule.

B. Relocation Based on Rank

Chunks will be relocated when the Chunk Info Holder notes that any Chunk Holder has lost or gained rank in the system.

C. Validity Checks by Chunk Holders

On request for a chunk to be delivered to another Node, a Chunk Info Holder will trigger an internal validity check.

D. Geographic Relocation of Chunks

A Chunk Info Holder that triggers a validity check will query their own routing table for Chunk Holders of the same status as the current Chunk Holders, but with a longer Round Trip Time (RTT). On finding a Node that is further away the data will be moved, preferably with a chunk swap. This swap will be negotiated via the Chunk Info Holder for the remote Node in question.

E. Caching of Chunks

When loading a data chunk from the network, if the Kademlia lookup phase took more than one iteration, a copy of the chunk will be cached upon successful completion of the chunk's retrieval. The CacheChunk RPC will be sent to the last Node contacted during the Kademlia lookup that did not have the chunk. Cached chunks should be located in a cache directory on the Node and be part of a First In First Out (FIFO) queue. This queue should only require a chunk removal when the Node requires the space.

This simple mechanism ensures data integrity is strengthened, but more importantly shares the load of any Node that hosts interesting or popular chunks. There are many other advantages, such as resistance to denial of service attacks or distributed denial of service attacks, or if web based data is stored (such as a web site) then the more popular it is the more responsive the network will be when data is requested from this web site. This is almost the opposite of the case in today's World Wide Web, but is, again, more logical.

VI. Conclusions

There are several dramatic improvements over contemporary paradigms described in this single paper. The ideas presented here allow the creation of a serverless network which gives users a chance for the first time ever to retain complete control of their own security and personal information.

They also yield massive potential gains in terms of space (via data deduplication), data integrity (via validity checks and autonomous data repair), resilience to churn and attacks (via validity checks, ranking, geographic relocation of chunk copies), and scalability and transfer rates (via chunk caching).

It is no overstatement to say that the autonomous network described in this paper represents a highly significant step forward for the world of computing.

References

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