

Membrane - Distributed File Backup

Dominic Hauton

April 26, 2017

Abstract

An application in which users can join a large public network of intelligent storage agents to trade their available file space. This will allow the user to backup and version files safely on potentially untrustworthy computers, increasing availability and redundancy. The hosts will act as intelligent agents, and trade space when they communicate with one another.

Declaration

I declare that...

Acknowledgements

With thanks to my supervisor, Russell Bradford

Contents

1	Introduction	5
1.1	Project Goals	5
1.2	Need for Membrane	5
1.3	Challenges	6
1.4	Modules	6
1.5	Conclusion	6
2	Literature Review	7
2.1	History of Problem	7
2.1.1	Rsync	7
2.1.2	Git	7
2.1.3	Bittorrent	8
2.1.4	Resilio	8
2.1.5	Storj	8
2.2	Peer Admission	9
2.2.1	Bootstrapping	9
2.2.2	Peer Exchange	9
2.2.3	Shard Discovery	10
2.2.4	Dynamic IP Address	10
2.3	Data Allocation on External Nodes	11
2.3.1	Negotiation	11
2.3.2	Negotiation Logic	11
2.3.3	Negotiation Tactics	11
2.3.4	Practical Negotiation	12
2.3.5	Service Level Agreements	13
2.4	Trust and Reputation	13
2.4.1	Trust and Reputation Attacks	14
2.5	Communication	14
2.5.1	Agent Communication Language	14
2.5.2	Ontology	15
2.6	Peer-to-Peer Connection	15
2.7	Distributed File System	16
2.8	Authentication	17
2.9	Proof of Ownership	18
2.10	Conclusion	18
3	Analysis	19
3.1	User Survey	19
3.2	Common Features	20
3.3	Requirements	21
3.3.1	Functional Requirements	21
3.3.2	Non-Functional Requirements	23

3.4	Use Cases	24
3.4.1	Add Watch Folder	24
3.4.2	Watched File Modified	24
3.4.3	Remove Watch Folder	25
3.4.4	Recover File	25
3.4.5	Recover Old File Version	25
3.4.6	Peer Connects	26
3.4.7	Peer Sends Contract Update	26
3.4.8	Peer Sends Proof of Ownership	26
3.4.9	Distribute Shard	27
3.5	Architecture	28
3.5.1	Structure	28
3.6	Technologies	30
3.6.1	Languages	30
3.6.2	Frameworks, Tools and Libraries	32
4	System Components	35
4.1	File System Monitoring	35
4.1.1	File Tree Walking	35
4.1.2	Polling	36
4.1.3	Native Watching	36
4.1.4	Watching Enhancements	36
4.1.5	Deduplication	36
4.2	Local Storage System	37
4.2.1	Shard Storage	37
4.2.2	File History Storage	37
4.2.3	Backup Management	38
4.3	Peer Appraisal	39
4.4	Networking	39
5	User Interfaces	40
5.1	GUI	40
5.2	CLI	40
6	Results & Evaluation	41
6.1	Unit Testing	41
6.2	Empirical Testing	41
6.3	Summary	41
7	Conclusion	42
7.1	Limitations	42
7.1.1	Limitations of the Study	42
7.1.2	Limitations of the System	42
7.2	Further Work	42
7.2.1	Expansion 1	42
7.2.2	Expansion 2	42
7.2.3	Expansion 3	42
7.3	Summary	42

List of Figures

3.1	Model View Controller	28
3.2	Client Server Architecture	29
3.3	Membrane N-Tier Architecture	29

Chapter 1

Introduction

Distributed storage is a well studied and explored domain with clear advantages over storage on a single machine. In order to implement a peer-to-peer distributed agent based storage system we must first decompose the problem into several parts and explore the advances made in those fields as well as looking at advantages of existing technologies.

1.1 Project Goals

The goal of Membrane is to allow users to easily backup and recover the contents of folders on their computer without needing to pay for a subscription-based service. Looking at breadth of products available it quickly becomes evident that there is a huge desire for external backup, and many people now take popular cloud storage services such as Dropbox and Google Drive for granted. [Lynley, 2015]

To make Membrane competitive with existing solutions we must aim for two important features, ease of use and the ability to access your files from any system provided you can log into your account. On a centralised backup system this a trivial task, however, with a distributed system the concept of creating an account, logging in, finding peers and discovering which users hold your files is a much greater challenge.

In this project we are working towards a proof of concept of the idea of distributed backup, creating the fundamentals of the Membrane, focusing on the key aspects of distributed storage, but we will leave certain convenience features out for later development if the project proves successful. This is consistent with the Minimum Feature Set

methodology described by Blank [2010], a practise which is now used heavily amongst technology startups. The technique aims to reduce engineering hours and get the hands in the product of early visionary users early to encourage feedback.

The first step of the project will involve looking at existing products, seeing what design lessons from them we can apply to Membrane. We will then approach potential users and inquire what features they think are important in a system such as Membrane. A technology survey will be performed to assess what technologies we can use to ease with development, the solution will be built, and tested by users, whose feedback we will incorporate in the final prototype.

1.2 Need for Membrane

With the advent of cheap high-speed internet users are now able to use monolithic cloud services to backup data. These tend to be expensive for anything but small amounts of storage and many people have expressed security concerns over holding their data in a data centre owned by another company. [Batters, 2016] Especially if their data leaves the users country where data protection laws may be different.

Over time personal storage capacities for users have also increased. It is now common for computers to come with large amounts of hard drive space which is often not filled to capacity. The project proposed promises to swap this free hard drive space to backup other users' data in exchange for their surplus space to backup your data.

To simplify the process, the system will be

able to negotiate contracts of varying complexity for space allocation on another machine, in exchange for space on itself. Unlike most distributed databases and cloud storage solutions frequent down-time will be expected across devices and contracts will have to account for this.

There is an open gap in the market for a distributed storage solution which focuses entirely on backup and trading data, rather than charging a fee for storing data. The users of Membrane would be able to benefit from the advantages of commercial services, without any of the previous highlighted downsides.

1.3 Challenges

There are multiple challenges associated with creating both decentralised systems and backup solutions.

While doing this it is important to keep resource usage down as Membrane is a background process and should not be notice by the user. To achieve this we need to explore methods of reducing bandwidth, memory and processor usage.

As part of this we need to find a way to copy the minimum amount of data required during backups, an effective way of managing where file chunks are stored, a way to encrypt the files on the remote host and a way to periodically authenticate that the host still holds the file.

Part of the required research also includes exploring the more recent technology of intelligent agents, in which we are most interested in trust metrics. Intelligent agents will allow the software to make storage decisions based on the trust and reputation of other nodes in the network that it has interacted with in the past.

1.4 Modules

Membrane will be built up of several predetermined modules performing distinct roles. This have been broadly identified at this stage as a guide for our literature review and for technology decisions.

The core of any backup system is the file system watcher that determines whether a file has been modified, added or removed. This must be able to cheaply assess if a file or folder has changed and determine whether the altered file needs to to be refreshed in the storage system.

Once a modified file has been detected that change needs to be catalogued and the bits required to restore the file must be stored locally for further processing. This causes temporary duplication, however this must be performed so modification to the real files can continue while a peer willing to store the file is located. This system must also be able to recreate the file from the bits stored.

Next a system needs to be put in place to package bits from various files into a block of data that a peer can store. This system needs to be able to determine which peer certain bits should be stored with and how many peers the file should be stored with, based on trust, availability and other useful metrics that will be explored.

Finally a module responsible for networking and authentication must be created. This must be able to authenticate the user, find peers, establish connections to new peers and deal with the challenges to creating a secure communication link to the peers.

1.5 Conclusion

We have established the need for Membrane and briefly covered the advantages compared to existing solutions. We have also discussed the basic structure of Membrane and what challenges we expect to face during development. We will explore these further in our literature review.

Chapter 2

Literature Review

The literature review aims to preempt challenges by finding solutions to similar problems in literature and existing pieces of software.

We will first explore the history of file backup, proceeding into the challenge of locating, trusting, connecting and communicating with peers. Finally we will look at authentication issues as well as exploring how a peer can prove they own a file, without re-sending the entire file back to us.

2.1 History of Problem

A simple file backup can be imagined as simply copying a file to another location. In order to keep the duplicated file in sync they must be compared. The program diff solves this through finding the longest common sub-sequence of bytes between files. In order to improve performance hashing, pre-sorting into equivalence classes, merging by binary search, and dynamic storage allocation are used. [Hunt and MacIlroy, 1976]. This allows the user to view changes and copy the file over again if required.

2.1.1 Rsync

In a networked scenario, bandwidth from source to destination is at a premium. Rsync, introduced in 1996 presents a much better solution through copying changed file chunks (deltas). [Tridgell et al., 1996]. Rsync splits the file into shards and calculates a weak rolling checksum and strong MD4 checksum for each block that allows quick comparisons of shards along the file. When a discrepancy is found, we assume an extra byte or bytes have been added to

the file. The weak checksum can be efficiently recalculated for the next offset and once there is a match, it is confirmed with the strong checksum. The new added chunk can now be transmitted. This results in a lot less data being copied than there would be with a diff file. [Tridgell et al., 1996] This combination of weak and strong checksums has been used across multiple distributed systems including low-bandwidth file systems [Muthitacharoen et al., 2001] and high performance transactional object stores. [Stephen et al., 2000].

Multiround Rsync improves on the rsync algorithm by allowing for more communication to lower bandwidth. Blocks of smaller and smaller sized are used to find holes in the old file in each round and the file until the minimum block size is reached and a copy occurs. [Langford, 2001] This works better than standard rsync in situations where the source file has been changed in many places distributed around the file.

Rsync requires both old and new copies of a file to exist on the host system during an update. This issue has been addressed by creating an in-place rsync (ip-rsync) that uses techniques used in delta compression and Lempel-Ziv compression to move the areas of the file around. In ip-rsync file sender sends add and move commands to the destination in an order that guarantees no files will be overwritten. [Rasch and Burns, 2003]

2.1.2 Git

Git is an improvement on Rsync as it provides both version history and minimises data transfer. To keep storage simple, a copy of the whole file is stored and a reference is put into the version history. By stor-

ing old files locally operations are fast. This is also an important distinction from other version control systems and one of the reasons why Git was chosen as an example versioning system compared to other versioning systems like SVN. The systems can continue to operate without a centralised server. To reduce file duplication all files are referenced using their SHA-1 hash. This means you can be sure the contents of the file hasn't changed since you last read it. [Torvalds and Hamano, 2010]

Git also uses a 3 stage workflow. A working directory, where the current files are stored, a staging area and a .git directory. The staging area prepares your next commit and then it is finally committed. When the staging is complete the change is irreversibly stored. This is a good approach that will be adopted in the final software solution. It will allow incrementally finding changed files, and assessing the need for a new version number to be saved.

2.1.3 Bittorrent

The BitTorrent protocol is a mechanism for sharing files between swarms of hosts. As BitTorrent splits files into parts, users start sharing data even before they have received the full file. Each file has a SHA-1 identifier, similar to Git. [Qiu and Srikant, 2004]

If a user wishes to download a file from the swarm, the user downloads a metadata file from the web and locates users sharing the data using a Tracker Server, Distributed Hash Table (DHT) or Peer Exchange (PEX). [Cohen, 2008]

A Tracker server is a centralised store of all current connected users along with how much of the file they hold. This approach is vulnerable to exploitation by authorities as all of the data about a swarm is stored on a single server and as a result cannot be used for the proposed system.

A DHT contacts other known users for information instead of a centralised server. The Mainline DHT as outlined in BEP No.5 is based on the Kademlia protocol that allows for decentralised peer discovery for a particular piece of content.

PEX is a method for two clients to share

a subset of their peer lists. Coupled with DHT, PEX removes a vulnerability from the Bittorrent network by allowing fully distributed bootstrapping, tracking and peer discovery.

A DHT with a form of PEX a tried and tested way of successfully mapping and finding files on a network and will be used within the proposed project.

2.1.4 Resilio

Resilio Sync is an example of a distributed file storage system that utilises the BitTorrent protocol to automatically synchronise folders between a user's systems. It is not a cloud backup solution and not intended as a form of off-site storage. There is no distributed file system and as a result, no redundant data block algorithm adding complexity. [Farina et al., 2014]

As Resilio Sync uses DHT to transfer data, there is no central authority to manage authentication or log data access attempts. This makes it difficult to determine whether a file has been accessed by another user. [Farina et al., 2014] As a result in the project the assumption will be made that everyone in the network has access to all encrypted file chunks. To access and reassemble a file, a user will be required to request all of the file chunks individually and then locally reassemble them.

2.1.5 Storj

Storj is a peer-to-peer cloud storage network which aims to allow users to store files on a decentralised platform. The platform takes special care to provide protection against Sybil attacks and other forms of fraud. [Wilkinson et al., 2014]. To store files it stores encrypted hashed shards on the network. In order to provide proof of storage it uses Merkle Audits and pre-generated audits with hash-challenges to determine whether the client still holds the required data. By adding a seed to the hash-calculation the client can enforce the workers are still in possession of the data. It prevents the client cheating a farmer through using blockchain proof-of-existence to keep

both parties honest.

The most efficient form for proof of storage is through using a deterministic heartbeat. Using Feistel permutations data can be verified with $n + 2\sqrt{n}$ challenges. Erasure encoding is added to shards to detect any minor changes to the data. This is less I/O intensive than a full heartbeat, but still allows an attacker to complete heartbeats with only a data integrity of $1/n$, where n is the number of nodes holding the data.

In order to add extra protection to files, we can use erasure encoding to allow file recovery if one of our shard types is lost. This can be investigated in our software, however, as the shards are expected to change on a regular basis because of versioning, this may not be possible.

To prevent Sybil based attacks, Storj encrypts each shard segment with a different salt. This stops workers completing proof of storage on behalf of another node.

2.2 Peer Admission

The first step in designing the distributed file system is locating other peers within the swarm. This is accomplished through Peer Admission. Once the first peer is found data within the swarm can be located using a DHT which guarantees content can always be found.

2.2.1 Bootstrapping

There are two types of PTP networks which must be examined:

- *Asynchronous*
- *Synchronous*

Within Synchronous networks the number of nodes on the network is constant and all of the nodes are aware of each others existence. This does not allow storage networks to scale but it does allow data to be kept private. [Saxena et al., 2003] This is the simplest and first approach that will be taken in locating nodes within Membrane.

In most current P2P systems such as Gnutella [Klingberg and Manfredi, 2002],

Chord and Tapestry as well cryptocurrencies such as in Bitcoin and Litecoin a bootstrapping node is contacted, which provides information about what clients are currently online. Once a bootstrapping node allows the client to find the edge of the swarm, more information can be found using peer exchange.

Within a local network we can also use Universal Plug and Play to find other nodes within the local network. This prevents an external call to a bootstrapping node and as a result is less prone to attack.

Through looking at availability metrics within Bittorrent systems Neglia et al. [2007] determined that both trackers and DHT should be used in creating a highly available distributed storage system such as BitTorrent. DHT tends to be slower at finding new data, however it is much more reliable.

Within Membrane, I plan to use a combination of Asynchronous and Synchronous techniques. Users will try to bootstrap from their last known neighbour nodes on the network, this takes advantage of the static nature of the backup data. Only if this fails, and with the user's permission will they contact a centralised bootstrapping node. This should only happen during a first install and if the user has no referrals. Throughout the lifetime of the application the centralised bootstrapping node would ideally be replaced with a referral system.

A further extension of this, would be to allow hosts to provide a DNS name along with their IP. Users that have setup Dynamic DNS (DDNS) [Bound and Rekhter, 1997] would be able to locate each other without the help of a bootstrapping server.

2.2.2 Peer Exchange

When bootstrapping is complete new Peers can increase their knowledge of the network through Peer Exchange. This is used by Bittorrent to help share swarm information with other nodes. As soon as a client connects to the swarm, peer information is collected using DHT or PEX.

There are two common extension protocols called AZMP and LTEP, which send at

most one message a minute when a client leaves or exits the swarm. To reduce congestion at most 50 peers can be added or removed in one PEX message. [Vuze, 2010]

2.2.3 Shard Discovery

Bittorrent also uses the Mainline DHT to find other hosts in the network. This is a Kademlia DHT which now according to Jones [2015], now supports 25M users. It works through assigning each node and file a 160-bit string as an ID. We can work out which node is meant to store a file metadata and crawl in the direction of the node using a hill climb algorithm. Once the metadata is stored on the host, if a host wants to download a file, it can take the metadata on the known host to find the IP of hosts with the file.

Hosts in Membrane will store a metadata file with all of the information required for the specific account to run, including friends, encryption keys as well as local mappings for which host owns which shard and which IP belongs to each host. As a result a DHT will only be required if this initial metadata is lost and needs to be recovered. The proposed version of Membrane will make use of this metadata, but recovery can be added in further iterations of the project.

2.2.4 Dynamic IP Address

Dynamic IP addresses have proven to be problematic in distributed computing, as ISP typically charge more for users to have a static, unchanging IP. Bittorrent tackles this issue through using a DHT to dynamically find the IP address of the user that owns a file. This approach is robust, however, it is a complex solution to IP address resolution.

Another widely used approach is using Dynamic DNS (DynDNS) as described by Bound and Rekhter [1997] in RFC 2136. This allows a client to automatically update a nameserver with a new IP or other information, this allows clients to have a persistent addressing method for devices that change their location. This approach re-

quires initial configuration by the user, however, it provides a reliable way to connect with a user when their IP is lost. There are several tools such as MintDNS, cURL and Iandyn that could be used to ease the development of a built in DynDNS. When setting up a relationship with another host, both an A/AAAA Address and CNAME could be provided, where the CNAME is a backup if the A/AAAA address does not work.

To resolve IP Address resolution within Membrane, I would like to take advantage of small-world networks. [Porter, 2012], in which the mean shortest-path between two nodes increases slowly compared to the number of nodes in a network. Within a group of users in Membrane hosts are likely to share multiple first, second and third degree connections. By storing a list of IP addresses from all of the hosts. It is highly unlikely that all connections within three hops will have changed IP address. We take inspiration from ARP [Plummer, 1982], and send broadcasts with a limited hop count in the network to see if anyone is aware of the current address of a host. The downside of this approach is that it relies on a node within your social network to be online. Measures will need to be put in place to reduce broadcast spam.

A broadcast storms runaway broadcast events, common in networks that used broadcasting for communication, particularly when areas of the network overlap. [Tseng et al., 2002] These can be mitigated by reducing broadcast traffic, however, Membrane will rely on broadcasts to find hosts. The first step to limiting these broadcasts, is implementing a hop count on broadcasts. This is commonly seen in routing protocols such as IPv6 [Deering, 1998]. The Spanning Tree Protocol (STP) as seen in IEEE 802.1d [Group et al., n.d.; Sharma, Gopalan, Nanda and Chiueh, 2004] provides loop-free routing in LANs by pruning redundant links. Topology changes are dealt with by rebuilding the tree.

Within Membrane rebuilding a Spanning Tree would be an expensive operation. If broadcasts become a problem a 'block request' system can be used, similar to that

of ICMP redirects. [Postel et al., 1981] If a node receives a duplicate broadcast message it sends a request back one hop to not send broadcasts from that source toward it for some time. The time limit would allow for corrections if the network topology changes. This preventative approach and could be improved by using the full Spanning Tree implementation. As a further step it could be improved by using a DHT 'closest jump' approach if required.

2.3 Data Allocation on External Nodes

In order to store data on another node Membrane must first have permission to store files on another node. In order to make a choice we must be able to look at trust information about other nodes on the network, and negotiate and trade space once a suitable candidate has been found. These two areas have been explored in the context of Multiagent Systems (MAS) in the past. [Wooldridge, 2009]

2.3.1 Negotiation

Negotiation aims to reach a level of resource allocation that is acceptable for all involved parties. [Rahwan, 2005] It allows two or more parties that value each others service, to participate in a mutually beneficial exchange of services, however, as there are multiple beneficial outcomes it can be defined as "distributed search through a potential space of agreements" [Jennings et al., 2001] Within Membrane, this service is storage space, that is physically separated from the current user. We now take a look at negotiation and how we can build a negotiation framework that our agents can use to exchange storage.

There are three main areas that are important for negotiation. Negotiation protocols, negotiation objects and the node's reasoning models. [Beer et al., 1999] The sophistication of the negotiation is determined by each of these and can take different forms such as auctions, argumentation and protocols in the style of a contract net. The sim-

plest negotiation uses fixed size items, which Membrane shall be initially using for this reason. More complex negotiation allows for counter offers and stronger guarantees.

A simple negotiation protocol issues a call for proposals to a number of nodes and waits for their bids. To formalise this a Agent Communication Language (ACL) such as KQML (Knowledge Query and Manipulation Language) [Finin et al., 1992] or the more modern FIPA (Foundation for Intelligent Physical Agents) [Fipa, 2002]. [Rahwan, 2005] Beer et al. [1999] tells us that within KQML the agent sending the query is expected to decide how the receiver will handle it, which places limits on negotiation. On the other hand, FIPA is newer and as a result can be more error prone.

2.3.2 Negotiation Logic

The form of negotiation within Membrane also needs to be decided. We must first decide on a reasoning mechanism to use within the agent. The distinction between monotonic and non-monotonic logic is important in the study of AI and multiagent systems. In non-monotonic logic new axioms (or knowledge) can invalidate old theorems. [McDermott and Doyle, 1980; Antonelli, 2008] This is important in the real world as we need to be able to make assumptions on facts and retain flexibility in our understanding of the world. Within a MAS a non-monotonic logic can be more difficult to implement as theorems need to be constantly asserted, and as a result they often result to first-order (or monotonic) logic. Within Membrane we need to implement a monotonic logic to assert trust in our contracts and negotiations. In the context of a negotiated contract, throughout it's duration, we cannot effort to re-evaluate our trust of the agent.

2.3.3 Negotiation Tactics

In order to exchange storage space we must find the most suitable node in the network. There are multiple negotiation tactics between agents for collaboration and coming to an agreement. [Beer et al., 1999] We shall

explore the advantages and disadvantages of game-theoretic approaches [Rosenschein and Zlotkin, 1994; Kraus, 2001; Sandholm, 2002], heuristic-based approaches [Faratin, 2000; Fatima, Wooldridge and Jennings, 2002] and argumentation-based approaches [Kraus, Sycara and Evenchik, 1998; Jennings, Parsons, Noriega and Sierra, 1998] as well as exploring practical implementations negotiation. Ideally a negotiation mechanism is computationally cheap, produces good outcomes, distributed, fair to all participants, compatible with fixed strategies and is able to function without complete information. [Rahwan, 2005]

Using a *game theory* approach we assume all agents are self-interested and allows agents to analyse optimal behaviour. [Osborne and Rubinstein, 1994] We can apply this in agent reasoning by giving each combination of collaborations a utility. Doing this an optimal set of interactions can be calculated. This can even be used to help agents interact in a certain way. [Varian, 1995]. The main downsides include the assumption of unbounded computational resources, complete knowledge of the outcome space. [Rahwan, 2005] This makes a game theory approach unusable in Membrane as the network is not fully understood by each agent.

A *heuristic* approach produces 'good enough' negotiations. Instead of exploring the full extent of possibilities they focus on the subset most likely to lead to positive interactions. In the context of Membrane, this may be hosts that you have had successful interactions with before. The downsides of this approach is that it becomes difficult to predict the negotiation actions of other agents and as the full search space is not explored the result may not be optimal. [Jennings et al., 2001]

A *argumentation* approach is beneficial when a flexible negotiation is required and the agents have limited knowledge of the world. It's commonly used in the human world by advertisers to convince humans to try products. [Slade, 2002]. Instead of simply rejecting an offer a agent can say why or offer a counter-proposal, which can result

in more successful negotiations. Although this approach offers far better negotiation, it is much more complex to implement and is not required within initial implementations of Membrane.

2.3.4 Practical Negotiation

We now look at practical real-world negotiation. Real world approaches take into consideration the practical implications of reasoning systems and prove that concepts work. Within cloud computing an agent is often required to request a service. The use of these resources is based on service-level agreements (SLAs) which are designed to provide users with a service when requested. [Paletta and Herrero, 2009] One critical issue in SLAs determining the Quality of Service (QoS) constraints of the offered service.

Yan et al. [2007] explores creating a SLA negotiation system, splitting SLA negotiation into three parts. Defining *Negotiation Protocol* that allows participants to send offers to each other, *Negotiation Coordination* which ensures the final result of the negotiation fulfils the QoS requirements of the agent and a *Decision Making Model* which allows the parties to decide if they are satisfied with the deal.

Within Membrane, hosts are unable to know if they will be compatible with another agent. It might be the case that they both have 50% uptime, however they are never online at the same time. As a result a system will have to be created to allow agents to find if they are compatible with each other and storage decisions will have to be based on this.

Paletta and Herrero [2009] presents an negotiation mechanism which makes use of mean algorithm and protocol, using key awareness concepts first proposed by Herrero et al. [2007]. It uses an quantitative metric in the range $[0, 1]$ to decide how much the agent should be collaborating with another agent. The agents can then communicate using the messages:

1. REQUEST - Can you perform A?
2. CONFIRM - Yes I will do A.

3. DISCONFIRM - No I will not do A.

Collaboration decisions decided using an Artificial Neural Network (ANN). Three metrics are used, physical resource availability, if the node is available for collaboration and the number of previous collaborations of the same type. The ANN used was a Multi-Layer Perceptrons (MLPs) using one hidden layer and two units.

When evaluated the mechanism was able to deal with 93% of situations and negotiation with the first node was successful 68% of times. [Herrero et al., 2007]

With Membrane we take this model and make it more practical by introducing reputation to allow third party ratings to be shared.

2.3.5 Service Level Agreements

In order to create a negotiation system we need to explore what makes up an successful software SLA. Keller and Ludwig [2002] defines the WSLA framework which sets out to create a SLA based negotiation frame work. They describes 3 key areas that must be present in an SLA.

1. Parties Involved
2. Service Description
3. Obligation

Within Membrane the parties involved will always be the two nodes exchanging information. The service description will be a promise to store a block of data on another host. The obligation will include various parts such as proof of storage and the ability to update and retrieve the data a set number of times. These will be explored when the SLAs for membrane is designed.

Another key area Keller and Ludwig [2002] discusses is the 5 stages of an SLA lifecycle.

1. SLA Negotiation and Establishment
2. SLA Deployment
3. Measurement and Reporting
4. Corrective Management Actions (in case of violation)

5. SLA Termination (in case of violation)

These are the backbone of every real SLA and will be explored in more detail while designing the negotiation system for Membrane.

2.4 Trust and Reputation

Agents in a distributed system need to be able to protect themselves from 'bad' agents. Pinyol and Sabater-Mir [2013] describes three main approaches to control the acts of agents. The *Security Approach* which guarantees the authenticity and integrity of interactions. The *Institutional approach* which relies on a central authority to enforce good behaviour and finally the *Social approach* which allows agents themselves to punish other agent for 'bad' behaviour. This final approach is where trust and reputation is used.

Trust can be used to predict the behaviour of an agent. [Wooldridge, 2009] A classification presented by Balke and Eyermann [2009] defines 5 stages that exist in reputation and trust models. Co-operative behaviour is first stored and then rated using a utility function. Within Membrane it is easy to see the utility being rated as shared transaction time. This cooperative behaviour is then stored in an image of the other agent at a predetermined level of detail. This image can now be recalled to infer trust. Finally the agent can use this trust to learn and adapt it's strategy.

The ReGreT model [Sabater and Sierra, 2001] is "one of the most complete reputation and trust models" [Pinyol and Sabater-Mir, 2013] so we shall use it to see what a good trust system includes. It uses direct experience, third party information and social structures to calculate trust, reputation and credibility of another agent. An important note for this model is that trust is contextual. Agent a will trust b while certain conditions are met. In the context of Membrane, perhaps another agent will refuse to return our data, if we cannot prove that we still have their data, which would mean that they would be useless if the client experienced completed data loss. We must there-

fore consider simulating a situation in which complete data loss was experienced.

2.4.1 Trust and Reputation Attacks

Attempts to misrepresent reliability and manipulate reputation are common in traditional communities and have been exploited by con artists for centuries. In a distributed system agents must be able to protect themselves against common trust attacks. Jøsang and Golbeck [2009] describes 9 potential attacks on reputation systems.

- Playbooks - Gain high reputation and burn it quickly with low quality actions
- Unfair ratings - Give incorrect reputation or image
- Discrimination - Give high quality service to one set of users and low quality to another set
- Collusion - A coordinated reputation attack.
- Proliferation - Offer a service through multiple channels
- Reputation Lag Exploitation - Provide a large number of low-quality services quickly
- Reentry - Change identity to recover reputation
- Value Imbalance Exploitation - Gain reputation with easy actions
- Sybil Attack - Inflate reputation with fake accounts.

When implementing a trust system in Membrane these attacks should be considered and special attention should be paid to prevent a new user being exploited for their storage space when joining the swarm.

2.5 Communication

As Membrane is a distributed system communication is key for nodes on the network

to interact. It is traditional to form a protocol that agents can use to share information with each other. These protocols are often very rigid and do not allow for expansion. Formalising communication using ACLs is a more flexible approach that aims to let agent share a common understanding of the domain, and allows hosts to reason about communication themselves. Instead of using a strict protocol we shall instead take an agent-based approach to communication using ACLs.

2.5.1 Agent Communication Language

On balance, we shall be using FIPA for communication. This is an ACL based on Speech Act Theory [Labrou et al., 1999]. It splits communication into a communicative act (CA), an ACL message structure and a set of communication protocols such as XML or OWL (Web Ontology Language). In FIPA communication should be rational, in that when sending a message:

- The sender believes the proposition.
- The recipient does not already believe the proposition.
- The recipient will believe the proposition after the proposition.

In the case of Membrane this could be put into the context of asking another node to store data, it would only be reasonable to send another node data to be stored, if the two nodes had negotiated storage of that block between them previously.

To keep communication simple for nodes, we shall use two CAs

- REQUEST
- INFORM

where *REQUEST* expects a reply of some sort and *INFORM* does not. This will enable easier implementation and can be expanded if required.

2.5.2 Ontology

An ontology is a way of defining basic terms and relations comprising the vocabulary of a topic area. Sugumaran and Storey [2002] tells us the the three most commonly used relationships in an ontology are *is-a*, *synonym* and *related-to* (which is a generic association between entities). So why should we create an ontology for membrane? Noy et al. [2001] lists the benefits of ontology within distributed systems:

- Enable common understanding of the structure of information
- Enable reuse of domain knowledge
- Make domain assumptions explicit
- Separate domain and operational knowledge
- Analyse domain knowledge

Within Membrane we could benefit from an ontology during negotiation for storage space. Obligations could be predefined between agents and an agent could request a set of obligations, instead of explaining an obligation during negotiation. This allows agents to be more concise and expressive. Noy et al. [2001] gives us 7 steps for ontology creation.

1. Determine the domain and scope of the ontology
2. Consider reusing existing ontologies
3. Enumerate important terms
4. Define classes and their hierarchy
5. Define class properties (slots)
6. Define slot facets
7. Create Instance

It is important to ensure the ontology goes into the right amount of detail for the requirements. An overly detailed ontology can make reasoning difficult, and an ontology that is too simple will limit expressiveness and reasoning. [Wooldridge, 2009; Sugumaran and Storey, 2002].

When considering ontologies for reuse Noy et al. [2001] points us towards the DAML ontology library (<http://www.daml.org/ontologies/>). Using this and Google I was able to locate 2 ontologies, the datastore schema provided by purl.org [2001] and an OWL based ontology for SLA services provided by Pande Joshi and Finin [2012]. Both of these will be examined further during the ontology creation process.

2.6 Peer-to-Peer Connection

Network Address Translation (NAT), defined in RFC 2663 by Srisuresh and Holdrege [1999] is used extensively to solve the IP exhaustion problem, however, it also creates a lot of well documented problems for peer-to-peer communication as the IP address a hosts has set, may be a private and only useful for peers on the same network. Multiple Membrane hosts may be using the same IP address and the TCP listening port used by a node will change between connections. In order to transfer data over NAT a NAT-Traversal Technique must be employed. There are four major techniques described by Ford et al. [2005] that we shall explore.

Relaying is a method of communicating through a well known server, that has a static IP address and port. This has the advantage of being simple, however, it has the disadvantage of using the server's processing power, network bandwidth and increases latency between peers. It is a good fallback strategy but also requires both clients to initiate the connection, which makes listening for incoming connections difficult. The TURN protocol [Rosenberg et al., 2005] describes a fairly secure method for relaying.

Connection Reversal is the use of an external server to request a 'call back' to a host. The connecting host contacts an external server with a target host and leaves it's public IP and port. The relaying host can then choose to send a request to host behind NAT and the host behind NAT can choose to forward the request. It has sim-

ilar limitations to relaying, however, it is easy to imagine an implementation that uses connection reversal for bootstrapping a P2P connection.

UDP Hole Punching is a technique defined in RFC 3027 5.1 [Holdrege and Srisuresh, 2001] that allows two clients behind NAT to connect with the use of a rendezvous (RV) server. Hosts contact the RV and leave their UDP port and address. You should be weary of poor NAT implementations translating IP addresses within data when packets are coming in and out. [Ford et al., 2005] When clients are both behind the same NAT, packets between them do not need to go through NAT. If Hairpin NAT is correctly configured they will be bounced back within the internal network, however, it could result in overheads otherwise. If this problem is detected during the implementation of Membrane private IP address could also be sent to the RV server. It's important to consider that the ports behind a NAT are dynamically allocated, so connections must be authenticated between connections, to ensure the IP address and port still direct towards the same host. The initial outbound packet from a host to the RV server is key in 'punching a hole' through the NAT. The timeout of a hole punched port can be as low as 20 seconds, so the host must maintain the hole by sending single packets to the RV on a regular basis.

TCP Hole Punching is more complex than with UDP as clients need to establish sessions across the NAT. For hole punching to work the same port needs to be used for listening and establishing a connection. This can be done using *SO_REUSEPORT* option, which allows an application to claim multiple sockets from the same endpoint. Within the application a new port needs to be used for handling the connection.

There are two practical solutions that I shall explore for implementation within Membrane. TURN [Wing et al., 2010] provides a set of methods for NAT hole punching. UPnP [Boucadair et al., 2013] is a method through which a host can request a NAT gateway to open a port for the application. There have been security concerns

with UPnP which has resulted in UPnP being disabled on many routers. This will need to be explored during implementation of Membrane.

2.7 Distributed File System

Membrane shards files and stores copies on multiple nodes for redundancy. This is a technique that is commonly found in distributed file systems (DFS) such as Hadoop File System (HDFS) [Shvachko et al., 2010], Google File System (GFS) [Ghemawat et al., 2003] and Parallel Virtual File System (PVFS) [Ross et al., 2000]. This section aims to explore the reasoning and decisions taken with these file systems.

HDFS has proved to be highly successful in storing large quantities of data over thousands of nodes and is used by over 100 organisations world wide. It stores file system metadata on a central dedicated server called a NameNode [Shvachko et al., 2010]. This is a similar approach to GFS [McKusick and Quinlan, 2010]. Membrane takes this approach by storing file metadata on the client. Ceph takes this approach further, storing file meta on a distributed cluster of NameNodes, using hash function to spread metadata over the name nodes. [Weil et al., 2006] This allows for larger, more distributed system, this could be used within Membrane if meta-data proves too large to store on the client.

Files and directories are represented by inodes, these record attributes such as permissions and location. Within Membrane this will need to store information such as checksums, encryption keys and SLA for each file, to allow each file to be verified on a regular basis. When writing data to the cluster the NameNode chooses the targets for shards. The client in Membrane will be able to negotiate shard targets on the network using mechanisms described previously.

The NameNode stores the metadata as a combination of a checkpoint and journal (a write-ahead commit log). If there is a fault on the NameNode, such as a power outage, the journal can be replayed over the lat-

est checkpoint. In HDFS a new snapshot is created during a node restart, however, in Membrane the journal is kept to be able to restore the metadata during any point in the journal. By not removing any shard that is still in the journal we are able to rollback to any previous state in ahead of the oldest checkpoint.

In HDFS, during startup a DataNode connects to a NameNode, and performs a handshake with a unique storage ID that can identify it. In addition a DataNode will also send heartbeats and reports on a regular basis. Shvachko et al. [2010] This is a great approach as it makes the DataNode responsible for notifying the NameNode that it still holds the data. Not the other way around. A similar approach will be used in Membrane, however, the NameNode will also request some proof that the DataNode still holds the file, as in Storj.

In both GFS and HDFS data is streamed directly from the client to the DataNode in 64k chunks and a checksum is sent with the original file to assert the transfer was successful. Both systems also have a default of 3 replicas per shard, which Membrane shall adopt. TCP is used for communication as it provides reliability and a session for transfers. Chunks are given a 64bit chunk handle. Membrane will use a hash to compute this identifier. This should help nodes prevent attackers predicting a chunk handle.

2.8 Authentication

In order to maintain a relationship a host needs to be able to verify it's identity when it connects to a host a second time. When visiting a website there is often a need to map a virtual identity to a real identity of a service user [Hericourt and Le Pennec, 2001] a bank for example, this is typically done using a certificate authority. Membrane requires authentication in two situations:

- Is this the same node I was talking to before?
- Have I found the correct node when adding a friend?

Session authentication is a common occurrence while using the web and it typically happens with every https site visit. A session cookie is stored on both clients to allow for authorisation and authentication without the use of a password in further interaction. [Mayo et al., 2008]

SSL/TLS, defined in RFC 5246 [Dierks, 2008] is often used for this authentication. It uses asymmetric public-key cryptography for initial connection, which uses a separate key is used for encryption and decryption. A client generates a public SSL certificate that it sends for any users that want to communicate with it. They encrypt their communication with it, and the server can decrypt it. This is called a SSL Handshake.

1. Server sends the client it's asymmetric public keys
2. Client creates a symmetric session key and encrypts it using the public key.
3. Server decrypts the encrypted session keys
4. Client and Server can now communicate using the symmetric key.

The client can ensure it is talking to the same server by ensuring the public key is the same. RSA is the commonly used algorithm for this.

Pre-shared key encryption uses algorithms like Twofish, AES or Blowfish to create keys. These come in two flavours; stream ciphers and block ciphers. Stream ciphers encrypt binary digit by binary digit in a stream, and block ciphers encrypt a block of data at a time. We shall be using pre-shared key encryption to encrypt data blocks using a password saved on the host, that never under any circumstances leaves the Membrane host.

Studies have shown that Twofish (and Blowfish which it is derived from) has the best performance and has no known security flaws. AES showed poorest performance. [Thakur and Kumar, 2011; Rizvi, Hussain and Wadhwa, 2011; Mushtaque, Dhiman, Hussain and Maheshwari, 2014] An interview with the creator of Blowfish suggested

“I’m amazed it’s still being used. If people ask, I recommend Twofish instead.” Schneier [2007]. One thing to note is that encrypted data is expanded, testing has shown a 240KB file encrypted with Blowfish expands to 955KB which is almost a 4 times increase. [Mushtaque et al., 2014]

2.9 Proof of Ownership

A Membrane client needs a method to ensure the storage host still holds the data it says it has. To prevent Sybil attacks where agents collude to store a shard between each other, a shard will be salted before encryption and transfer. This has proven effective in Storj [Wilkinson et al., 2014].

In order to determine consistency of what the storage agent has stored a mix of different hash verification is used.

- Full Heartbeat - Expensive and complete
- Cyclic Check - This checks a sections of the file in sequence, wrapping to the start when the end is reached. Cheap but could be exposed to attacks if the storage agent wants to remove part of the file over time.
- Deterministic - Audits shards in a deterministic order known only to the client. This stops the storage agent being able to predict the next shard.

To stop the client pregenerating hashes the shard or shard chunk is seeded before hashing.

filecoin.io [2014] is another cryptocurrency operated file storage network. Transactions are stored in a ledger to assist with trust. At any point a client can issues a challenge to the server to clients who must the calculate the corresponding proof. These challenges are then confirmed by another node on the network who takes the file and recomputes the challenge itself.

Within Membrane we can build challenge issues into the SLA agreed upon by both users. If a client has no trust of another user, then they value their data less. The initial cost of building a relationship with a

user stops a storage node defecting by deleting their stored shard early. It is important to also run file retrieval spot checks, to simulate a situation in which a host loses all of their files.

2.10 Conclusion

In this literature review we discussed the key important areas of creating a distributed storage system.

We first explored the history of the problem by looking at solutions like Git. We saw how to exchange file and swarm information in distributed systems including BitTorrent and studied file storage systems such as Resilio and Storj that achieved distributed file storage.

Then we moved onto different challenges in creating the system, namely Peer Admission, Data Allocation, Communication and the challenges of connecting to other peers. We looked at the lessons learnt from distributed files systems and the challenges of authentication and proving ownership of a file.

Drawing on techniques explored in the literature review we will proceed to design Membrane, avoiding the pitfalls discovered by predecessors.

Chapter 3

Analysis

As Membrane is primarily an application built for home users the first step of analysis is querying potential users about what features they would like to see. This is proceeded by looking at what features existing solutions provide to create a Minimum Feature Set which, with the user requests taken into consideration, we will formally describe as software requirements. From this feature set we are able to create a development plan and explore helpful technologies that we will be able to reuse during product development. Finally we describe a target architecture for Membrane.

3.1 User Survey

In order to gather relevant user opinions we first decide at who the target user for Membrane is. Although the final product is for as wide a range of users as possible, initial versions may be a bit more difficult to use, as bugs are ironed out and features are added. We focus on those early visionary users that will be able to provide constructive and informed feedback during the early life of Membrane.

To narrow the search, we will target Linux users, who will be more accustomed to manually installing software and debugging potential issues. To gather feedback we used an online survey. These are typically answered by more technical users, have a short response time and require minimal financial resource allocation [Ilieva et al., 2002]. Eight participants were selected and asked to describe five key features they would like to see in the proposed distributed backup system.

We gathered all of the requested features

and placed the most popular into five categories that we will look to implement in the first version of Membrane:

- Data Security
- File Versioning
- Maintenance Free
- Fast Recovery
- Lightweight

The most important request by survey participants is *data security*. The concerns focused on how Membrane will ensure files stored on another user's computer are guaranteed to be inaccessible to the host user. Special care will need to be taken while showcasing Membrane to address this.

File versioning is seen as a key feature. Users want to be able to recover files that have been deleted or changed accidentally. This is expected many of the survey participants are accustomed to file versioning software such as Git. This poses a storage space challenge, as files that change frequently will take up valuable space.

Users also requested that after the first set-up Membrane would act transparently, with *no need for re-configuration* unless a change in functionality is required. During design we will aim to provide flexible configuration options, that will be able to gracefully recover from failure.

An unexpected feature requested is *fast recovery*. Users expressed concern about the amount of time that will be required to access files stored on the swarm. This is an interesting challenge as it is difficult to guarantee that users storing data will be online at any given point in time.

The final key request by potential users is that Membrane should not interfere with normal computer usage by the user, particularly processor time, memory and network utilisation. The request is particularly common among gamers who do not want to experience any slow-down to their games during backup.

We will focus on these five requests in the next steps of our analysis, however, more categorised requests can be found in the appendix. Given enough time we will explore more of these once the minimum viable product is created.

Insert
Ap-
pendix
entry

3.2 Common Features

Following the user survey we will look at popular features of existing solutions. In order to discover what features users are looking for we will explore features used to differentiate backup software in comparisons. The archlinux.org [2017] wiki has a comprehensive list of available backup options and uses a feature table to allow readers to quickly determine which solution is best suited to them.

We will try to explore all of these features, and incorporate feasible features into the Minimum Viable Product for Membrane.

Compressed Storage for files. The type and style of compression used. Within software this is not a difficult feature to add as there are many compression libraries. Implementing compression will reduce the amount of bandwidth Membrane uses during operation.

Storage Encryption is extremely important in Membrane as discovered during the user survey. We will need to look into potential encryption options in-depth while designing Membrane and ensure that there is no chance of a data leak.

Delta Transfer only transfers the modified part of the file if there is a change. This is important for users as it limits bandwidth and storage requirements. Within Membrane we will need to implement this as multiple copies of any stored data need to be transferred to peers.

Encrypted Transfer checks if the data is

transferred over a secure connection. This is less important for Membrane as files are encrypted for storage, however, an encrypted transfer would prevent a third party being able to see the interaction between two Membrane peers.

File system meta-data can be stored with the data so it is restored along with the file. This could be implemented in Membrane, however, as this wasn't requested by users and does not help demonstrate the advantages of peer-to-peer storage it will not be added to the Minimum Viable Product.

Easy access to files is an important feature in backup systems, however, as the MVP of Membrane seeks to show peer-to-peer backup instead of a perfect backup system, we will add this feature to future work.

Resumable Backup is very important in Membrane. If a connection to a peer is lost Membrane needs to be able to resume the backup with other peers.

Another vital feature in Membrane is the ability to *handles renames*. If a file is moved or renamed, the software needs to be able to detect this to reduce data duplication.

Users searching for backup solutions were also very interested in the software's *interface*. This needs to work well and be suited to the user. Within Membrane we will be looking to implement a fully featured *command line interface* (CLI) for configuration, as well as a *graphical user interface* (GUI) for backup monitoring. This is common in most of the featured backup solutions, so to remain competitive these need to be implemented.

The comparisons also covered platforms supported by the backup solutions. An interesting observation is that although multi-platform support has clear benefits, many of the backup solutions covered only supported one or two of the available platforms. This will be taken into consideration while creating the formal specification for Membrane.

Finally users were also interested in the *software licence*, with an emphasis on how open-source the code for the backup solution is.

We can see there is an overlap between the features requested by users during the initial

survey and what users in need of a backup solutions use to determine which software is best for them.

3.3 Requirements

In the requirements we will formalise the aforementioned research into goals to be completed during development. These requirements will be able help with planning development and designing an architecture for Membrane.

We will begin with functional requirements which describe technical features within the desired system, allow us to evaluate it's behaviour and will aid in its design by ensuring key features of interest are present followed by Non-Functional Requirements which help support functional requirementscite.

There is some debate about the usefulness of software requirements, so the requirements below will be kept brief, however, they will be able to help guide the software development goals and produce targets for development stages.

3.3.1 Functional Requirements

Functional requirements describe components found within software and their function; inputs, the behaviour, and outputs. These requirements consist of name and number, a brief summary, and a rationale. It's important to ensure the requirements are clear to prevent misinterpretation.

1. File System Monitoring

The system must be able to be given the name of one or more folders, and monitor all the files for changes in those folders. Another module must be able to subscribe to a stream of these changes for further processing of the modified files. The system should make use of preexisting file system features for monitoring if available.

This needs to be done to allow the user to select a folder for backup

2. Sharding Module

The module must be able to receive the path of a file and determine whether the file has

changed since the last read. If a change, addition or deletion is confirmed this update must be sent to any subscribers for further processing. Individual file chunks for the modified file (if any exist) must be sent to any subscribers to be persisted. Updates for preexisting or suppressed files must be suppressed. On launch the system must be able to receive a list of files already stored by the system for suppression.

We do this because the second person said so.

fixme

3. File History

Must be able to keep a log of file system modifications. It must be able to remove entries from the log to match storage requirements by analysing the storage requirements for storing the entire history and selectively removing elements of the history to meet requirements. The history must be immediately persisted, so an application crash does not affect the operation.

This needs to be done to match the request in the user survey to store file history. In addition this will allow for delta transfers as files can be compared between versions.

3. Shard Storage

Must be able to persist and fetch requested shards to a given folder. It must perform consistency checks on retrieval and support hard limits on total storage size. Finally it must be able to return a complete listing of all shards inside the storage unit.

This is a key feature of the backup system as it persists shards, which can then be assembled into files.

4. NAT Traversal

Must be able to establish a secure TCP connection when two peers are behind a NAT Gateway. The suggested mechanism is to use UPnP Port Forwarding offered and enabled on most routers. In future iterations consider using TCP Hole Punching. This must be able to temporarily forward and maintain a port on the target router. If a port is taken another open port should be used. The system must also be able to return the external IP address.

This is required to allow two users who wish to use each others external storage, who are both behind a NAT Gateway wish to connect. It is a common feature of distributed storage such as BitTorrent

5. Authentication

Must be able to generate, persist and reload a public private key pair and X509 self-signed certificate used for establishing secure SSL connections. These details must be used throughout the application lifetime and will act as a unique identifier for the user.

This is required to allow users to create accounts on the distributed network.

6. Peer Connection

Must be able to establish a secure connection to a given an IP and port. When this connection is established the user must be authenticated using their provided SSL certificate public key. This must be converted into a unique user identifier. All messages from the peer must contain this unique identifier or they should be dropped.

This module forms the core of the connectivity in Membrane allowing users to dial each other.

7. Peer Exchange (PEX)

Must be able to send and receive connection information about yourself and other peers. Peer information about yourself needs to be signed with a time stamp for future verification. Incoming verification about specific peers needs to be verified. Needs to support peer discovery, finding peers that have never been contacted before.

This is done to allow peers to find each other if they have changed external IP address or port and to allow new peer discovery.

8. Connection Management

Given a target user count, maximum connection count and whether new users should be found, should dial known contracted peers on a regular basis, request PEX information for connected peers and send up-to-date PEX information to connected peers.

Should connect to trackers if peer target has not been reached in time.

This module is responsible for providing live connections to peers that can be contracted and have data blocks sent to them.

9. Shard Distributing

Given a list of local shards and file history, needs to be able to package appropriate shards into blocks and send them to peers, depending on which peer is most suitable for storing the shards. Needs to send contract updates on a regular basis and respond to and issue proof of ownership requests for blocks to and from peers. Needs to be able to remove peers that are not worthwhile.

This is a core module connecting the local backup to the network module and required for distributed backup.

9.1 Distribution Storage

Must be able to store information about generated blocks, which blocks are given to which peers and what blocks have been provided by what peer. This includes any proof-of-ownership verification information that may be required in the future, shard contents and block unique identifier. The module must be able to persist this information and load it from disk.

This is a core module required for (9.0) to verify module information and compute which shards should be given to which connected peer.

9.2 Block Encryption

Any blocks of data sent to peers must be encrypted using TwoFish encryption using the private modulo used for network SSL connection.

This is required for data security requested by potential users during the Analysis survey.

9.3 Peer Appraisal

Peer interactions must be logged at a resolution of one hour for rating peers. Given the number of expected shards, all of the shards that have been proven to be held by the peer and any lost shards this must produce a single number between 0.0 and 1.0 rating the peer to allow for comparison.

This is required for selecting which peer is most appropriate for upload.

10. Proof of Ownership

A system that can determine whether a peer is holding a block is required. The suggested method is a mixture of salted hashing and asking for the entire block to be returned.

This needs to be done to determine whether the given peer is holding a block without having to request all of the data every time.

11. Tracker

Membrane needs to be able to run in tracker mode, in which the backup backup is disabled but peer exchange is active for other hosts to share PEX information through the tracker.

This is required for new peers, or peers that have expired PEX information.

12. Command Line Interface

A fully featured CLI needs to be present. It must allow users to monitor network activity and their backups, as well as adding other folders to backup and recovering files.

This feature is required to allow users to interact with Membrane.

13. Graphical User Interface

A GUI needs to be present to give users the ability to monitor membrane activity.

This feature was common in existing solutions and is required for Membrane to be competitive.

3.3.2 Non-Functional Requirements

Non-functional requirements, also aptly named quality requirements help support the functional requirements described above. These can be grouped into two broad categories:

- Execution Qualities - Those observable at run time including security and usability.
- Evolution Qualities - Those found in the structure of the system including extensibility, scalability, maintainability and testing.

13. Maintenance Free

The software must be able to run without external monitoring or reconfiguration. The user cannot be asked to solve any non-configuration related issues. For example, if a peer loses all of our blocks, the software should be able to recover without any user intervention.

This was requested by users during the analysis survey.

14. Fast Recovery

The software must be able to recover files quickly after data loss.

This was requested by users during the analysis survey.

15. Resource Usage

Membrane's resource usage must be as low as possible to minimise impact on other software running on the system.

This was requested by users during the analysis survey.

16. Open Source

The implementation must be fully open source. No closed source libraries can be used during development.

This feature will help with adoption as Membrane will be marked as open source in backup utility comparison charts.

17. Extensibility/Transparency

The project should be built to allow for easy contribution from other open source programmers.

This will help with community engagement in the product. More technical users will be able to add their own features, which they will be able to push back into the main code base.

18. Testability

The project should be built with extensive tests asserting the behaviour. Tests should be automated to assist with requirement 16. It will allow easy verification of the behaviour of the modification.

This will help to gain the trust of potential users.

3.4 Use Cases

A use case defines a set of goal-oriented interactions between a user and the system. Actors are any external parties that interact with the system [Malan and Bredemeyer, 2001]. Cockburn [1997] tells us use cases can have multiple forms and purposes.

We opt to use the scenarios below to help strengthen the requirements, with the purpose of helping developers check if the software is able to perform the use cases.

As with the requirements, we keep the use-case brief, as we expect they may change during development as better approaches to problems are found.

3.4.1 Add Watch Folder

The expected behaviour of Membrane during a normal backup operation when a new folder is added. The computer owner is primary actor.

Goal

Store redundant copies of files on the system

Steps

1. User adds folder for backup with files.
2. Folder is persisted to the configuration file.
3. New files in folder are detected.
4. Files are sharded into equal chunks and sent to shard storage.
5. The change is logged and saved in the file's history.

Extensions

1. Submitted folder does not exist:
 - (a) Accept entry and wait until it does.
2. Cannot persist watch folder due to IO error
 - (a) Inform the user
 - (b) Do not continue
3. Shard storage full:
 - (a) Do not log the change in file history.
 - (b) Shorten the history until soft limit is reached.
 - (c) Retry submission later.
4. Cannot log change in file history:

- (a) Leave new shards.
- (b) Retry submission later.

Variations

1. Folder is file:
 - (a) continue and wait until file becomes a folder
2. File present in multiple watch folders:
 - (a) De-duplicate file by sharding
 - (b) log changes from both folders if path different

3.4.2 Watched File Modified

The expected behaviour of Membrane when a watched file is modified. The computer users are the primary actor.

Goal

Store a copy of the modified file in shard storage

Steps

1. User modifies files in watch folder.
2. During next scan file change is observed.
3. Assert if file modification time has changed.
4. File is sharded.
5. File shards are hashed to check for similarities.
6. Assert at least one shard has changed.
7. Chunks and sent to shard storage.
8. The change is logged and saved in the file's history.

Extensions

1. Modification time has not changed:
 - (a) Ignore change
2. None of the file shards have changed:
 - (a) Ignore change
3. Shard storage full:
 - (a) Do not log the change in file history.
 - (b) Shorten the history until soft limit is reached.
 - (c) Retry submission later.
4. Cannot log change in file history:
 - (a) Leave new shards.
 - (b) Retry submission later.
5. File cannot be read:
 - (a) Retry later.

Variations

1. File renamed:
 - (a) De-duplicate file by sharding
 - (b) Log as different files.

3.4.3 Remove Watch Folder

The expected behaviour of Membrane during a normal backup operation when watch folder is removed. The computer owner is primary actor.

Goal

Stop watching the provided folder

Steps

1. User removes folder for backup with files.
2. Removal is persisted to the configuration file.
3. Any queued modified files are removed.
4. Stop scanning for changes in folder.

Extensions

1. Submitted folder is not watched:
 - (a) Report user error to user.
2. Cannot persist watch folder due to IO error
 - (a) Inform the user
 - (b) Do not continue
3. Files from folder persisted
 - (a) Do not remove file history.

Variations

1. Folder is file:
 - (a) treat as folder

3.4.4 Recover File

The expected behaviour of Membrane when user tries to recover a file. The computer owner is primary actor.

Goal

Recover a file into a new location

Steps

1. User asks to recover a file to a given location.
2. File log checks for most recent version of file.
3. File shards read and reassembled.
4. File written to target destination.

Extensions

1. File never saved to Membrane:
 - (a) Report user error to user.
2. File deletion most recently recorded in Membrane:
 - (a) Report user error to user.
3. Shards cannot be retrieved or not present
 - (a) Inform the user
4. Target destination already has file in it
 - (a) Do not overwrite
 - (b) Report error to user

Variations

1. Given file is a folder:
 - (a) Report error. Only accept files.

3.4.5 Recover Old File Version

The expected behaviour of Membrane when user tries to recover a previous version of a file. The computer owner is primary actor.

Goal

Recover a previous version of a file into a new location

Steps

1. User requests for all available versions of file.
2. Show users all known
3. User asks to recover the version of the file at a specific point in time to a given location.
4. File log checks for most recent version of file.
5. File shards read and reassembled.
6. File written to target destination.

Extensions

1. File never saved to Membrane:
 - (a) Inform user no versions exist.
2. File deleted in period of time user requested:
 - (a) Inform the user.
3. Shards cannot be retrieved or not present
 - (a) Inform the user
4. Target destination already has file in it
 - (a) Do not overwrite
 - (b) Report error to user

Variations

1. Given file is a folder:
 - (a) Report error. Only accept files.

3.4.6 Peer Connects

The expected behaviour of Membrane when a peer connects and wants to exchange storage. The peer should be given one free block to allow them to start storing data.

Goal

Establish a new storage contract with peer if required.

Steps

1. Peer connects.
2. Peer identity is verified.
3. If contracts are required.
4. Peer certificate is persisted
5. Peer is given to the contract manager.
6. Contract Manager gives the peer one new shard of storage.
7. Contract Manager sends a Contract Update to the Peer.

Extensions

1. Peer already contracted:
 - (a) Proceed whether or not contract is required.
2. Peer identity verification fails:
 - (a) Drop connection.
3. No contracts required
 - (a) Keep connected
 - (b) Do not contract
 - (c) Do not persist certificate
4. Disconnects before contract update can be sent
 - (a) Keep contracted
 - (b) If peer does not provide you with one contract inequality before shutdown remove them.

Variations

1. Peer is a tracker:
 - (a) Do not contract them.

3.4.7 Peer Sends Contract Update

The expected behaviour of Membrane when a peer sends a contract update. The peer should be provided with proof-of-ownership

instructions for any owned blocks they have not authenticated this hour.

Goal

Provide peer with proof-of-ownership instructions.

Steps

1. Verified Peer sends Contract Update
2. Remove any blocks we thought the peer had but they have lost. Note these down in the appraisal.
3. Request any unexpected blocks (We may have lost information about them on our side)
4. Request deletion of any expired blocks which we cannot assert ownership of.
5. Request salted hash for any blocks they have not confirmed yet this hour. Include salt.
6. Record that peer has been in contact this hour.
7. Send requests back to peer.

Extensions

1. Peer not contracted:
 - (a) All blocks will be unexpected.
 - (b) Request all of the blocks. They might have been lost during a data loss incident.
 - (c) Blocks will be deleted on the next request.

3.4.8 Peer Sends Proof of Ownership

The expected behaviour of Membrane when a peer sends a proof of ownership response. Any successful responses should be noted down for the peer.

Goal

Check ownership details for the peer and record them in appraisal if successful.

Steps

1. Verified Peer sends Proof of Ownership Response
2. Process any full blocks returned, adding missing shards.
3. Check salted hash was correct.
4. Add correctly confirmed or returned blocks to peer appraisal.

Extensions

1. Peer not contracted:
 - (a) Ignore any salted hashes

Variations

1. Full block returned as response:
 - (a) Decrypt the block
 - (b) Decompress any compress shards in the block
 - (c) Place shards in local shard store
 - (d) Place any file history in the block into the file history store.

3.4.9 Distribute Shard

The expected behaviour when a shard is stored in local backup. The shard should be packaged with other similar shards into a block, and sent to a peer offering storage space.

Goal

Deploy shard to peers.

Steps

1. Shard is placed into local shard storage.
2. Distributor performs periodic local shard storage scan and finds shard.
3. Connected peers are ranked from best to worst.
4. Any peers with a rating $<1.5\sigma$ are ignored and the contracts are cancelled.
5. Shards not yet sent to each peer are collected.
6. Shards are compressed and placed into a block until the block reaches a predetermined size.
7. The history of any files associated with the shards is placed into the block.
8. The block is encrypted.
9. The block is sent to the peer.
10. The peer sends a contract update on receipt.

Extensions

1. No contracted peers connected:
 - (a) Wait for new contract or contracted peer to connect.
 - (b) Retry.
2. Shard is missing from local shard storage during read:

- (a) Log the error and skip the shard during this pass.
 - (b) Retry.
3. No shards fit into the block:
 - (a) Log the error and move onto the next peer.
 - (b) Continue as normal.
4. Shard compression is unsuccessful:
 - (a) Place the shard into the block with the compression flag disabled.
 - (b) Continue as normal.

Variations

1. Block does not arrive at peer:
 - (a) Mark loss of block on peer appraisal.
 - (b) Mark incomplete hourly report on peer appraisal.
 - (c) Remove block from local contract.

3.5 Architecture

Before deciding on the technology stack used in Membrane we must consider the application architecture.

Although Membrane is a peer-to-peer network application, which is distinctive from a client server application, because peers act as a *Servent*¹, we can still make use of recent developments in client server architectures, in particular the arrival of widespread adoption of Representational State Transfer (REST)ful Web services [Rodriguez, 2008], and the tooling that the popularity has generated.

3.5.1 Structure

The first major decision in the application structure is whether the GUI and CLI will be built directly into the application or if Membrane will expose an API for interaction. To prevent coupling and enable software reuse within the application [Gamma, 1995] we will use a 3.1 or the more general n-tier architecture components of the application.

There are clear advantages to both a built in user interface and the server client architecture seen in figure 3.2 which we shall explore before selecting technologies to use.

Native Interface vs. Client Interface

A native application interface is popular among server software and frequently used for desktop applications such as Microsoft Word and Microsoft PowerPoint, however, there has recently been a shift back to using a client, running back-end services in the cloud, exposing an API which can then be interacted with client side applications.

A separate client interface has multiple advantages and with modern networking speeds a user is typically unable to distinguish between running the interface on the server or a thin remote client [Schmidt et al., 1999].

The user is given the flexibility of using their preferred client to interface with the

application, switching client without needing to worry about losing data, and developers are given the option to design a new interface without affecting the core application.

The user is also given the ability to interact with multiple instances of the application running remotely with no issues, which is particularly useful when running the application on a headless server.

Development of both the client and interface can also happen in parallel leading to better time to market in large development teams.

However, we must not forget the benefits of using a native application. By making the UI interface with the application over the network, a whole host of complexity and security issues arise. The development team needs to ensure that any user interacting with the API is authenticated and secure, as well as ensuring that there are no network issues between the interface and application.

The software itself also becomes more complex with a separate client as functionality needs to be built for handling network in both the UI and core application.

On balance, it makes sense for membrane to operate use a client interface. As Membrane will need to be run constantly, it needs to be able to operate as a soft-

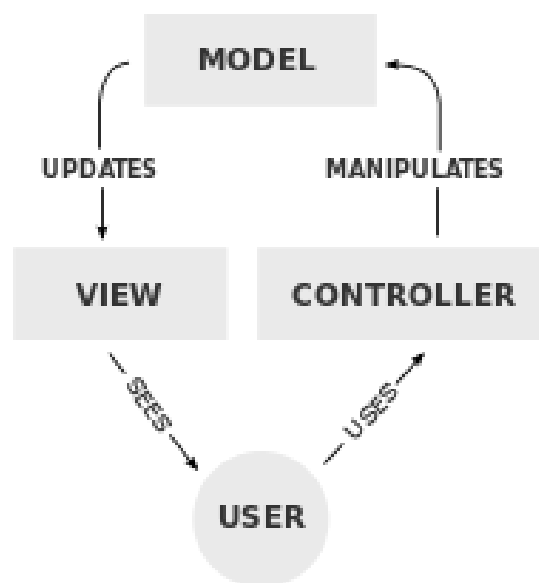


Figure 3.1: Model View Controller

¹A term coined by Schollmeier [2001] to describe a peer acting as both a Client and Server

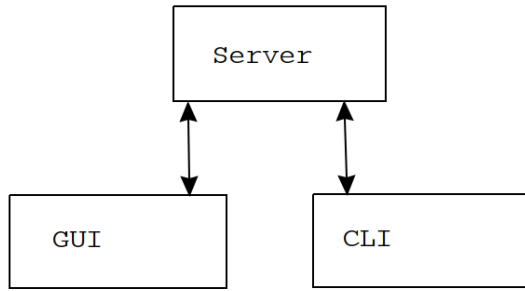


Figure 3.2: Client Server Architecture

ware daemon. One of the requirements is to keep Membrane lightweight and maintenance free, running any user interface constantly would distract users.

Developers will also be able to build addition interfaces to Membrane if required, fulfilling the extensibility requirement, and as both a GUI and CLI are required, this provides a nice way to develop both and gives users the desired flexibility between both.

Model View Controller (MVC)

The Model View Controller (MVC) pattern, first described by Krasner and Pope [1988], decouples the interface from the main logic of the program by establishing a subscribe notify protocol between them. The view must always reflect the state of the model and the model must always update the view when a change occurs.

This approach allows for code-reuse and lets the developer to create new models without rewriting the software, and provides high cohesion, meaning groups of related actions can be grouped together.

This has the disadvantage of making code harder to navigate due to added structural complexity, increasing scattering meaning multiple representations of objects need to be modified at the same time if required and increases the learning curve for developers, as they need to learn how to structure the application in a new way.

Although the MVC model provides a structurally sound way of building the application we need to generalise in into the N-Tier architecture for larger applications.

N-Tier Architecture

An N-Tier architecture is simply an extension of the MVC model, it takes the idea of layers in an application further, allowing sections of the software to be developed and tested without interfering with each other.

One of the key advantages to this approach is extra layers can be added and removed, even during run time. In Membrane we will look to take advantage of this when running in Tracker or Local Backup mode. The tracker can disable all of the local backup functionality, while the tracker only needs to run the network layer.

We can see the Membrane N-Tier architecture in fig 3.3. Only modules next to each other are able to communicate directly and modules can be disabled as required.

The local backup performs all the basic tasks of file monitoring and local shard storage. The distributor monitors the local backup for shards to send to peers and packages them up to the Peer Networking

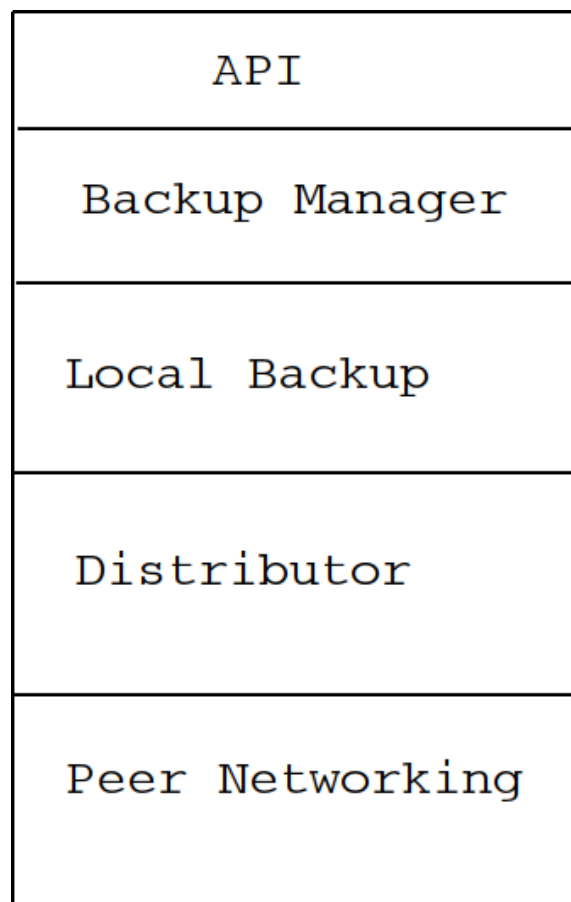


Figure 3.3: Membrane N-Tier Architecture

layer for transmission. The Peer Networking layer is able to locate new peers and provides them to the Distributor if connected.

3.6 Technologies

The technologies used in a project can define the success of a software project. Incorrectly chosen tools can slow development and in some cases even cause projects to fail. While selecting technologies we will need to be weary of excitement surrounding modern technologies and look for unbiased advantages and disadvantages of each.

We will begin by selecting the language to use, followed by key libraries used within the software itself. Some of these libraries were chosen during development when the need for them arose.

3.6.1 Languages

A wide variety of languages are available for software development. We need to consider the best language for development of the Membrane Daemon, the GUI and the CLI. One of the most important features of a language is support from the developer community. An active developer community means any issues we encounter during development will most likely, already be solved by someone else.

We used the 2017 Stack Overflow Developer Survey [stackoverflow.com, 2017] to find language popularity statistics. This is an annual survey held for users of one of the most popular developer forums, covering everything from languages, to frameworks, to demographics.

Daemon

There are a few requirements when selecting a programming language for the daemon. The language must scale efficiently, restricting our choice to object-oriented languages. It must also be compiled, to reduce the chance of runtime exceptions the user would need to debug. Platform Independence is also key so Membrane can be easily ported to other platforms after the proof of concept is developed.

JavaScript

The most popular language by far is JavaScript, a high-level, dynamic, untyped language that is interpreted during runtime. [Flanagan, 2011] It is one of the three core technologies used in the world wide web, which has done a lot to push it's popularity. It has been standardised in the ECMAScript language specification, which has recently reached it's 7th edition. [Stefanov, 2010].

High-level languages uses strong abstraction to hide details of the underlying computer, such as memory management. This makes them easier to use and usually makes them much less bug prone.

Dynamic languages are a class of high level programming languages that execute behaviours usually performed at compile time in static languages during runtime. [Tratt, 2009] This typically means the language is dynamically typed (although not always). In the case of JavaScript, this is true.

An interpreted language is one for which most of the implementations execute instructions directly on the machine without compilation. The interpreter translates each statement into a sequence of subroutines that have already been compiled into machine code. In the case of JavaScript most of the runtime environments include a just-in-time (JIT) compiler, which compiles the program during runtime, allowing for dynamically recompiling frequently used code sections, optimising those sections. [Aycock, 2003]

Java

We move onto Java, the second most popular (applicable) language. This is a general-purpose language that is object-oriented, class based and concurrent, [Gosling et al., 2014] designed to run anywhere, meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. [Gosling and McGilton, 1995]

Object Orientation is a programming paradigm based on "objects" which contain data and methods that can act on that

data. The class based aspect of java means that these objects are in fact instances of classes that determine their type. [Kindler and Krivy, 2011] Object are able to offer encapsulation, an desirable feature that means only objects can manipulate data inside them. This facilitates code refactoring and encourages decoupling, which makes it very desirable for large software projects.

C#

The final language we consider is C#, which is another general purpose programming language that offers strong typing, is imperative, declarative, functional and object-oriented. This makes it very similar to Java [Kreft and Langer, 2003], and in fact has been criticised as a “imitation” of Java, although this has been disputed. C# is a younger language than Java, so it was playing catch-up to Java’s feature set for a long time, however, it has at this point surpassed Java.

On balance Java’s popularity, larger community of users and open source nature means it is more suited to Membrane, although C# has some clear advantages, it is not as well supported on Linux, requiring Mono to be run. It is also important to take into consideration the ability of the programmers coding Membrane, who have coded predominately in Java in the past.

CLI

We also consider the language to use in the GUI and CLI. For the CLI a fast, lightweight language is required. The three languages under consideration are Python, C and Go, selected because of their popularity, and the programmers familiarity with them.

Python

Python is a general purpose, dynamic, high level programming language. It is known for its design philosophy, which emphasises readability through forced indentation and a syntax tailored to representing algorithms in as few lines as possible. This is a strong contender for the CLI, however, the most popular implementation of Python, *CPython*, is a runtime interpreter of language making it

slightly more difficult to ensure code correctness.

C

C is a general-purpose, imperative, statically-typed language developed in 1973 at Bell Labs. [Banahan et al., 1988] It has become one of the most widely used programming languages of all time, providing low-level access to the hardware, uses language constructs that map efficiently to machine instructions, and requires minimal run-time support.

One of the key disadvantages of C is that its power makes it easy to make mistakes. This low level access is not required for the CLI of Membrane and it could potentially lead to subtle errors.

Go

The final language, Go is a compiled, statically typed language with garbage collection. It is designed to be scalable, not require IDEs, support networking and concurrency, and be productive and readable. [golang.org, 2017]

This is a great compromise between the ease of programming in Python and the strictness found in C without sacrificing any of the benefits. Critics of Go say the lack of generics limit static

We also refer back to the Stack Overflow developer survey and see that Go is one of the most wanted and loved languages by developers, leading us to select Go for the CLI of the Membrane.

GUI

Finally a language for GUI development needs to be selected. This depends strongly on the choice of Electron for the GUI framework discussed in detail below. Although, as we have selected a web technology we need to use JavaScript, there are two newer languages which can transcompile to JavaScript that enhance JavaScript’s feature set. We will compare these.

CoffeeScript

CoffeeScript aims to add syntactic sugar inspired by Python, Ruby and Haskell to

improve JavaScript's brevity and readability. [MacCaw, 2012] The compiled output is readable and tends to run as fast or faster as equivalent JavaScript. [coffeescript.org, 2017]

There have been concerns using CoffeeScript adds bloat to development, every change must be compiled before the effect can be seen, however, this can be solved by automating the process during development, any time a page is reloaded, the server automatically transcompiles any modified CoffeeScript files to JavaScript. [Wheeler, 2012]

Another concern is that using a transcompiled language makes debugging difficult as the debugger steps through a file you have not created, however, because the compiled output is intended to be readable, this has not caused issues for developers. [Wheeler, 2012]

TypeScript

TypeScript is a super set of JavaScript and developed and maintained by Microsoft as open source software. It is designed for development of larger applications and aims to address the challenges of dealing with JavaScript code. It supports static typing and adds features such as classes, modules and arrow function syntax, leading to better, more readable code. [typescript-lang.org, 2017]

There are two major concerns with using TypeScript. To get the most out of TypeScript, developers need to add type annotations everywhere in their code, which makes coding a bit more cumbersome in comparison to JavaScript. In addition, although TypeScript's type system is more flexible than mainstream statically typed languages, it still makes some common JavaScript software patterns difficult for developers to use. [DataArt, 2014]

This is in addition to all of the considerations of CoffeeScript. On balance, we chose to use TypeScript for the Membrane GUI. The extra type safety provided outweighs the downsides of making it slightly more tricky to write the code. In addition if any quick coding is required, it is fully compatible with JavaScript so any sections of

code unsuitable for TypeScript can be written using JavaScript seamlessly.

Conclusion

We have decided on three programming languages to use to create Membrane. Java, Go and TypeScript. These are three well-supported, popular languages best suited for both the programming task required and for the developers coding the Membrane proof of concept, to produce a stable and complete product.

Other languages including Bash, HTML, CSS, Python, Markdown and Groovy will be used during development, however, these are simply a requirement of the languages, frameworks and tooling chosen.

3.6.2 Frameworks, Tools and Libraries

To aid with development multiple frameworks will be used some of these will be chosen at the start of development, whereas other will be selected during development as a need for them arises.

Java Build System

There are three major build automation systems available for Java that help with application development, namely Ant, Maven and Gradle. These are responsible for compiling source code, running automated testing, detecting code coverage and packaging.

Apache Ant was first released in 2000 and was the first modern build tool available for Java. It is able to accept plugins and is designed to have a low learning curve, enabling anyone to use it without special preparation.

Ant uses XML based configurations, to create build scripts, which is not ideal for writing the procedural scripts required. The use of XML also makes Ant difficult to use with larger projects. The build process is the biggest advantage of Ant.

Apache Maven was released in 2004, designed to address problems developers encountered with Ant. Although it uses XML configurations like Ant, it is better designed relying on conventions and provides subroutines that can be invoked. Unfortunately

the inherent downside of large build scripts remains.

One of the major advantages of Maven is the ability to download libraries and other dependencies over the network. Since this is the main purpose of Maven, custom build scripts are actually more difficult, however, these are not required by most users. [Farcic, 2014]

Gradle, developed in 2012, is the most modern build tool and attempts to combine the power of both Maven and Ant. It uses Groovy (a JVM based language) for build scripts which results in shorter and clearer build scripts and reduces the boilerplate required for creating basic configurations. [gradle.org, 2017] Gradle's flexibility is one of its major advantages. [Casperson, 2015]

The two contenders for the build system become Gradle and Ant. Ant is preferred for developers new to build tools because of its low learning curve, however, as the developers creating Membrane have experience using all three build system, we opt to use Gradle, because of its clearer configurations.

Java Networking

The core feature of Membrane is its ability to communicate with other peers. Using Java core networking libraries for this is an option, however, they provide an unnecessarily granular a control over network communications.

We look at various networking libraries in Java to see which is most appropriate for Membrane. One of the first requirements is the ability to send HTTP requests. These will be required for creating the API to interface with Membrane. Learning two networking libraries to develop Membrane would take more time with very little benefit.

The networking library of choice must also be able to establish SSL connections, which serve both the purpose of User Authentication and provide data security.

There are three major contenders Netty, Vert.x and Akka.

Netty is a low level widely used library that promises quick and easy development

without maintainability or performance issues. [netty.io, 2017] It is based around the reactor pattern to handle requests concurrently, and provides a wide variety of protocols for communication. It is more difficult to debug than traditional networking libraries because of the inherent concurrency, but has a wide variety of features and options.

Vert.x is a polyglot library supporting seven languages including Java. It is built on top of Netty, focusing on a simple concurrency model, removing the complexity of debugging concurrent programs and has a simple asynchronous programming model to allow handling multiple requests at once. It aims to be a JVM alternative for Node.js, a popular JavaScript framework. [vertx.io, 2017]

Akka is a library made for concurrent and distributed JVM applications. It draws on inspiration from Erlang for its actor-based concurrency model, focusing on immutability and allowing actor interactions across multiple hosts when required. [Gupta, 2012]

We opt to use Vert.x in Membrane, it offers all the required features for peer networking with an appropriate concurrency model and without the complexity that comes from using a low-level library.

Java Cryptography

Security was a major concern for potential Membrane users and therefore the library we use for encryption needs to be selected carefully. We again consider three libraries for this purpose. Membrane requires TwoFish encryption support and the ability to generate RSA public private key pairs and certificates, as well as signing and authenticating messages using those keys.

Apache Shiro is an open source security framework that offers authentication, authorisation, cryptography and session management, and aims to provide robust security while being easy to use. It was first released in 2010 and the latest release was 8 months prior to selection. [Shiro, 2017]

Bouncy Castle is a collection of cryptography APIs for Java and C# founded in May 2000. It is maintained by "Legion of the Bouncy Castle Inc.", a registered charity in

Australia and prides itself on its extensive test suite and standards compliance. It is by far one of the most popular cryptography libraries available. [BouncyCastle.org, 2017] The latest release was two months prior to selection.

Keyczar is a newcomer created by Google, aiming to make it easier and safer for developers to use cryptography in their applications, providing simple one line cryptography functions to developers instead of being a general-purpose cryptography library. It was first released in 2010 and the latest release was one month prior to selection. [Willden, 2017]

Shiro is a good option, however, it provides a lot of unnecessary features. Its focus is not aligned with the requirements of Membrane. Keyczar is a good library for limited use-cases, however, Bouncy Castle is best suited to Membrane. It is the oldest library by far, and its popularity during that time makes it much less likely to have bugs.

Version Control

Version Control is an important aspect of any software project. We will be mostly focusing on the extensibility requirement, which requests easy contribution for other open source programmers.

There are two leading platforms *Github* and *Bitbucket*. We opt to use Git for version control because of its clear advantages, allowing for decentralised version control while working from multiple workstation, over competitors such as SVN and CVS.

Github was launched in 2008. Offers code review, issue tracking, documentation, wikis, pull requests and releases. It has 5.8M active users and 19.4M active repositories. [Github.com, 2016] offering multiple integrations for continuous integration and code quality analysis. It also offer public repository hosting for free.

Bitbucket was launched at the same time as Github in 2008 and started only hosting Mercurial projects but quickly expanded to Git as well. [bitbucket.org, 2017] Its main advantage is its close integration with JIRA, a popular issue tracker. [Upguard, 2014]

On balance, given the popularity and available integrations, we opt to use Github for version control. We use four Github integrations in the project. *TravisCI* offers continuous integration, running testing on all targeted JVMs after every commit and notifying developers if a test fails. *VersionEye* ensures that all external dependencies used are up-to-date and have no security flaws. *CodeCov* automatically checks test coverage for every commit, warning the developer if coverage is decreasing due to the commit. Lastly *rtfd.io* provides documentation for the project, hosting the web page.

Other Libraries

We will also use *Google Guava* and *Apache Commons Collections* in Membrane, to enhance the standard Java libraries, offering a wider range of functions and desirable features such as immutable collections.

For marshalling data we opt to use *Jackson*, both for transmission and local storage, serialising objects to both JSON and YAML. Advanced date functionality is added using *Joda* and finally *Junit Jupiter* in conjunction with *Mockito* will be used for unit testing.

Chapter 4

System Components

The components that make up Membrane are all designed individually and connected using Java Interfaces. In this section we discuss, in detail, the goals and functionality of each explaining decisions in the final architecture of Membrane.

We begin with file ingestion and file system monitoring, following into file storage and what inspired the design of the file history storage. Continuing on through Peer Appraisal, how we determine who to give file blocks to we finish in Networking, covering the details of authentication and security.

4.1 File System Monitoring

Monitoring the file system for local changes is key to any backup system. All file changes in the watched folders need to be recorded, including deletions. First we explore methods of walking the file tree when asked to back up recursive folders. We then take two approaches to watching folders, followed by augmentations made to the chosen technique. We finish by discussing methods of deduplication for repeated files and future enhancements that could be made to the implemented monitor.

4.1.1 File Tree Walking

We allow users to select folders for backup, either explicitly, with wild-cards or recursively. When allowing users to specify folders recursively we must inspect the contents of each visited folder and record any encountered folders. This can be achieved using the Visitor pattern in Java, which defines operation for each element visited by the class. [Sugrue, 2010]

Once all of the relevant folders are found we can start watching those folders using one of the methods below. If a folder addition or deletion is observed by the watcher, this can be added or removed to the watched folder list respectively.

One of the problems with this method is when wild-card directories are added. These might not necessarily be created in watched folders so these folders must be manually scanned for on a regular basis.

When finding folders it is important to note the difference between a file or folder, hard link and soft link (also known as a symlink or symbolic link). To understand these we must first observe how files are stored in a Unix-style file system, which supports ordinary files, folders and special files.

Index Nodes (commonly refereed to as inodes) are things in the file system stored by a unique number. Directories are a way to structure and name these inodes. These contain files and each file is mapped to exactly one inode, however, each inode can have more than one file mapped to it. [Bar, 2001]

A hard link is an inode that has multiple names in the file system. A symlink is any file that contains a reference to another file or directory which will affect path name resolution. The link is an inode that exists independent of the target, and does not affect the target if deleted and continues pointing to a non-existent target when that is deleted. [Yue et al., 2011] These can be used to link different file systems, but they change the file system from a tree into a directed graph, which greatly complicate file tree walking as files can loop and repeat.

Within Membrane we opt to ignore sym-

links while walking the file tree. Users are able to specify the target of the symlink if required. To prevent backup duplication due to hard links we use a system that mirrors the inodes described, but treat them as different files for backup.

4.1.2 Polling

The first naive methods for check for folder changes simply uses polling to record the contents for folders, finding the difference between the last scan and the current scan. To check whether a file has changed the modification data provided in the file's metadata is used.

This is a simple method to implement, however it consumes a lot of memory when implemented as it needs to retain a copy of the last modification date and all of the files in the watched folders.

This method is also IO intensive. The file listing for a directory on the file system contains no file metadata, so every single inode in the folder needs to be accessed to gather it's metadata. This mostly affects potential users using storage media with poor IOPS (Input Output Operations per Second), common in older hard disk drives (HDDs). [Mansurov, 2017]

This makes polling a poor solution for Membrane, as during user surveys low resource usage was requested. Periodic spikes in IOPS could adversely impact user experience in other software.

4.1.3 Native Watching

A more advanced method of watching files is built into some file systems. This sends out File Change Notifications to subscribed services when an inode changes. There are three common scenarios when this is used. When caching a model of file systems objects, logging file system activity and gate-keeper services that ensure only permitted operations happen on the file system. [Kerrisk, 2014]

The first implementation of file change notification, *dnotify*, appeared in Linux in 2001, it suffered from a cumbersome API and most notably, it kept file descriptors

open causing problems when trying to unmount the file system. [Kerrisk, 2014]

The *inotify* API was released in 2005 and aimed to address all of the issue lessons learned in *dnotify*. It consists of three dedicated system calls *inotify_init()* which maintains a list of file system objects that should be watched and a list of events generated for those objects. The commands *inotify_add_watch()* and *inotify_rm_watch()* are used to alter the list of monitored file system objects. *inotify* provides more information than *dnotify*, allows programs to deal with notifications synchronously and includes the file name modified. [Kerrisk, 2014]

Java implements the WatchService API that provides low level access to the underlying *inotify* API of the file system. [Oracle, 2017] We opt to use this when the API is available, but fall back to polling when *inotify* or similar services are unavailable.

4.1.4 Watching Enhancements

Despite the comprehensive feature set of *inotify*, we still require some enhancements to prevent accidental or missed file system event notifications.

In order to detect changes between Membrane runs we need to load all known and expected files into memory and scan the file system once when Membrane loads. This means resorting back to maintaining a list of all monitored files in local memory, an unfortunate but necessary step.

We now match any reported modified or deleted files against the internal state, using the modification date and hash of the file to determine if the file has been changed. In order to lower resource usage we read any modified files in sections, never loading more than 4MBs into local memory. These chunks become shards within the backup system.

4.1.5 Deduplication

There are two viable approaches to file deduplication for Membrane. We can choose to split the file into shards and check if shard hashes match, or use a rolling checksum, similar to Rsync to find the section

of the data changed, and only storing that. [Tridgell et al., 1996]

Simply splitting the file into chunks has the disadvantage of being unable to detect overlapping data, which can lead to huge duplication. When data is inserted into the middle of a file any blocks after and including the change will register as modified and need to be restored.

The approach of simply sharding the data is simple and much less prone to bugs, so we take this approach with the initial implementation of Membrane, although we will maintain support for variable size chunks for future implementations.

We need a methods of generating a unique has for file shards. Git and Subversion opt to use SHA1 [Torvalds and Hamano, 2010] to generate unique keys for files, however, we have decided to use MD5.

Both SHA1 [Wang et al., 2005] and MD5 [Stevens, 2006] are cryptographically broken, meaning that it is possible to generate a collision¹ without having to try every possible input into the hashing function. It is also possible to launch a first preimage attack on these hashes, something that would make them unusable for security as an attacker would be able to generate an input matching any given hash. As Membrane only uses this hash as an internal shard identifier, we do not have to be concerned about an external attacker.

The only situation in which this may cause issues in the future, is if the user stores two shards with the same hash accidentally, however, the chances of that are minute enough that it is not worth worrying about. This was discussed in detail by Torvalds [2006] and other git developers. An inadvertent collision was deemed so unlikely that it was reasonable to use SHA1 for Git.

MD5 only has a 128 bit output vs. the 160 bit output of SHA1, meaning collisions are slightly more likely, however we mostly concerned about hashing speed, seeing as this hash will be run very frequently on any files that have been altered, and studies have shown that MD5 is on average about 1.75 times faster than SHA1. [SAPHIR, 2007]

¹Two files matching a given hash

4.2 Local Storage System

As Membrane is a backup system, it needs to contain a log of file activity, including file meta-data and contents at that point in it's life time. In this section we first discuss the shard storage mechanism, followed by the file history persistence, and finish on the file expiry policy used in Membrane.

4.2.1 Shard Storage

When a file is backed up the file watcher shards the file into 4MB chunks and sends them directly to shard storage. The shard store manages a folder of the file system and provides functions for shard storage and retrieval.

When a shard is inserted into storage the file system first checks if the shard is already stored. If so the shard is ignored altogether and the write is avoided, saving disk IO. Otherwise the hash of the data is confirmed and the shard is written to a file on the disk.

Because of file system limitations we use a hierarchical directory structure for storing shards. A new directory is created every five characters of the shard, overcoming any restrictions on the maximum number of files per directory, of older file systems such as ext2 and ext3 which reportedly have performance issues with anything above 10,000 files. [Johnson, 2014]

The shard storage runs scrubbing on a regular basis, a concept taken from the ZFS file system. [Oracle, 2012] The shard store periodically checks that every shard stored is consistent with it's hash and accessible. This is a technique commonly used to prevent error before they result in failure. Although the shard store also verifies shard integrity during the read it may falsely report damaged shards as available to the user. This prevents the shard from being patched by versions stored in the peer swarm.

4.2.2 File History Storage

The file history is the central store of the backup system. It contains a log of every file addition, modification and removal providing all of the information required to re-

trieve it. It is important that the store is immediately persisted to the file system upon backup so nothing is lost upon unexpected shutdowns.

The naive approach to this problem uses a log to record these event, implemented in the form of a linked list, flushing entries immediately to a log file. Although this approach satisfies all of the requirements it is very expensive to search and quickly becomes very large.

Taking inspiration from distributed file systems such as HDFS [Shvachko et al., 2010] and GFS [McKusick and Quinlan, 2010] which use NameNodes to store file system modification in a checkpoint and journal. The journal is a write-ahead commit log that takes the place of the aforementioned event log. The checkpoint is a known file system state. In the event of system failure these file systems can replay the log to a checkpoint.

Within Membrane we take advantage of checkpoints to speed up search through the journal. The start and end of the journal are often queried to check current backup status, so these are given permanent checkpoints, that are always kept up to date. When the user queries for a specific point in time, the journal can be replayed from the start checkpoint to the requested time, quickly generating the required time slice.

This proved to be an effective way of storing file history, providing quick responses to the GUI, CLI and other system components, even when full of data. If this proves insufficient for future demands, indexes for specific queries can be created for quick search.

It is important to address the distinct lack of a transactional database, in a situation where the use of one seems like a natural fit. Traditional databases such as MySQL have very high memory and CPU usage [James, 2017] and although this usage can be tuned, running a full database on a users computer would go against the resource usage requirement.

4.2.3 Backup Management

One of Membrane’s proposed features is flexible file history. Instead of providing a

backup storage duration, backups are stored until the space allocated to Membrane is completely used.

This is a step towards the maintenance free experience requested by potential users. Many cloud solutions such as Dropbox [2017] allow users to access version history for a set amount of time, requiring extra subscriptions for additional versioning time.

In order to implement this we take the concepts used in garbage collection within programming languages and apply them to the file history. In addition to the maximum shard storage size, a target size is set. The file system monitor will add shard until the maximum limit, regardless of the requested target size.

A history compacting and garbage collection is then executed on a regular basis. This is a three step process that attempts to reach the target storage size in the least destructive way using:

1. Garbage Collection
2. Remove Unwatched Files
3. Trim Older Journal using FIFO

Garbage collection is implemented using an adapted version of Dijkstra et al. [1978]’s mark and sweep algorithm. This works in three passes.

1. Certain shards have are protected from garbage collection as they have just been added into shard storage and therefore might not have their reference inserted into the journal yet, generating set P .
2. The second pass walks the journal and removes shards mentioned in the journal creating set R
3. The final pass finds $S \cap (P \cup R)^C$ where S is the set of all stored shards, and removes them.

This step is intended to remove any shards that were inserted into shard storage but never followed up with a file entry. This would be possible in the case of error during shard storage or file history storage. The

code for garbage collection can be found in listing 7.1.

The next step is intended to remove any entries the user is not interested in any more. These could have been inserted after recovery from peers or may have been files that were temporarily backed up. If any files were deleted garbage collection is re-run to remove the relevant shards.

Finally the Journal is trimmed, removing any history necessary. The algorithm creates a shard count dictionary to see if removing the last file in the log will affect the storage. If it does it is removed, otherwise the algorithm moves along to the next entry. This is repeated until the target storage size is reached. The method used means no file entries are unnecessarily removed. The storage clamping procedure can be found in listing 7.2.

4.3 Peer Appraisal

4.4 Networking

Chapter 5

User Interfaces

5.1 GUI

5.2 CLI

Chapter 6

Results & Evaluation

6.1 Unit Testing

6.2 Empirical Testing

6.3 Summary

Chapter 7

Conclusion

7.1 Limitations

7.1.1 Limitations of the Study

7.1.2 Limitations of the System

7.2 Further Work

7.2.1 Expansion 1

7.2.2 Expansion 2

7.2.3 Expansion 3

7.3 Summary

Bibliography

- Antonelli, G. A. [2008], ‘Non-monotonic logic’, *Stanford Encyclopedia of Philosophy* .
- archlinux.org, w. [2017], ‘Synchronization and backup programs’, *URL* https://wiki.archlinux.org/index.php?title=Synchronization_and_backup_programs&oldid=473791 . [Accessed April 12th, 2017].
- Aycock, J. [2003], ‘A brief history of just-in-time’, *ACM Computing Surveys (CSUR)* **35**(2), 97–113.
- Balke, T. and Eymann, T. [2009], ‘Using institutions to bridge the trust-gap in utility computing markets—an extended “trust-game”’.
- Banahan, M., Brady, D. and Doran, M. [1988], *The C book*, number ANSI-X-3-J-11-DRAFT, Addison-Wesley New York.
- Bar, M. [2001], *Linux file systems*, McGraw-Hill Professional.
- Batters, M. [2016], ‘Dropbox security comment’, *URL* <https://www.legaltechnology.com/latest-news/security-comment-why-are-people-still-using-dropbox-for-business/> .
- Beer, M., D’inverno, M., Luck, M., Jennings, N., Preist, C. and Schroeder, M. [1999], ‘Negotiation in multi-agent systems’, *The Knowledge Engineering Review* **14**(03), 285–289.
- bitbucket.org [2017], ‘Bitbucket site’, *URL* <https://bitbucket.org/> . [Accessed April 25th, 2017].
- Blank, S. [2010], ‘Minimum feature set’, *URL* <https://steveblank.com/2010/03/04/perfection-by-subtraction-the-minimum-feature-set/> .
- Boucadair, M., Penno, R. and Wing, D. [2013], ‘Universal plug and play (upnp) internet gateway device-port control protocol interworking function (igd-pcp iwf)’.
- BouncyCastle.org [2017], ‘Bouncycastle site’, *URL* <https://bouncycastle.org/> . [Accessed April 24th, 2017].
- Bound, J. and Rekhter, Y. [1997], ‘Dynamic updates in the domain name system (dns update)’.
- Casperson, M. [2015], ‘Maven vs gradle one year later’, *URL* <https://dzone.com/articles/maven-vs-gradle-one-year-later> . [Accessed April 24th, 2017].
- Cockburn, A. [1997], ‘Structuring use cases with goals’, *Journal of object-oriented programming* **10**(5), 56–62.

- coffeescript.org [2017], ‘Coffeescript faq’, *URL* <http://coffeescript.org/> . [Accessed April 23th, 2017].
- Cohen, B. [2008], ‘The bittorrent protocol specification’.
- DataArt [2014], ‘What is typescript - pros and cons’, *URL* <https://designmodo.com/typescript/> . [Accessed April 23th, 2017].
- Deering, S. E. [1998], ‘Internet protocol, version 6 (ipv6) specification’.
- Dierks, T. [2008], ‘The transport layer security (tls) protocol version 1.2’.
- Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S. and Steffens, E. F. [1978], ‘On-the-fly garbage collection: An exercise in cooperation’, *Communications of the ACM* **21**(11), 966–975.
- Dropbox [2017], ‘Recover older versions of files’, *URL* <https://www.dropbox.com/en/help/11> . [Accessed April 26th, 2017].
- Faratin, P. [2000], Automated service negotiation between autonomous computational agents, PhD thesis, University of London.
- Farcic, V. [2014], ‘Java build tools: Ant vs maven vs gradle’, *URL* <https://technologyconversations.com/2014/06/18/build-tools/> . [Accessed April 24th, 2017].
- Farina, J., Scanlon, M. and Kechadi, M.-T. [2014], ‘Bittorrent sync: First impressions and digital forensic implications’, *Digital Investigation* **11**, S77–S86.
- Fatima, S. S., Wooldridge, M. and Jennings, N. R. [2002], Multi-issue negotiation under time constraints, in ‘Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1’, ACM, pp. 143–150.
- filecoin.io [2014], ‘Filecoin: A cryptocurrency operated file storage network’, *URL* <http://filecoin.io/filecoin.pdf> .
- Finin, T., Weber, J. et al. [1992], ‘Specification of the kqml agent-communication language’.
- Fipa, A. [2002], ‘Fipa acl message structure specification’, *Foundation for Intelligent Physical Agents*, <http://www.fipa.org/specs/fipa00061/SC00061G.html> (30.6. 2004) .
- Flanagan, D. [2011], *JavaScript: The definitive guide: Activate your web pages*, ” O’Reilly Media, Inc.”.
- Ford, B., Srisuresh, P. and Kegel, D. [2005], Peer-to-peer communication across network address translators., in ‘USENIX Annual Technical Conference, General Track’, pp. 179–192.
- Gamma, E. [1995], *Design patterns: elements of reusable object-oriented software*, Pearson Education India.
- Ghemawat, S., Gobioff, H. and Leung, S.-T. [2003], ‘The google file system’, *SIGOPS Oper. Syst. Rev.* **37**(5), 29–43.
URL: <http://doi.acm.org/10.1145/1165389.945450>
- Github.com [2016], ‘Github octoverse 2016’, *URL* <https://octoverse.github.com/> . [Accessed April 25th, 2017].

- golang.org [2017], ‘Go faq’, *URL* <https://golang.org/doc/faq> . [Accessed April 23th, 2017].
- Gosling, J., Joy, B., Steele, G. L., Bracha, G. and Buckley, A. [2014], *The Java language specification*, Pearson Education.
- Gosling, J. and McGilton, H. [1995], ‘The java language environment’, *Sun Microsystems Computer Company* **2550**.
- gradle.org [2017], ‘Maven vs gradle’, *URL* <https://gradle.org/maven-vs-gradle> . [Accessed April 24th, 2017].
- Group, I. . S. W. et al. [n.d.], ‘Ieee standard for local and metropolitan area networks: media access control (mac) bridges’, *IEEE Std* **802**.
- Gupta, M. [2012], *Akka essentials*, Packt Publishing Ltd.
- Hericourt, O. and Le Pennec, J.-F. [2001], ‘Method and system for using with confidence certificates issued from certificate authorities’. US Patent App. 10/007,750.
- Herrero, P., Bosque, J. L. and Pérez, M. S. [2007], An agents-based cooperative awareness model to cover load balancing delivery in grid environments, *in* ‘OTM Confederated International Conferences” On the Move to Meaningful Internet Systems”’, Springer, pp. 64–74.
- Holdrege, M. and Srisuresh, P. [2001], ‘Rfc 3027’, *Protocol Complications with the IP Network Address Translator* .
- Hunt, J. W. and MacIlroy, M. [1976], *An algorithm for differential file comparison*, Cite-seer.
- Ilieva, J., Baron, S. and Healey, N. M. [2002], ‘Online surveys in marketing research: Pros and cons’, *International Journal of Market Research* **44**(3), 361.
- James, R. [2017], ‘Mysql memory allocation’, *URL* <http://mysql.rjweb.org/doc.php/memory> . [Accessed April 26th, 2017].
- Jennings, N. R., Faratin, P., Lomuscio, A. R., Parsons, S., Wooldridge, M. J. and Sierra, C. [2001], ‘Automated negotiation: prospects, methods and challenges’, *Group Decision and Negotiation* **10**(2), 199–215.
- Jennings, N. R., Parsons, S., Noriega, P. and Sierra, C. [1998], On argumentation-based negotiation, *in* ‘Proceedings of the International Workshop on Multi-Agent Systems’, pp. 1–7.
- Johnson, C. [2014], ‘How many files can be put into a directory?’, *URL* <https://stackoverflow.com/questions/466521/how-many-files-can-i-put-in-a-directory> . [Accessed April 26th, 2017].
- Jones, B. [2015], ‘Bittorrent’s dht turns 10 years old’, *URL* <https://torrentfreak.com/bittorrents-dht-turns-10-years-old-150607/> .
- Jøsang, A. and Golbeck, J. [2009], Challenges for robust trust and reputation systems, *in* ‘Proceedings of the 5th International Workshop on Security and Trust Management (SMT 2009), Saint Malo, France’.
- Keller, A. and Ludwig, H. [2002], Defining and monitoring service-level agreements for dynamic e-business., *in* ‘Lisa’, Vol. 2, pp. 189–204.

- Kerrisk, M. [2014], ‘Filesystem notification, part 1: An overview of dnotify and inotify’, URL <https://lwn.net/Articles/604686/>. [Accessed April 25th, 2017].
- Kindler, E. and Krivy, I. [2011], ‘Object-oriented simulation of systems with sophisticated control’, *International Journal of General Systems* **40**(3), 313–343.
- Klingberg, T. and Manfredi, R. [2002], ‘The gnutella protocol specification v0. 6’, *Technical specification of the Protocol*.
- Krasner, G. and Pope, S. [1988], ‘A cookbook for using the model-view controller user interface paradigm in smalltalk-80 j. object oriented program., 1, 26-49’.
- Kraus, S. [2001], *Strategic negotiation in multiagent environments*, MIT press.
- Kraus, S., Sycara, K. and Evenchik, A. [1998], ‘Reaching agreements through argumentation: a logical model and implementation’, *Artificial Intelligence* **104**(1-2), 1–69.
- Kreft, K. and Langer, A. [2003], ‘After java and c sharp - what is next?’, URL <http://www.artima.com/weblogs/viewpost.jsp?thread=6543>. [Accessed April 23th, 2017].
- Labrou, Y., Finin, T. and Peng, Y. [1999], ‘Agent communication languages: The current landscape’, *IEEE Intelligent Systems and Their Applications* **14**(2), 45–52.
- Langford, J. [2001], ‘Multiround rsync’.
- Lynley, M. [2015], ‘Dropbox 400m users’, URL <https://techcrunch.com/2015/06/24/dropbox-hits-400-million-registered-users/>.
- MacCaw, A. [2012], *The Little Book on CoffeeScript*, ” O’Reilly Media, Inc.”.
- Malan, R. and Bredemeyer, D. [2001], ‘Functional requirements and use cases’, *Bredemeyer Consulting*.
- Mansurov, N. [2017], ‘Nvme vs ssd vs hdd’, URL <https://photographylife.com/nvme-vs-ssd-vs-hdd-performance/>. [Accessed April 25th, 2017].
- Mayo, M. A., Neemann, T., Pearson, H., Sekhar, C. C. and Toraason, D. [2008], ‘Security session authentication system and method’. US Patent 7,356,694.
- McDermott, D. and Doyle, J. [1980], ‘Non-monotonic logic i’, *Artificial intelligence* **13**(1-2), 41–72.
- McKusick, K. and Quinlan, S. [2010], ‘Gfs: evolution on fast-forward’, *Communications of the ACM* **53**(3), 42–49.
- Mushtaque, M. A., Dhiman, H., Hussain, S. and Maheshwari, S. [2014], ‘Evaluation of des, tdes, aes, blowfish and two fish encryption algorithm: based on space complexity’, *International Journal of Engineering Research & Technology (IJERT)* **3**(4).
- Muthitacharoen, A., Chen, B. and Mazieres, D. [2001], A low-bandwidth network file system, in ‘ACM SIGOPS Operating Systems Review’, Vol. 35, ACM, pp. 174–187.
- Neglia, G., Reina, G., Zhang, H., Towsley, D., Venkataramani, A. and Danaher, J. [2007], Availability in bittorrent systems, in ‘IEEE INFOCOM 2007-26th IEEE International Conference on Computer Communications’, IEEE, pp. 2216–2224.
- netty.io [2017], ‘Netty’, URL <https://netty.io/>. [Accessed April 24th, 2017].

- Noy, N. F., McGuinness, D. L. et al. [2001], ‘Ontology development 101: A guide to creating your first ontology’.
- Oracle [2012], ‘Oracle solaris zfs administration guide’, *URL* https://docs.oracle.com/cd/E23823_01/html/819-5461/gbbwa.html . [Accessed April 26th, 2017].
- Oracle [2017], ‘Watching a directory for changes’, *URL* <https://docs.oracle.com/javase/tutorial/essential/io/notification.html> . [Accessed April 25th, 2017].
- Osborne, M. J. and Rubinstein, A. [1994], *A course in game theory*, MIT press.
- Paletta, M. and Herrero, P. [2009], A mas-based negotiation mechanism to deal with service collaboration in cloud computing, in ‘Intelligent Networking and Collaborative Systems, 2009. INCOS’09. International Conference on’, IEEE, pp. 147–153.
- Pande Joshi, K. and Finin, T. [2012], ‘Ontology for cloud services sla (service level agreement)’, *URL* <http://ebiquity.umbc.edu/resource/html/id/344/Ontology-for-Cloud-Services-SLA-Service-Level-Agreement-> .
- Pinyol, I. and Sabater-Mir, J. [2013], ‘Computational trust and reputation models for open multi-agent systems: a review’, *Artificial Intelligence Review* **40**(1), 1–25.
- Plummer, D. [1982], ‘Ethernet address resolution protocol: Or converting network protocol addresses to 48. bit ethernet address for transmission on ethernet hardware’.
- Porter, M. A. [2012], ‘Small-world network’, *Scholarpedia* **7**(2), 1739.
- Postel, J. et al. [1981], ‘Rfc 792: Internet control message protocol’, *InterNet Network Working Group* .
- purl.org [2001], ‘Datastore schema’, *URL* <https://www.w3.org/2001/05/rdf-ds/datastore-schema> .
- Qiu, D. and Srikant, R. [2004], Modeling and performance analysis of bittorrent-like peer-to-peer networks, in ‘ACM SIGCOMM computer communication review’, Vol. 34, ACM, pp. 367–378.
- Rahwan, I. [2005], *Interest-based negotiation in multi-agent systems*, Citeseer.
- Rasch, D. and Burns, R. C. [2003], In-place rsync: File synchronization for mobile and wireless devices., in ‘USENIX Annual Technical Conference, FREENIX Track’, Vol. 100.
- Rizvi, S., Hussain, S. Z. and Wadhwa, N. [2011], Performance analysis of aes and twofish encryption schemes, in ‘Communication Systems and Network Technologies (CSNT), 2011 International Conference on’, IEEE, pp. 76–79.
- Rodriguez, A. [2008], ‘Restful web services: The basics’, *IBM developerWorks* .
- Rosenberg, J., Mahy, R. and Huitema, C. [2005], Traversal using relay nat (turn), Technical report, October 2003. Internet-Draft (Work in Progress).
- Rosenschein, J. S. and Zlotkin, G. [1994], *Rules of encounter: designing conventions for automated negotiation among computers*, MIT press.

- Ross, R. B., Thakur, R. et al. [2000], Pvfs: A parallel file system for linux clusters, in ‘Proceedings of the 4th annual Linux showcase and conference’, pp. 391–430.
- Sabater, J. and Sierra, C. [2001], Regret: reputation in gregarious societies, in ‘Proceedings of the fifth international conference on Autonomous agents’, ACM, pp. 194–195.
- Sandholm, T. [2002], ‘Algorithm for optimal winner determination in combinatorial auctions’, *Artificial intelligence* **135**(1-2), 1–54.
- SAPHIR, P. R. [2007], ‘Sécurité et analyse des primitives de hachage innovantes et récentes’.
- Saxena, N., Tsudik, G. and Yi, J. H. [2003], Admission control in peer-to-peer: design and performance evaluation, in ‘Proceedings of the 1st ACM workshop on Security of ad hoc and sensor networks’, ACM, pp. 104–113.
- Schmidt, B. K., Lam, M. S. and Northcutt, J. D. [1999], The interactive performance of slim: a stateless, thin-client architecture, in ‘ACM SIGOPS Operating Systems Review’, Vol. 33, ACM, pp. 32–47.
- Schneier, B. [2007], ‘Bruce almighty: Schneier preaches security to linux faithful’, *URL* https://www.computerworld.com.au/article/46254/bruce_almighty_schneier_preaches_security_linux_faithful.
- Schollmeier, R. [2001], A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications, in ‘Peer-to-Peer Computing, 2001. Proceedings. First International Conference on’, IEEE, pp. 101–102.
- Sharma, S., Gopalan, K., Nanda, S. and Chiueh, T.-c. [2004], Viking: A multi-spanning-tree ethernet architecture for metropolitan area and cluster networks, in ‘INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies’, Vol. 4, IEEE, pp. 2283–2294.
- Shiro, A. [2017], ‘Apache shiro’, *URL* <https://shiro.apache.org/>. [Accessed April 24th, 2017].
- Shvachko, K., Kuang, H., Radia, S. and Chansler, R. [2010], The hadoop distributed file system, in ‘2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)’, pp. 1–10.
- Slade, C. [2002], ‘Reasons to buy: The logic of advertisements’, *Argumentation* **16**(2), 157–178.
- Srisuresh, P. and Holdrege, M. [1999], ‘Ip network address translator (nat) terminology and considerations’.
- stackoverflow.com, s. [2017], ‘Developer survey 2017’, *URL* <https://stackoverflow.com/insights/survey/2017>. [Accessed April 23th, 2017].
- Stefanov, S. [2010], *JavaScript Patterns: Build Better Applications with Coding and Design Patterns*, ” O’Reilly Media, Inc.”.
- Stephen, Z. H., Blackburn, S. M., Kirby, L. and Zigman, J. [2000], Platypus: Design and implementation of a flexible high performance object store., in ‘In Proceedings of the Ninth International Workshop on Persistent Object Systems’, Citeseer.
- Stevens, M. [2006], ‘Fast collision attack on md5.’, *IACR Cryptology ePrint Archive* **2006**, 104.

- Sugrue, J. [2010], ‘Visitor pattern tutorial with java examples’, *URL* <https://dzone.com/articles/design-patterns-visitor> . [Accessed April 25th, 2017].
- Sugumaran, V. and Storey, V. C. [2002], ‘Ontologies for conceptual modeling: their creation, use, and management’, *Data & knowledge engineering* **42**(3), 251–271.
- Thakur, J. and Kumar, N. [2011], ‘Des, aes and blowfish: Symmetric key cryptography algorithms simulation based performance analysis’, *International journal of emerging technology and advanced engineering* **1**(2), 6–12.
- Torvalds, L. [2006], ‘Hash collisions in git’, *URL* <https://marc.info/?l=git&m=115678778717621&w=2> . [Accessed April 25th, 2017].
- Torvalds, L. and Hamano, J. [2010], ‘Git: Fast version control system’, *URL* <http://git-scm.com> .
- Tratt, L. [2009], ‘Dynamically typed languages’, *Advances in Computers* **77**, 149–184.
- Tridgell, A., Mackerras, P. et al. [1996], ‘The rsync algorithm’.
- Tseng, Y.-C., Ni, S.-Y., Chen, Y.-S. and Sheu, J.-P. [2002], ‘The broadcast storm problem in a mobile ad hoc network’, *Wirel. Netw.* **8**(2/3), 153–167.
URL: <http://dx.doi.org/10.1023/A:1013763825347>
- typescriptlang.org [2017], ‘Typescript website’, *URL* <https://www.typescriptlang.org/> . [Accessed April 23th, 2017].
- Upguard [2014], ‘Github vs bitbucket’, *URL* <https://www.upguard.com/articles/github-vs-bitbucket> . [Accessed April 25th, 2017].
- Varian, H. R. [1995], Economic mechanism design for computerized agents., in ‘USENIX workshop on Electronic Commerce’, New York, NY, pp. 13–21.
- vertx.io [2017], ‘Vert.x’, *URL* <http://vertx.io/> . [Accessed April 24th, 2017].
- Vuze [2010], ‘Peer exchange’, *URL* http://wiki.vuze.com/w/Peer_Exchange .
- Wang, X., Yin, Y. L. and Yu, H. [2005], ‘Collision search attacks on sha1’.
- Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. and Maltzahn, C. [2006], Ceph: A scalable, high-performance distributed file system, in ‘Proceedings of the 7th symposium on Operating systems design and implementation’, USENIX Association, pp. 307–320.
- Wheeler, D. [2012], ‘Dropbox dives into coffeescript’, *URL* <https://blogs.dropbox.com/tech/2012/09/dropbox-dives-into-coffeescript/> . [Accessed April 23th, 2017].
- Wilkinson, S., Boshevski, T., Brandoff, J. and Buterin, V. [2014], ‘Storj a peer-to-peer cloud storage network’.
- Willden, S. [2017], ‘Keyczar github’, *URL* <https://github.com/google/keyczar> . [Accessed April 24th, 2017].
- Wing, D., Matthews, P., Rosenberg, J. and Mahy, R. [2010], ‘Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun)’.
- Wooldridge, M. [2009], *An introduction to multiagent systems*, John Wiley & Sons.

- Yan, J., Kowalczyk, R., Lin, J., Chhetri, M. B., Goh, S. K. and Zhang, J. [2007], ‘Autonomous service level agreement negotiation for service composition provision’, *Future Generation Computer Systems* **23**(6), 748–759.
- Yue, Y., Guo, L. et al. [2011], Unix file system, *in* ‘UNIX Operating System’, Springer, pp. 149–185.

Appendix

Listing 7.1: Garbage Collection Implementation

```
/**
 * Removes any shards not referenced in the catalogue from the storage.
 */
long collectGarbage() {
    logger.info("Garbage collection - Start");
    Set<String> requiredShards = fileCatalogue.getReferencedShards();
    Set<String> garbageShards = shardStorage.listShardIds();
    garbageShards.removeAll(requiredShards);
    garbageShards.removeAll(tempProtectedShards);
    logger.info("Garbage collection - Found {} de-referenced shards",
        garbageShards.size());
    AtomicLong removedSize = new AtomicLong(0L);
    garbageShards.stream()
        .peek(x -> logger.info("Garbage collection - Removing shard: [{}]",
            x))
        .forEach(x -> {
            try {
                long fileSize = shardStorage.removeShard(x);
                removedSize.addAndGet(fileSize);
            } catch (ShardStorageException e) {
                // Ignore - It doesn't exist already
            }
        });
    logger.info("Garbage collection - Complete - Removed {}MB", ((float)
        removedSize.get()) / (1024 * 1024));
    return removedSize.get();
}
```

Listing 7.2: Storage Clamping Implementation

```
public synchronized long clampStorageToSize(long bytes, Set<Path>
    trackedFolders) throws StorageManagerException {
    long currentStorageSize = getStorageSize();
    long spaceToRecover = currentStorageSize - bytes;
    logger.info("Space Recovery - Reducing storage to {}MB. Current size {}MB.
        Need to remove {}MB", ((float) bytes) / (1024 * 1024), ((float)
        currentStorageSize) / (1024 * 1024), ((float) Math.max(spaceToRecover,
        0)) / (1024 * 1024));
    if (spaceToRecover > 0) {
        logger.info("Space Recovery - Collecting unnecessary shards.");
        spaceToRecover -= collectGarbage();
    }

    if (spaceToRecover > 0) {
        logger.info("Space Recovery - Finding un-tracked files.");
        Set<Path> notTrackedFiles = fileCatalogue.getCurrentFiles().stream()
```

```
        .filter(x -> !trackedFolders.contains(x.getParent()))
        .collect(Collectors.toSet());
    if (notTrackedFiles.size() > 0) {
        logger.info("Space Recovery - Removing {} un-tracked files.",
            notTrackedFiles.size());
        notTrackedFiles.forEach(x -> fileCatalogue.forgetFile(x));
        spaceToRecover -= cleanStorage(fileCatalogue.getOldestJournalEntryTime());
    } else {
        logger.info("Space Recovery - No un-tracked files found.",
            notTrackedFiles.size());
    }
}

if (spaceToRecover > 0) {
    logger.info("Space Recovery - Retiring older journal entries.");
    fileCatalogue.removeOldestJournalEntries((int) spaceToRecover);
    spaceToRecover -= cleanStorage(fileCatalogue.getOldestJournalEntryTime());
}

if (spaceToRecover > 0) {
    logger.warn("Space Recovery - Could not meet {}MB requirement. Excess is
        {}MB.", ((float) bytes) / (1024 * 1024), ((float) spaceToRecover) /
        (1024 * 1024));
} else {
    logger.info("Space Recovery - Successful file size reduction. Reduced to
        {}MB.", ((float) (bytes + spaceToRecover)) / (1024 * 1024));
}
return spaceToRecover;
}
```
