CloudBox

Private, Reliable and Distributed Storage

Rafael Vassalo Cortês

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Doctor Pedro Miguel dos Santos Alves Madeira Adão
Doctor Luís Manuel Antunes Veiga

Examination Committee

Chairperson: Doctor João Emilio Segurado Pavão Martins
Supervisor: Doctor Pedro Miguel dos Santos Alves Madeira Adão
Member of the Committee: Doctor Carlos Nuno da Cruz Ribeiro

November 2015

Acknowledgments

I would like to thank to all of those without concluding this thesis would not have been possible. To those who contributed directly to it, and also to those whose support has made endure to the end.

First of all, I would like to thank my advisors, Professors Pedro Adão and Luís Veiga, for guiding me through the labyrinths of this project. It truly would have not been possible without them. They were always available and brought fresh perspectives and new ideas to the work when I was shortsighted and overwhelmed.

I am also infinitely grateful to my parents Carlos and Graça, to family and to my friends, for their patience and understanding, when I could do nothing but work on my thesis; for the sacrifices they made so that I could keep my peace of mind. They were the ones from where I drew strength during the hardest days.

Abstract

This work describes a solution for cooperative storage of files. Since it is intended to store data remotely in non-trusted nodes, it is necessary to take steps to protect the users privacy. It is shown how to build a system, Cloudbox, capable of storing files in a distributed network, respecting the inherent privacy issues. The Cloudbox system is also capable of supporting groups: sets of files whose access is restricted to a set of users. The resulting system allows a user to store their data in the cloud, ensuring that the contents of his files are safe. It takes into account the transitional nature of distributed networks and presence of possible attackers. This dissertation quantifies the impact of several techniques used on the performance of a possible implementation, taking into account various parameterizations.

Keywords: Peer-to-peer, Secure storage, Distributed, Community Cloud, Cooperative

Resumo

Este trabalho descreve uma solução para armazenamento cooperativo de ficheiros. Dado que existe intenção para armazenar dados remotamente em nós não confiáveis, torna-se necessário tomar medidas para proteger a privacidade dos utilizadores.

É demonstrado como construir um sistema, o Cloudbox, capaz de armazenar ficheiros de forma distribuída, respeitando as questões de privacidade inerentes. O Cloudbox é também capaz de suportar grupos: conjuntos de ficheiros cujo acesso é restrito a um conjunto de utilizadores.

O sistema resultante permite que um utilizador guarde os seus dados na nuvem, assegurando-se de que o conteúdo dos seus ficheiros está seguro. Tem em conta a natureza transitória de uma rede distribuída e da presença na rede de possíveis atacantes. Esta dissertação quantifica o impacto das várias técnicas utilizadas sobre o desempenho de uma possível implementação, tendo em conta várias parametrizações.

Palavras-Chave: Partilha P2P, Armazenamento seguro, Distribuído, Nuvem comunitária, Cooperativo

Contents

List of Tables

List of Figures

Acronyms

API Application Programming Interface. 20, 31, 33, 34

CLI Command Line Interface. 33

ORM Object-relational mapping. 34

P2P Peer-to-peer. 4, 25, 26, 31–33, 35

PubSub Publish-Subscribe. 34

UI User Interface. 33, 36

Many years ago the great British explorer George Mallory,
who was to die on Mount Everest,
was asked why did he want to climb it.
He said, "Because it is there."

Well, space is there.

— John F. Kennedy

Chapter 1

Introduction

In recent years, public cloud storage has been popularized by providers that offer free services. The commodity of having content synchronized across devices and safe from local hardware failure seems to have been enough for users to overlook the privacy implications of using such a service.

The concept of Community Cloud, where a group of interested parties contribute with computational resources towards a common system, from which they all benefit, is an alternative storage paradigm that can yield the same benefits as the public cloud approach, and, furthermore, it can be extended to solve its weaknesses.

The existing cloud storage systems either have weak security, require trusting confidential data to third-parties or, are unable to efficiently deal with dynamic content, or have none to poor collaboration functionalities. There are a lot of solutions available but none simultaneously solves the mentioned problems.

The solution proposed in this thesis is designed for a distributed peer-to-peer environment with secure storage even when handed to untrusted devices whilst providing efficient support to content modification and sharing among dynamic groups.

1.1 Motivation

The emergence of these cloud storage systems is explained by a need developed throughout the last decade due to the appearance of new consumer devices, like camera phones, digital cameras and other multimedia devices, that generate new forms of media. Each year new hardware versions of such devices increase their quality resulting in an increase of disk space required to store their artifacts. Storing this media online enables us to share them with others and gives us peace of mind knowing that it is safe from local hardware failure.

Finding new ways to provide scalable storage systems is thus a critical problem if we intend to keep the innovation rate of multimedia peripherals and sharing and backing up our personal creations online.

1.2 Goals

The goal of this work is to develop a decentralized architecture for secure file storage. The solution will have the ability to easily share files among groups, a feature that popularized cloud file storage. Also, all files managed by the solution should be unintelligible as soon as they leave a users' machine. The distributed setting of the architecture will rely on a peer-to-peer topology in order to have built-in replication as well as improve the system's scalability.

1.3 Document Organization

This thesis is organized as follows:

- Chapter 2 - Related Work - Presents a study of the state of the art for Peer-to-peer (P2P) networks. It also provides an overview on different optimization techniques useful to the final solution. In a cryptography section, it explores secure storage solutions and key establishment protocols. Finally, it compares existing storage systems and discusses their limitations.
- Chapter 3 - Solution - Describes a proposal of an architecture for the Cloudbox system. It explains the details for every algorithm necessary to have a group membership system featuring an underlying secure storage.
- Chapter 4 - Implementation - Presents the implementation details of the Cloudbox system developed in order to obtain experimental data to support the conclusions of this work.
- Chapter 5 - Evaluation - Consists of an analytic evaluation of the metrics captured.
- Chapter 6 - Conclusions and Future Work - Elaborates some final comments about the work done and the achieved results. Some thoughts about future work are also presented.

Chapter 2

Related Work

This section describes the most relevant research work for the definition of the distributed secure storage solution. First, section 2.1 features a survey of the most relevant peer-to-peer topologies. Next, on section 2.2 are described typical optimization techniques used in distributed storage systems. On section 2.3 are presented encryption methods and key management infrastructures. Finally, in section 2.4 are described and compared the capabilities and shortcomings of existing systems.

2.1 Peer-to-peer Topologies

Peer-to-peer architectures are an alternative to classic centralized architectures, where multiple clients obtain a service by contacting a control server. In turn, peer-to-peer systems distinguish themselves from that approach by transferring more responsibility to the clients. In a peer-to-peer system, every peer runs the same client application and are equivalent in functionality. The peers communicate directly without assistance of a central server but also without its control. Peers organize themselves in an overlay that constitutes the system network.

These architectures result in systems where the peers are able to share resource like storage, CPU cycles or bandwidth and feature characteristics such as self-organisation, fault-tolerance, scalability and no central mediation that are desired for most distributed applications.

There are three key problems addressed by the proposed peer-to-peer systems [1]. They are not common to all peer-to-peer based applications, but all are fundamental challenges for applications to be built on top of peer-to-peer networks.

- **Routing and location** This challenge is common to all peer-to-peer systems: routing messages to a given peer and translating its address, required for the direct communication. Attempts to solve this problem efficiently have spawned systems that can be grouped in different classes according to their level of centralization and network structure.
- **Anonymity** To protect the identity and physical location of individual peers, some peer-to-peer systems may desire to use special overlays and routing techniques to make the participation in the system anonymous.

- **Reputation Management** Without a central point of management there must be found an alternative mean of classifying ,qualitatively or quantitatively, the reputation of a peer and who keeps that information secure in the system.

Since the routing and location problem is common to all peer-to-peer systems it also determines its classification. The organization of the peers in the network—the overlay—determines its classification. As previously mentioned, the classification occurs according to two characteristics of the overlay: its centralization level and structure [2].

Network Centralisation

Considering that peer-to-peer systems encompass all systems with direct communication between peers, they can be classified in different level of centralisation:

- **Hybrid Decentralised** In such systems, even though the main communication is made strictly between peers, there is a central server that is responsible for helping peers joining the overlay, helping them finding other peers or storing metadata. From a complexity point of view, this is the most efficient approach, however, such a system does not scale and has a single point of failure.

  An example of such a system is the Napster sharing application. Users installed the client application becoming peers in a network. They were able to share files among themselves. When a user made a file available for sharing, its identifier would be registered in the central repository along with the identification of the peer. The system was later shutdown by criminal authorities given that the network was used to illegally share copyrighted material, precisely by taking down the central repository.

- **Decentralised** This is the scenario of a fully distributed system, there is no central management and peers have exactly the same responsibilities. These systems are required to solve the routing and locating problem efficiently.

- **Partially Centralised** This approach is similar to the distributed as there is no central server, however, some peers have more responsibilities than others in managing the system. It tries to mitigate the problem of a single point of failure as there can be more than such a peer and in case of failure they can be replicated by a standard peer.

Network Structure

According to the routing and location technique used, a peer-to-peer system may present either a structured network or an unstructured network.

- **Unstructured** A system with an unstructured overlay does not take advantages of the way the overlay is constructed to find peers or content in the system. Instead, there are search mechanism in place to perform the location. these mechanism can be simple, such as flooding the network

until obtaining a reply, or more elaborated as random search paths using, for instance, a depth-first algorithm.

- **Structured** Structured systems address the location problem in unstructured systems by forcing a structure in the way the overlay is constructed and then take advantages of that fact to efficiently locate peers.

Unstructured networks are more performant for systems that have to handle frequent connection and disconnection of peers, given that they do not have the same overhead that structured network systems have maintaining updated information of the overlay. Conversely, structured network provide more performant routing and locating operations.

In table 2.1 the systems described in this report are classified according to the previous categorizations.

		Centralisation		
		Hybrid	Partial	None
Network	Unstructured	BitTorrent Napster	KaZaa	Gnutella
	Structured			Chord CAN Tapestry Pastry Viceroy

Table 2.1: Classification of the systems mentioned in this report

Structured Systems

When building a distributed system, some peer-to-peer topologies allow to build a fully decentralised architecture. Scalability is an obvious advantage of this systems, but there are different configurations of the peers, the topology, and each one has different advantages and disadvantages, that balance performance, anonymity and fault-tolerance.

Chord As a structured peer-to-peer system, Chord [3] focus on constructing and maintaining an overlay. It was designed to efficiently perform the lookup operation: given a key, it maps the key onto a peer. A consistent hashing function is used to assign a set of keys to each peer, in a way that the number of keys is evenly balanced between peers, with each peer storing roughly the same number of keys.

The previously mentioned consistent hashing function, assigns each peer and key a m-bit identifier using an hash function. The peer identifier is the result of hashing the peer's IP address and a key identifier is the result of hashing the data item—usually its name. m should be large enough so that the probability of two peers or keys being assigned the same identifier is not significant. The identifiers are ordered in an identifier circle of modulo 2^m.

The successor of a peer p—$successor(p)$—is defined as the first peer clockwise from p, given that the identifier circle allows to represent the peers in the system in a circular arrangement.

A given key, say k, will be assigned to the first peer whose identifier is equal or follows k in the identifier space. The consistent hash algorithm is designed to be compatible with the transient nature of the peers: to maintaining the mapping when a peer p joins the system, some of the keys currently assigned to its successor are reassigned to p. For the converse, when a peer leaves the system, all its keys are reassigned to its successor.

To accelerate the lookup operation, each Chord peer maintains a routing table, called a finger table, that maps the identifier to the IP address of the consecutive power of two successor peers—$successor(p + 2^{i-1})$, where p is the identifier of the current peer and i the i^{th} finger of p. This additional information allows the lookup operation to resolve a query with $O(\log n)$ messages.

The finger table is only meant to speed up the lookup operation, for that reason, if a peer joins or leaves the system and a lookup query is issued before the finger tables are updated, Chord ensures the correctness of the lookup algorithm by forcing an update of the finger tables when a peer joins and this guarantees that the address of the correct peer is found, although slower. However, refreshing the finger tables of all peers in the system when a peer joins can become inefficient for a large number of peers, for that reason, Chord features an alternative stabilisation protocol [4]: each peer p, periodically contacts its successor and asks for its predecessor; if the reply is not p, then p's finger table is outdated and is updated to set $successor(p)$ the received reply. With this protocol, if a lookup query occurs during the stabilisation protocol, there are three possible results for the query:

- If none of the peers involved in the lookup is affected by the joining of the peer, the lookup returns the correct result in $O(\log n)$ messages.
- If a peer, in the affected region that is used for the lookup, has the up-to-date successor but outdated fingers, the lookup will also yield successfully, although slower.
- If a peer, in the affected region that is used for the lookup, has an incorrect successor or still has not finished migrating all keys to the new peer, the lookup will fail.

In case of intermittent failures or network partitioning, the overlay may reconfigure and partition itself. If the ring-like overlay is ever partitioned into multiple rings, the stabilisation protocol will be unable to heal the overlay into a single ring.

In a nutshell, Chord goals are: load balancing, by consistent hashing; decentralisation, as there is no central repository; scalability, by guaranteeing the efficiency of the lookup operation; and flexible naming, by using a hash function to generate the identifiers.

CAN CAN—Content Addressable Network [5]—was developed parallel to Chord and other structured systems like Tapestry and Pastry, so it aims to solve the same goals: scalability, efficiency in locating peers, dynamic and balanced.

CAN organises the overlay by assigning each peer a zone. the overlay itself is represented as a d-dimensional Cartesian coordinate space, where each zone corresponds to a segment of the coordinate space. Each zone is identified by the boundaries of d points. As such, with a key, say k, being mapped

to a point p in the coordinate space, the lookup function will assign k to the peer whose zone contains the point p.

Each peer maintains a routing table of all its neighbours in the coordinate space. Two peers are considered neighbours if they share a $(d-1)$-dimensional hyper-plane. The rationale behind the lookup operation is to forward the query message through a path that approximates to that of a straight line in the d-dimensional space. Every time a peer receives a lookup request, it forwards it to the neighbour closest to the peer storing the key. The peers require to keep a routing table of size $O(d)$ for the location of its neighbours. Seeking the straightest path, from the source peer to the destination peer is achieved by at most $O(dN^{1/d})$ messages.

When a peer joins the network, it chooses a random point p within the coordinate space and performs the lookup for p. The peer with the zone containing p will partition its zone in two halves and assign one to the new peer, which should then announce itself so that the remaining neighbours can update their routing tables. When a peer gracefully leaves the network, it chooses a neighbour to take the responsibility of its zone. The neighbour will to merge its zone and the newly assigned zone into a larger unified zone; if it fails due to the zone not being valid, it will then be responsible for maintaining the two separate zones.

CAN also has a stabilisation protocol that is periodically run in background. The object is to find zones that can be merged into larger valid zones and reassigned. For that, each peer exchanges with its neighbours its zone coordinates, its list of neighbours and their zone coordinates in order to detect possible zone merges or silent disconnections.

Tapestry Tapestry [6], like the previously described systems, aims at producing a self-organising network topology with efficient routing. Tapestry routing algorithm is designed to take network locality in account. Contrary to the previous structured systems, it does not travel outside the local area unless its necessary to do so.

The Tapestry overlay is based on a Plaxton Mesh [7], a distributed data structure that allows peers to locate objects and route messages to them in an arbitrarily-sized overlay network. Each peer keeps a map of its neighbours where every entry is a pointer to the closest peer in the network whose identifier matches the identifier on the neighbour map. An identifier is l-digits long and every digit corresponds to a level—a unit of distance in the overlay.

The lookup operation is made by resolving a level of the identifier at each peer and redirecting for the next level, starting from right-to-left.

The Plaxton Mesh was originally designed for a static network, so Tapestry enhances it to support the transient nature of a peer-to-peer network. Apart from the neighbours map, each peer also keeps a back-pointer to every peer where it is referenced as a neighbour. When a peer joins or leaves the overlay these back-pointers are used to regenerate the neighbours maps.

Pastry Pastry [8] shares the same goal as Tapestry: to provide efficient routing with a notion of network locality.

Pastry routing is a tree-based algorithm [4]. Each peer p chooses randomly an identifier indicating its position in the identifier circle. It also keeps a leaf set—L—comprised of the $|L|/2$ set of closest and identifiers smaller than p, and a set with the $|L|/2$ peers that are closest and with identifiers larger than p.

The lookup operation checks the L leaf set for the target peer using the given key. If it is present then is a matter of jumping to the mapped peer, otherwise, it checks a routing table for a known peer that has a longer shared identifier prefix with the sought key.

The join operation for a Pastry overlay is designed so that the new peer is able to build its leaf set and routing table as he is looking for the nearest peer to the identifier it chose. When a peer leaves the system, only the leaf sets are immediately updated, the routing tables are only corrected later, as they are not required for the correctness of the lookup operation. This approach allows a Pastry system to find the correct peer in $O(\log n)$.

Viceroy Viceroy [9] is an alternative peer-to-peer system that was designed for maximum performance at scale. It was designed so that the cost of the lookup operation is evenly distributed among the participating peers; for the join and leave operations to change as little state in the least number of peers as possible; and for the lookup operation itself to require the least amount of messages.

The overlay is organised as a double-linked ring where each peer has a link to its successor, its predecessor and five additional long-range links to other distant peers. The overlay is also structured in $\log P$ levels, where P is the total number of peers, given that each peer selects a random level at the join operation. The long-range links are used to jump to a different level and skip ahead and backwards in the ring. With this overlay the lookup routing id achieved with $O(\log n)$ hops. The join ans leave events generate at most $O(\log n)$ messages and require $O(1)$ peers to change state.

Unstructured Systems

The following topologies present different search mechanisms to compensate the fact that they do not impose rules to the structure of the network. In order to guarantee performant solutions some approaches sacrifice the level of decentralisation.

Gnutella Gnutella [10] is a fully decentralised peer-to-peer system, though it also has been extended to support hybrid and fully centralised approaches as well.

On the decentralised version, there is no network organisation, when a peer join the network it announces itself and waits for its neighbours to reply with their address and shared objects. It is up to the new peer to choose to which replying neighbours to connect. After that, the peer will periodically ping its neighbours to obtain their neighbours and expand its map of the network.

For the lookup operation, a peer broadcasts a request for the object that it is trying to find. The request is tagged with a maximum range that the message should propagate, similar to the Time-To-Live field in the IP protocol. When a result is found the response is back-propagated: it is sent the opposite

path through every peer it took to get to the destination peer. To prevent flooding the network with consecutive requests and loops in the lookup operation, each peer maintains a cache with the response of all recent requests, which is the reason every message has a unique random identifier, as there is no guarantee a peer will not receive a the same message more than one time, given that the overlay is not structured.

KaZaa KaZaa [1] is an example of a partially centralised system, as it introduces the notion of super-peers to facilitate the lookup operation. Super-peers are peers that usually have better resources than the remaining peers—higher bandwidth and more processing power—that are elected to participate in the routing and locating protocol.

For the lookup operation, a peer sends a query to a super-peer, which then broadcasts the request to the remaining super-peers. When a peer desires to start sharing an object it must announce it to a super-peer. The super-peer will store a mapping of peer-object of only those that were communicated.

Super-peers are essentially proxies for the communication in order to reduce the required bandwidth. Some messages, like the one a peers uses to find a super-peer will still use a brute-force approach of flooding. As a result, KaZaa benefits from the best of both approaches: fast routing and location without a single point of failure; and low overhead given that the network is unstructured for most peers, only super-peers will pay in performance for the keeping state of the network.

BitTorrent Similarly to Napster, the BitTorrent [11] is an hybrid system that uses a centralised server to solve the location problem.

The system was designed for file-sharing, and while the discovery of peers that stores given file is efficient, it has poor scalability and is vulnerable to attacks, surveillance and censorship given the single point of failure.

FreeNet FreeNet [1] is a fully decentralised loosely structured system. It does not classify as a structured nor unstructured network, because it can only produce an estimate of where the content will be located. It is designed to provide the sharing peer anonymity that is not possible in the other structured networks.

Each stored object is represented by a key—a hash of the object's identifier (it also uses other mechanism depending on the level of security required). To insert a key, the routing algorithm finds at each peer the peer with similar stored keys. This results in peers storing a set of objects with similar keys.

For the lookup, it uses a chained propagation approach: every peer makes a decision of where ti route the request next using a depth-first algorithm that backtracks when reaches a dead end or finds the result. While the responses are backtracking the peers involved in the chain update their routing tables to include new peers and objects or clean the ones that are no longer available. A successful response, instead of containing the location of the peer like in the previously discussed systems, it returns the data itself, protecting the identity of the publisher. While the data is being chained back to the peer that

initiated the lookup it is cached in every peer along the way, in a Least Recently Used—LRU—fashion cache.

The fact that the requests are chained and the requesting peer does not know the identity of who is sharing the content hides the underlying topology of the network. It does however allow a malicious peer to perform a man-in-the-middle attack by corrupting cached object—this problem is common to all structured networks. To prevent this attack, the Mnemosyne [12] system extends the FreeNet overlay to use steganographic storage of the object, preventing them from being corrupted.

Discussion

Having seen various peer-to-peer solutions, it is interesting to see how they compare against each other and what better fits the objectives of the solution proposed in this report.

	Routing performance	Routing state	Peer join/leave
Chord	$O(\log N)$	$\log N$	$\log^2 N$
CAN	$O(dN^{1/d})$	$2d$	$2d$
Tapestry	$O(\log N)$	$\log N$	$\log N$
Pastry	$O(\log N)$	$\log N$	$\log N$
Viceroy	$O(\log N)$	$O(N)$	$O(\log N)$
FreeNet	Guaranteed	Constant	Constant
Unstructured	Not Guaranteed	Constant	Constant

Table 2.2: Comparisson of the main operations in the described systems

In terms if complexity, the systems are very similar, the only major difference being between structured and unstructured systems, as seen in table 2.2. Since the systems with a structured topology offer greater performance on the retrieval of objects using scalable routing and search algorithms, they are more appropriate for the intended solution, as it is expected that the lookup operation will be a frequent one. As previously discussed, the main advantage of choosing a structured is the effect of three join an leave operations, however they are an acceptable compromise, as the converse, a non scalable search algorithm in a unstructured overlay, is not an acceptable trade-off, given scalability being a key attribute of the desired solution. Furthermore, some unstructured systems do not even guarantee that a lookup query is successful.

Considering just the structured network approach, choosing one becomes harder, as complexity is no longer a criteria, since they all perform so similarly. The decision must then be based in the unique features each topology offers. Chord offers a simple and efficient overlay; CAN is the one with worst scalability of the bunch; Tapestry, Pastry and Viceroy all are inspired by Chord and CAN but introduce the concept of network locality that is useful for a application of global scale, without an impact on performance. Of those three Pastry exhibits the simplest routing algorithm.

The conclusion that a structured overlay with a notion of network location is appropriate for file storage is supported by systems like Mnemosyne and OceanStore, that use Pastry and Tapestry respectively, as the foundation of their work.

2.2 Optimisations

Geo-replication

Geo-replication refers to the scalability model realized through the replication of data across several geographic locations, which aims at improving availability, while also reducing network latency by making content available closer to the clients.

As well as any distributed system, a geo-replicated system's design is influenced by the CAP theorem [13], according to which is impossible to offer simultaneously consistency, availability, and partition-tolerance. The advantage of increased availability is paid in the increase of complexity to maintain a certain level of consistency. Replication can be implemented using different techniques: active replication, passive replication, quorums and chain replication.

Apart from availability advantages, geo-replication is also a better scaling model than a vertical one. The infrastructure costs of such a system can be lower [14].

Data Deduplication

Data deduplication [15] is a technique for automatically eliminating coarse-grained and unrelated duplicate data. It is intended to eliminate both intrafile and interfile duplication over large datasets, even if the files are changed at different times or even in a distributed setting. It aims to reduce the occupied memory—primary or secondary—by identifying duplicated chunks of data. Data deduplication yields saving in data storage by reducing the amount of used space, but also favourable impacts network bandwidth in a distributed architecture.

Since deduplication relies on the notion of identical data it might seam that it is incompatible with secure storage, due to the fact that ciphers attempt to make the data appear as random. Three different approaches [16] can be taken:

- Deduplication can be applied directly to cyphered chunks. Though, much less space would be saved since different ciphers produce different ciphertext strings.
- As an improvement to the problem of the previous approach, the same key could be used to content that is identified as identical. However, this introduces an additional key sharing problem.
- The third approach is to generate encryption keys in a consistent manner from the chunk data, in a way that identical chunks will always encrypt to the same ciphertext. This approach is called convergent encryption[17] and it has the disadvantage of revealing if two ciphertext strings decrypt to the same plaintext value.

Erasure Coding

Erasure coding is a redundancy scheme and alternative technique to raw replication [18]. It achieves redundancy by dividing the input in m fragments and then transforms these into a greater number of

fragments (say n) [19]. The n fragments are the base mapping to reconstruct the input object, while the m fragments contains redundant data from two or more n fragments. The transformation applied to obtain the n fragments controls the redundancy factor obtained: $k_c = n/m$.

Coding introduces complexity, by requiring a complex system design to spread and retrieve the fragments, and by the encoding/decoding algorithms. It can also have a negative impact on latency when compared to other replication techniques, since fragments are fetched from several different locations, as opposed to a single (and possible the closest) location. Still, it provides better availability than full replication [20].

Delta Encoding

Delta encoding—also known as delta compressio [21]—is a technique that can be used for efficiently transfer files over slow communication links where the receiving party already has a similar file.

In systems where both communicating parties share the same file and one wishes to synchronise a modification to the other, instead of sending the entire file, it is more efficient, especially for multiple receiving parties, for the sender to generate a "patch" *a priory*, i.e., calculate the differences between the original file and the updated version and send that difference. It is up to the receivers to then reconstruct the new file by applying the patch to the original file. The process of generation of the patch and reconstructing the file imply that both the sender and receiver agree on the same delta encoding algorithm.

Delta encoding is also used by version control systems to achieve efficient storage. In order to maintain all versions of a file available, version control systems store a baseline version of a file and record the consecutive modification to the file as deltas. To retrieve a file at any given version, these systems recover the base file and apply all the deltas up to the desired version.

There are different classes of delta encoding algorithms, each of which uses a different string matching technique to compute the delta itself. They are the insert/delete and the copy/insert classes [22].

- The insert/delete class of algorithms uses a string matching technique that finds the longest common sub sequence between the two files and then considers the missing parts of the sequence as a delete instruction and the new parts as insert instructions.

- The copy/insert class of algorithms uses a string matching technique to record the range of matching regions—unmodified regions, common to both files—and computes the offsets where the inserts belong. A copy instruction consists of a matching region, that corresponds to a common region and records the starting and ending positions of that region. An insert instruction contains the new text and the offset relative to a matching region where they must be inserted.

For example, consider two files, f_{old} with content: "my file"; and f_{new} with content "your file". On one hand, the insert/delete approach, with this example would generate a patch with 6 instructions total: 2 delete instructions—one for m and another y—and 4 insert instructions. The copy/insert approach, on

the other hand, would generate for this example a total of only two instructions, an insert instruction—insert $your\_$ and a copy instruction—copy $file$.

From the previous example, it is observable that the insert/delete class gives an insert the same significance as it does to a delete instruction. This makes it more verbose than the copy/insert class algorithms which usually produce less instructions per modification, resulting in smaller patches. For this reason the insert/delete approach is typically less appropriate for storage, network or other automated purposes, although being more appropriate for text-based files and human-readable formats than binary files.

2.3 Cryptography

Secure Storage

The challenges assumed by secure storage go beyond encrypting data. Secure storage of data [23] is established on three concepts: confidentiality, integrity and availability.

- **Confidentiality** Keeping data confidential means that users can access it only if they are authorised for such an operation. Confidentiality can be achieved by storing the data encrypted or by having an authorisation scheme that controls the permissions for reading data.
- **Integrity** Keeping the integrity of stored data requires to ensure that no unauthorised user can modify the data. This is usually achieved by storing the data with a MAC—Message authentication code—or by digitally signing the data.
- **Availability** Availability is the concept of keeping data ready for users' requests when needed. This concept was already discussed in the previous section.

Besides these goals there are often relevant decisions to explore in a secure storage system: how are the keys stored; what encryption mechanism is used; are they bound to a particular size of keys; how is metadata stored; and also are there any protocols to ensure data deletion. These factors influence the security evaluation of secure storage system.

Secure storage system can be implemented with specific hardware or using software-based approaches [24]. The software-based approaches, although cannot achieve the performance of dedicated hardware are interesting as they are much more flexible. Software-based secure storage implementations can be classified according to the level where the cryptographic engine executes: application level, file system level, or block level.

Block-level Block-level secure storage systems operate at the lowest level, below the file system layer, encrypting at the granularity of a disk block, hence its name. They are completely decoupled from the file system and, for that reason, are able to secure the file system metadata: filenames, file sizes, directory structures, and user permissions. Such systems present better performance when compared with file system solutions but are unable to provide the same flexibility as other secure storage solutions given that they encrypt entire partitions at once.

File system level At the file system level there are multiple ways to implement secure storage: disk-based, network loopback, or a stackable approach.

In a disk-based secure storage the cryptographic module is an extension to the file system. A disk-based approach as access to all of the file system metadata: files, directories and block-location. It can use that information to provide fine-grained authorisation schemes, for example, at a level of a key per file, as opposed to the block-level approach that required a key to encrypt an entire partition. As far as performance is concerned, all the encryption and decryption operations are performed in kernel-space. This approach, as other file system-based solutions, does not offer confidentiality for the metadata.

Network loopback-based systems leverage the Network File System available in some file system implementations. The Network File System is a module that redirects local file systems to a remote server used to create distributed file systems. Cryptographic file system solutions use this mechanism to redirect the file system primitives, from kernel-space, to a daemon, running in user-space, that consists of a Network File System server and an encryption engine. Implementations using this approach achieve a certain level of decoupling from the underlying file system—they are required to support a network loopback protocol and implement an interface for every file system. However, such implementations cannot control the representation of the files on disk, and have to deal with possible vulnerabilities of the network protocol.

Stackable storage systems try to combine the advantages of both previous approaches. They consist of a virtualized file system working on top of the native file system that intercepts the operations and processes them in an encryption engine in the kernel-space. This approach achieves the decoupling of network loopback implementations whilst not paying the overhead of transitioning between kernel and user-space.

Application level When secure storage is implemented at application level, as a component of an application, the choice of cipher algorithm is open for the developer, as is the decision of what to encrypt. However, there are a few drawbacks: the storage the keys must be dealt by the developer; the metadata of the encrypted files is not protected; and, the process of encrypting and decrypting might temporarily reveal the data in temporary files that are possibly visible to attackers.

In application level systems another influential decision to the confidentiality of the system is the way the encryption is employed. On a distributed secure storage system, where there is a need for the data to leave the physical machines of the user, two approaches can be taken:

- **Server-side encryption** In this scheme, the data is only encrypted at the remote location. A secure channel is established between the user machine and the remote storage provider before data is uploaded. That way, a malicious third-party cannot eavesdrop on the transfer. However, the entity responsible for the remote storage has access to all the data. Measures like using an encrypted file systems reduce the risk but the data will still be visible to the remote third-party and out of control of the user.

- **Client-side encryption** This approach is inherently more secure as all data is encrypted locally on the user machine before it is uploaded. Whether it is done using symmetric or asymmetric

encryption, no one other than the client is able to view the plain text content of the data, since the key is keep exclusively on the client.

Keys For secure storage systems the cipher algorithm is an important component that confers the confidentiality to the system. The encryption engines are usual modular so that the systems can be tailored by changing the used cipher and key size, that affect significantly the performance of the system, depending on the data that is being stored. For file storage the most appropriate ciphers are symmetric block cipher [25]. When deciding between asymmetric and symmetric, symmetric ciphers are a better choice as they have a lower computation cost. The reason to use a block as the operating mode is that it is very flexible considering the size of the input. Block ciphers allow us to encrypt the input as a block of the same size as the size of the key or, with help from an initialization vector, transform the cipher into a self-synchronising stream cipher, that generates a continuous key stream that can be used to encrypt inputs of arbitrary length.

The most popular ciphers for a file encryption system are the DES, Blowfish, and AES cipher [24].

Key Establishment Protocols

Key establishment protocols refer to the technique used to establish a shared secret between two or more parties. Key establishment protocols can be classified in two categories [26]: key distribution and key agreement protocols.

- **Key Distribution** This approach implies a centralised or hierarchical architecture where one of the involved parties creates a secret key and then securely distributed it to the remaining parties.
- **Key Agreement** The algorithms in this category allow the individual parties to negotiate a secret key collaboratively, even over insecure public networks.

A key distribution approach has several disadvantages. The fact of being a centralised architecture makes it a single point of failure and a performance bottleneck. It is also very attractive for an attacker given that all the system's secrets are kept in one location. It is also very difficult to achieve perfect forward secrecy and resistance to known-key attacks (these are detailed further ahead in this section) with such protocols.

For a key agreement protocol to be effective there are a number of required properties that guarantee its security:

- **Key Authentication** Authentication of the parties involved in the exchanges guarantees that no main-in-the-middle attacks—communication interception followed by impersonation—are possible. The authentication can be partial if only a party is authenticated, or mutual if all parties identity is verified by all the other.

  This problem is usually solved by the parties joining the system with authentication keys or by providing a public-key infrastructure with certificate authorities that distribute public-key certificates.

Step	Alice	Bob
1	p,g	
2	$A = random()$ $a = g^B \bmod p$	$B = random()$ $b = g^A \bmod p$
3	Send a	Send b
4	$K = g^{BA} \bmod p \equiv b^A \bmod p$	$K = g^{AB} \bmod p \equiv a^B \bmod p$
5	Send $E_k(data)$	

Table 2.3: Technologies employed by the described solutions

- **Perfect Forward Secrecy** Forward secrecy, also called perfect forward secrecy [27], is a property that establishes stronger confidentiality of a key-agreement protocol. An authenticated key exchange protocol featuring forward secrecy guarantees that if a long-term secret is disclosed, then all previously generated session keys from that secret will not be compromised.

  Partial forward secrecy refers to when only a participant on the key exchange protocol verifies these properties. If all participants verify them then the algorithm has forward secrecy.

- **Resistance to Known-key Attacks** A protocol is resistant to a known-key attack if it guarantees that in case a session key is disclosed, it cannot be used to compromise other sessions, that use different session keys, or long-term secrets.

- **Key Confirmation and Key Integrity** Key confirmation refers to a later step of the exchange algorithm, where the parties validate that they all possess the same generated key.

  Key integrity refers to the fact that a malicious party cannot influence the generation of the key. Even if it intercepts the communication of the exchange protocol it should not be able to derive any information that facilitates an attack.

2-Party Key Agreement Protocols The Diffie-Hellman protocol [27] is a key agreement where the two parties contribute to the generation of the key. In it, each party selects a random secret number, A and B, and calculates $a = g^A \bmod p$ and $b = g^B \bmod p$, respectively, given p and g are large prime numbers generated and publicly shared by one of the parties *a priori*. Each party can then derive the key by computing $K = b^A \bmod p$ or $K = a^B \bmod p$ to establish encrypted communication as seen in table 2.3.

The security of the Diffie-Hellman protocol is supported by the Decisional Diffie-Hellman Problem, which consists in determining whether $g^c = g^{ab}$, given g^a, g^b, and a random c, that is computationally hard for very large prime numbers.

The first proposed version of the Diffie-Hellman protocol was vulnerable to man-in-the-middle attacks as it provides no authentication mechanism. Proposed solution for this problem include the Authenticated Diffie-Hellman protocol, which uses a public-key infrastructure to verify the identity of the parties. There are other alternative approaches, know as password-authenticated agreement protocols, that use a initially shared secret to achieve the authentication. An example is the Diffie-Hellman Encrypted Key Exchange protocol [28], which in the transfer step—step 3 of table 2.3—is modified so that the transmission of a and b are encrypted by this shared secret. These solutions have some disadvantages: they are

not contributing—each party does not contribute equally to the generation of the key; they rely on a third party; or, in the case of Encrypted Key Exchange protocol, does not provide perfect forward secrecy.

The original authors of the Diffie-Hellman protocol extended it to support authentication while maintaining its characteristics: contributing and perfect forward secrecy. The protocol is also called Station-To-Station [27], and assumes that the two parties each possess an asymmetric key pair and a public key certificate. The step of the original Diffie-Hellman in which the parties exchange the calculated exponent values is extended to include each party signature and public key certificate. Thus, even if a malicious party is able to intercept the messages, it cannot forge the messages as it cannot forge the signatures.

Group Key Agreement Protocols An interesting characteristic of the Diffie-Hellman protocol is that besides featuring all the desired properties for a key agreement protocol, it is also extendable to n-party negotiation protocols. The Group Diffie-Hellman family consists of three protocols [29]: GDH.1, GDH.2 and GDH.3. They share the same idea, but GDH.2 focus on minimising the number of messages sent during the protocol and GDH.3 tries to minimise the computation costs.

The idea behind these algorithms consists of having two stages: In the first stage the individual contribution of each party is collected, in a chained formation, where each party sends its contribution to the next, and the final one broadcasts the computed keyring material. These protocols, like the original Diffie-Hellman protocol do not support authentication, but they can be extended to authenticate the involved parties, as described in the SA-GDH.2 protocol [26].

However, all previous algorithms consider a group as a static entity, that means that are unable to cope with members joining or leaving a group. A possible solution could be to simply delete all information and start the protocol from scratch for the new members. Such an approach is computationally expensive an unscalable for environments where groups are subjected to frequent modifications. Another, more efficient solution, is to resume the information from the establish group. The A-GDH.2-MA protocol [26] achieves just that. It is an authenticated member addition version of GDH.2. The members cache the partial key from the initial run of the protocol, and for every modification of the group, they incrementally update the key.

The STR protocol [30] for group key agreement is an interesting solution as it was designed for groups with dynamic membership. Instead of the previous protocols that communicated in a chained approach, the STR is built on hierarchic tree data structure, which means that the required number of messages in the protocol is constant, and not bound to the size of the group. It also expands the notion of groups by introducing subgroups that can be partitioned and merged. The disadvantage of this protocol is that it has fairly high computational cost, which is bound to the number of members in the group.

2.4 Existing Storage Systems

Dropbox is the most popular cloud storage service. It distinguishes itself from its competitors by offering easy to use sharing features.

Dropbox

The Dropbox architecture is built on top of Amazon AWS[1]. Its service is powered by two different components[31]. On the one hand, *control* servers are responsible for managing files meta-data, Application Programming Interface (API) requests, and the web interface. On the other hand, *data storage* instances, relying directly on Amazon S3[2], receive and serve request for files upload and download. Data upload and download is over the HTTPS protocol. Dropbox, Inc claims that all user content is encrypted server-side using AES algorithm with 256 bits keys [32]. However, the authentication mechanism is a simple username and password which if broken allows full access to the unencrypted files.

Every file in the Dropbox folder in split into 4MB chunks. Each of these is assigned an identifier obtained by SHA256 hash. It is not known if any technique of deduplication is applied at this stage.

Dropbox allows to view previous versions of each file. This feature results of an optimisation used to transfer the file chunks, by using delta encoding when updating existing chunks, since the Dropbox client only uploads and downloads the differences made to the file between the current version and the latest version.

Other solutions

There are a number of commercial cloud services competing with Dropbox offering client-side encryption and peer-to-peer storage. The most proeminent are BitTorrent Sync, Cubby, and SpaceMonkey.

Cubby [33] offers client-side encrypted storage on a cloud architecture. A symmetric key is used to cipher the content. This generated on Cubby's servers when an account is created and kept on a database ciphered by an asymmetric key which in turn is ciphered by a transformation of the user's password. When a user installs the software client the folders and the asymmetric key are downloaded.

BitTorrent Sync [34] leverages the technology of their existing peer-to-peer torrent client BitTorrent for the transfer of the files and allows for client-side encrypted storage. Despite its foundation on BitTorrent, instead of disseminating the files through the peer-to-peer network, it will only store files on trusted devices—the users—which must be explicitly added. The Sync infrastructure consists of a peer-to-peer tracker which maps folders—each shared folder is identified by a unique hash which does not depend on the content—to users.

SpaceMonkey approach is similar to BitTorrent Sync, but instead of relying on an open peer-to-peer network, they sell dedicated hardware—a network-enabled HDD—where the files are stored. Similarly to Cubby, since they offer a web interface for their service, they must keep the content ciphering key on their server. Their solution also does not employ any deduplication technique [35].

There are countless more services for online storage, but they do not differ in the technologies used from those already mentioned. Table 2.4 briefly compares the key technologies that these services are known to use. We say that a key is lent if the secret key is kept by the provider of the service, even if ciphered, and is, at some point, transferred from or to the client machine.

[1] Amazon Web Services
[2] Simple Storage Service

	Paradigm	Encryption	Encryption algorithm	Key size	Lends key
BitTorrent Sync	P2P	Client-side	AES	128 bits	No
Cubby	Cloud + P2P	Client-side	AES	256 bits	Yes
Dropbox	Cloud	Server-side	AES	256 bits	Yes
SpaceMonkey	P2P	Client-side	AES	N/A	Yes

Table 2.4: Technologies employed by the described solutions

Chapter 3

Solution

The systems presented in the previous section have problems that prevent them from being true secure storage services. The following list addresses those problems and proposes how they can be solved in the scope of this project.

- **Cloud-based infrastructure** Most storage providers presented previously rely on cloud infrastructure to build their services. A better approach is to use a peer-to-peer network to host the service. On the one hand, not only will the infrastructure scale by itself as the number of users grows, as it is also much more cost-effective to maintain for the service provider. On the other hand, nodes join and leave the network in an uncontrolled way, and for that reason, other techniques like replication are required to prevent data loss.

- **Server-side encryption** Services that use server-side encryption can not be labelled as secure. Privacy is an important requirement of a system that claims to be secure. Giving the entity responsible for the service plain-text access to the user files can not be considered a secure practice. Thus, the only way to assure that the user content is private is by making sure it is never comprehensible in an unauthorised device. For that reason, client-side encryption must be used.

- **No perfect forward secrecy** The solutions offering client-side encryption require a user to create an account online and then install a client. During the account creating process a key to cipher the files' content is generated. When the user installs the client, that key is transferred from the cloud servers to the user local device. This violates the principle of forward secrecy by potentially exposing a long-term secret, the most important of all in those systems. To prevent these secrets from being exposed, the solution must avoid the transfer of keys, mostly the long-term ones, recurring to key agreement protocols for generation of shared keys.

- **No data-deduplication** As concluded in section 2.4, systems using client-side encryption do not use data-deduplication techniques among different users files. It is an inherit limitation of these systems since once a file is handed to the system it is already ciphered, so, even if two files have the same content, as they were encrypted with different keys, from the point of view of the system they will be different files. It is possible to apply deduplication techniques across a single user files,

even if it is not as effective. Applying deduplication to the files of the all users would yield much more duplicates, hence, saving much more space.

The solution presented next aims at having a system that provides a durable secure storage. The properties above are the main objectives of the system and therefore have driven the main decisions of the architecture.

3.1 Proposed Solution

Like previously mentioned, it is intended for the solution to feature: encrypted and replicated storage, and the ability to create and manage groups that share files among them. It will allow a user to backup his files and share them with others. There are two use cases for the solution: first, a single user using the system for backup and multiple device synchronisation; second, a group of users sharing a set of files between them, and with the ability to add and remove users from the group, having access to the files only those who belong to the group at a given moment. The solution can be described as three layers where each one sets the foundation for the operations defined in the layer above. These layers are File operations, Group operations and Orchestration. Starting from the bottom one, this section exhibits the algorithms and decisions used in each layer.

3.1.1 File-level Operations

Considering in the first place the use case of a single user files synchronization, there are already requirements to shape the basic operations concerning files. These are the create file, update file and delete file operations. They map directly to the actions applied by a user to the files in the file system and thus, can be interpreted as events that require handling in order to sustain state consistency between the solution's representation and the file system. There is another fundamental operation, though a higher-level one, the consolidate operation.

Before detailing the intent of each of these operations it is necessary to explain the concepts and techniques incorporated in them.

Epoch

Given the delta encoding system, as described in Section 2.2, the solution will be able to achieve efficient updates. Once a file is added to a shared folder a copy will be created as the first version of the file. Consecutive updates to the original file cause the system to calculate the difference and record it as the following version. In order for a user to retrieve a file at its latest version, the system needs to fetch the N versions of the original file. This approach is better than uploading the entire file when it is updated due to the small overhead, but also because it is easier to restrict access to only certain versions of a given file, an important concept for the later discussed group membership concept.

A sequence of versions, from version 1 to version N of a file is dubbed as an *epoch*. Nevertheless, this does not mean that a file has a single epoch. Any file can have many epochs. A given file can start

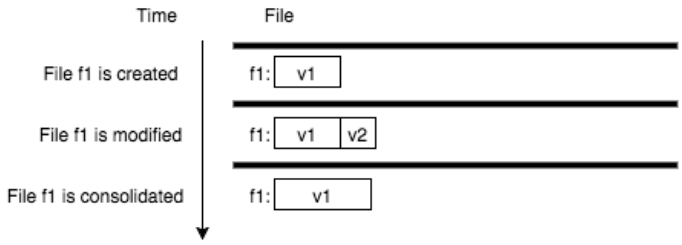


Figure 3.1: File versioning evolution

a new epoch at any moment, meaning that all it's current versions will be merged into a new version 1. From now on, the process of reseting the versions and a file contents will be referred as *consolidating*. The existence of this operation is explained by the fact that over time the cost of fetching all versions to obtain a file will overcome the cost of consolidating the file and starting over. Figure 3.1 illustrates a simplified example of the lifecycle of a file versioning.

Criptography

In order to securely store every file, let's assume the existence of a file key mapped to a file for a given epoch. This key will serve two purposes:

On the one hand, it will be used to cipher the content of each partial version that is physically stored, for the duration of a single epoch. Besides being a mechanism that allows for key rotation, it will also be crucial to enforce membership restrictions.

On the other hand, a file key can be used to generate secure identifiers for storing the files. From the Related Work section we conclude that a structured peer-to-peer topology is better suited for the requirements, as we can efficiently discover the node storing a file from a given file identifier. However, this approach poses itself a problem: if the file identifier can be used to discover all the locations where a file is stored, an attacker could use this information to try to attack the system. For that reason, to guarantee the anonymity of the content stored, the file identifier is itself ciphered with the owner's key, in a way that only users that know the file key will know its location. The proposed file identifier for file version v_i is $hash(\{filename + v_i\}_{K_F})$ where K_F is the file key.

Operations

Having established the previous concepts, it is now possible to formalize the core file operations: create, update and delete. The auxiliary operations $VersionIdentifier$ and $KeyIdentifier$ are used to obtain the file versions and their keys identifiers respectively. The $VersionIdentifier$ corresponds to $hash(\{filename+v_i\}_{K_F})$; the $KeyIdentifier$ is defined later in this chapter. The $PutToDHT(key, value)$ operation corresponds to propagating a given value identified by a determined key in the underlying P2P overlay.

Create The create operation inside the system is very similar to its counterpart in the actual file system: simply consider the whole file as the first version.

Algorithm 1 Add file operation

```
1: procedure ADDFILE(F_N)
2:     K_F ← GenerateKey
3:     PutToDHT(KeyIdentifier(K_F), K_F)
4:     CreateVersion(F_N, K_F)
5: procedure CREATEVERSION(content, K_F)
6:     cipheredContent ← Cipher(content, K_F)
7:     PutToDHT(VersionIdentifier(K_F), cipheredContent)
```

Update The update operation involves the most concepts and it is thus the most complicated one. Besides having to create a new version as the AddFile operation, in addition, it has to calculate the differences between the updated file and the current version.

Algorithm 2 Update file operation

```
1: procedure UPDATEFILE(F_N)
2:     oldContent ← Decipher(F_{N-1}, K_F)
3:     newVersionContent ← Diff(oldContent, F_N)
4:     CreateVersion(newVersionContent, K_F)
```

In this operation, the complete content of the file in the previous version can be maintained either as a whole or recreated at any time from the aggregation of all versions, incurring in the overhead of deciphering and merging them. It is a memory/storage versus CPU/time tradeoff that can be implemented according to the expected frequency of updates of the tracked files.

Delete The delete operation causes a file to no longer be tracked by the system. It causes the file to be deleted (i.e. disapear) from the node where the event is triggered. Not propagating the delete event immediately to other nodes reduces greatly the complexity of the metadata and communication protocol. The consistency of the folder is eventually restored after the occurrence of a folder-level consolidate operation (whose trigger will be explained in the group operations). Propagating the event immediately would not prevent any participant with previous access from retrieving the keys and versions of the deleted file, as they are stored in the underlying P2P network from where they are never explicitly deleted.

Consolidate Finally, the consolidate operation, at file level, is a combination of the create and update operations. As previously mentioned, its purpose is to set a new epoch for a file, which implies the generation of a new key. In the scope of file operations, the consolidate operation is useful, as an optimization technique that can be used to prevent the infinite growth of a file versions. It can be defined as presented in algorithm 3.

Figure 3.2 shows the lifecycle of a file versions and keys given the occurrence of a consolidate operation.

Algorithm 3 Consolidate file operation

```
1: procedure CONSOLIDATE(F, K_F)
2:     versions ← ListVersions(F)
3:     decipheredContent ← {}
4:     for version in versions
5:       versionDecipheredContent ← Decipher(version, K_F)
6:       decipheredContent ← Merge(decipheredContent, versionDecipheredContent)
7:     K_{F+1} ← GenerateKey
8:     PutToDHT(KeyIdentifier(K_{F+1}), K_{F+1})
9:     CreateVersion(decipheredContent, K_{F+1})
```

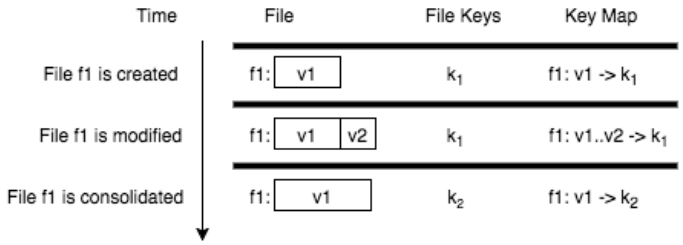


Figure 3.2: File versioning evolution with file keys

3.1.2 Group-level Operations

Having established the basic operations to manipulate files, it is now possible to describe the operations to manage groups. A group can be described as a set of files who are shared between any number of users. The main objective of the group operations is to control the members access to those files, which results in two core operations: the add/join and remove/leave operations. Once again, these are supported by an epoch concept and some cryptography rules.

Epoch

In the context of a group, an epoch is again defined by a key, in this case a group key. The change of key and therefore epoch, is determined by a change in the group members: adding or removing a member to a group terminates the current epoch and starts a new one.

Criptography

The existence of a group key per epoch—K_G—which is shared by all members, is what enables the groups' privacy.

Unlike file keys, group keys are not meant to be used to cipher file contents, but rather, to cipher metadata concerning a group and its files. Given the distributed nature of the underlaying network and non existence of a central repository, the metadata is stored in the system as regular files. In order for them to be private, their identity is derived from the group's key: $FileMetadataIdentifier = hash(\{filename + v_0\}_{K_G})$; $GroupMetadataIdentifier = hash(\{groupname\}_{K_G})$.

Operations

Having separate keys, K_G and K_F, allows to reduce the burden of re-ciphering the files when there are changes to the group membership. There are two possible modes of operation according to the policy selected for group history. The first mode of operation leaks the history of a file to members that join the group later, however, it only stores each version once, as fig. 3.3 shows. The second mode, showcased in fig. 3.4, hides the history of a file for new members at the cost of consolidating the file—merging all versions and re-ciphering with a new key—when a new member joins the group, which means the total size of a file in the system is a function of the file size times the number of member additions to the group. Given that the first mode does not perform consolidation, for a file with many modifications, the fetch operation gets slower for new members as they have to fetch all previous versions to reconstruct the file.

The auxiliary operation $FileMetadata$ and $GroupMetadata$ are used translate a files or groups names and obtain its metadata files. The $KeyIdentifier$ operation can be defined as $hash(\{filename + key\}_{K_G})$.

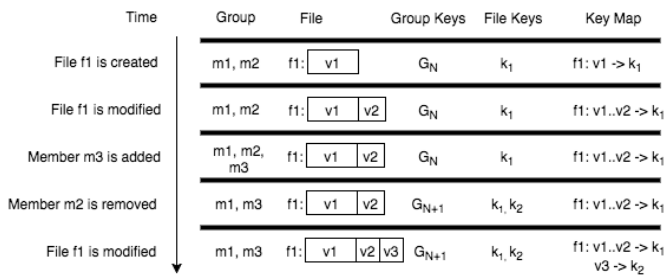


Figure 3.3: File versioning evolution with history

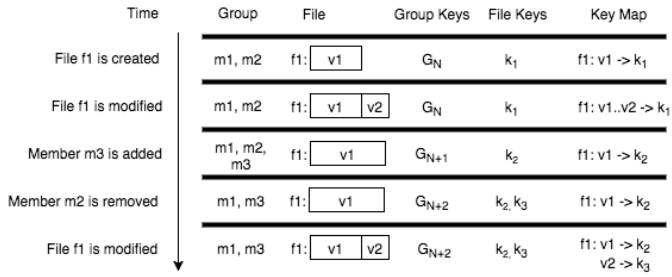


Figure 3.4: File versioning evolution without history

Add User/Join Group Independently of the key exchange/negotiation protocol, the process of extending a groups' membership to include a new member, always requires action from both parties, whether they already belong or are the one joining. Algorithm 4 and algorithm 5 detail the add operations. In the mode that preserves history, no action needs to be taken other than distribute the current group key to the new member. This way the members joins the group without an epoch change, and is allowed access to all the versions of the current epoch. When we do not want to expose any previous group history, we consolidate all the files in the group. From the new member perspective he is joining a new

group, as the consolidate operation reseted all files history.

Algorithm 4 Add user to group operation preserving history

```
1: procedure ADDUSER(G)
2:     Distribute(K_G)
```

Algorithm 5 Add user to group operation without preserving history

```
1: procedure ADDUSER(G)
2:     K_{G_{N+1}} ← GenerateKey
3:     files ← ListFiles(GroupMetadata(G), K_{G_N})
4:     for file in files
5:       K_{F_{N+1}} ← GenerateKey
6:       PutToDHT(KeyIdentifier(K_{F_{N+1}}), K_{F_{N+1}})
7:       Consolidate(file, K_{F_{N+1}})
8:       cipheredFileMetadata ← Cipher(FileMetadata(file), K_{G_{N+1}})
9:       PutToDHT(FileMetadataIdentifier(K_{G_{N+1}}), cipheredFileMetadata)
10:    cipheredGroupMetadata ← Cipher(GroupMetadata(G), K_{G_{N+1}})
11:    PutToDHT(GroupMetadataIdentifier(K_{G_{N+1}}), cipheredGroupMetadata)
```

Remove User/Leave Group The rationale behind the remove operation is similar to the one for the delete file operation: since there is not made any explicit effort for it to be deleted, a removed member is allowed to keep access to the current state of the group, but from its perspective there will be no new versions.

The remove operation is equivalent whether operating in a history preserving mode or not. Since a member is leaving there is no need to rewrite history, because all members in the groups already had access to the current versions. Instead, generating a new group key and reciphering the metadata is sufficient so that the evicted member can't detect that new versions are available, and generating new file keys so that new versions are ciphered with a different key not available to the member that left, in a way that he will not be able to access the file versions by guessing their identifiers.

Algorithm 6 Remove user from group operation

```
1: procedure REMOVEUSER(G)
2:     Generate K_{G_{N+1}}
3:     Generate K_{F_{N+1}}
4:     files ← ListFiles(GroupMetadata(G), K_{G_N})
5:     for file in files
6:       cipheredFileMetadata ← Cipher(FileMetadata(file), K_{G_{N+1}})
7:       PutToDHT(FileMetadataIdentifier(K_{G_{N+1}}), cipheredFileMetadata)
8:     cipheredGroupMetadata ← Cipher(GroupMetadata(G), K_{G_{N+1}})
9:     PutToDHT(GroupMetadataIdentifier(K_{G_{N+1}}), cipheredGroupMetadata)
```

3.1.3 Orchestration

The solution will be a single client running on every user machine. This client is composed of different components that can be grouped in three categories: Network, cryptography, and storage, as shown is fig. 3.5.

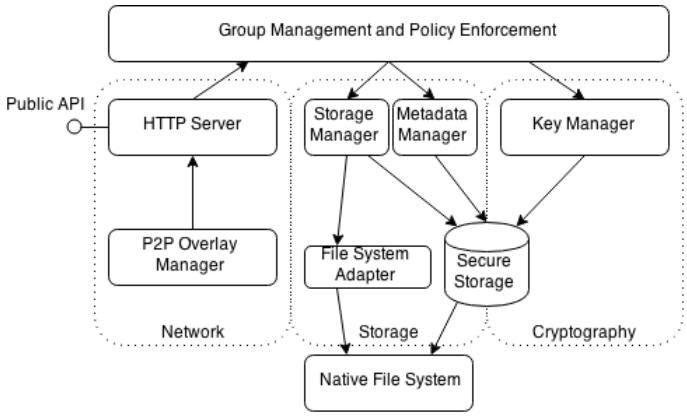


Figure 3.5: Client architecture components

Group Management and Policy Enforcement This component defines the high-level algorithms responsible for managing the essential functionalities of the solution, like handling group events, counting epochs, recognizing user actions, including the previously described algorithms in this section. Essentially, it orchestrate the remaining components in order to offer the sum of the expected functionalities of the solution. It is also comprised of a policy engine that allows to enforce on validate certain conditions concerning running operations of the solution. Such conditions can be used to limit storage capacity for a given user, the replication factor, or even whether or not to use secure storage at all.

A policy is defined by the operation it is applied to, by whether or not the operation should be executed, by a precondition, and by a postcondition. This is intended to be a flexible way of injecting new behavior, rather than using a configuration based system with flags and parameters that require all behavior to be defined beforehand, the policy system allows the definition of functions that modify and extends the solution original behavior without having to change their core components.

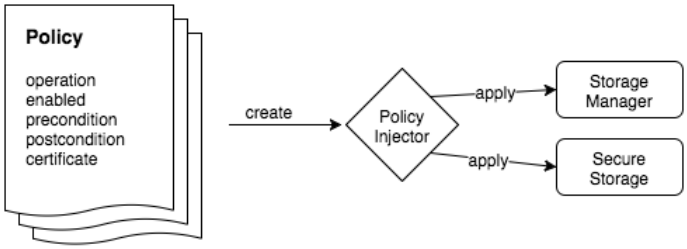


Figure 3.6: Policies and their application

A policy also includes a certificate. This certificate can be used to guarantee that the author of the policy is trusted by the system, and to prevent attackers from creating policies that would disable storage space restrictions or disabling the encryption operation of other users.

Since policies are applied to operations, no action is taken after applying one that is conflicting with the current state. Considering a storage space limit policy, it would not affect the system once applied, but by extending the add file operation, it would prevent new files from being created while more space is used than the new limit.

HTTP Server Every client will have a network server which will be the foundation of all communication, whether it be for the peer-to-peer protocol or for the application itself that will work over a public API.

Peer-to-peer Overlay Manager This component will contain the logic responsible for joining a node to the existing overlay and update the data structures required to maintain the overlay. Note that the P2P implementation is completely orthogonal to the solution.

Other aspects of the solution will also be implemented at this level. Optimisation techniques like erasure-conding and replication, that are only advantageous in a distributed setting, are therefore a problem that this component has to deal with.

Considering, first of all erasure-coding, which in the scenario of retrieving a file version, allows for the parallelisation of the download, given that all chunks that compose the version are transferred simultaneously and results in a performance capped by the transfer time of the slowest chunk. Having erasure-coding working, only at the P2P Manager level, not only simplifies the storing logic of shadow files but it would also be redundant since there is no advantage in physically storing the shadow files as chunks. To support this mechanism, both file identifiers and metadata need to be extended: File identifiers are easy to extend by appending the chunk number at the end, similar to the versions schema, like $-c_i$ given that $i \in [1, C]$ and C is the number of total chunks in which a file is divided; File metadata, besides the keys for each version, also needs to include the number of chunks into which each chunk will be divided.

Replication is another technique that given the context of the solution is only beneficial according to the network layer presence. Replication can be easily achieved by the same technique described earlier, of modifying the file identifiers to include the number of the replica. That is, given a replication factor of $R = 3$, we can simply extend the identifier by adding $-r_i$, given $i \in [0, R-1]$, and storing the multiple replicas in the overlay. This decision also takes into consideration the capability of some overlays to supply replication natively, in which case, it would be able to provide a better dispersion than that guaranteed by the transformation of the file identifier. Considering the transient nature of the network, it is necessary to assure that an healthy amount of replicas are still available. The same mechanism responsible for verifying the latest version of a file must monitor the number of available replicas. It should not take immediate action if less than desired replicas are available, as nodes are allowed to leave the network, starting from the assumption that is likely that they return. Redistribution of replicas should be taken after a period of time of low replica availability or no replication scenario, in which only the current node would have the data.

Key Manager This module will manage the lifecycle of the keys, the creation of new keys and the revocation of old keys. It will also be responsible for mapping those keys to to the respective files, individuals or groups.

In order to map the keys to their respective entities, it is necessary to define the structure of the metadata that supports that information. As previously hinted there are several levels of metadata: information concerning files, groups, and users. Given the hierarchical nature of these concepts, also

the metadata structure can follow the same organization.

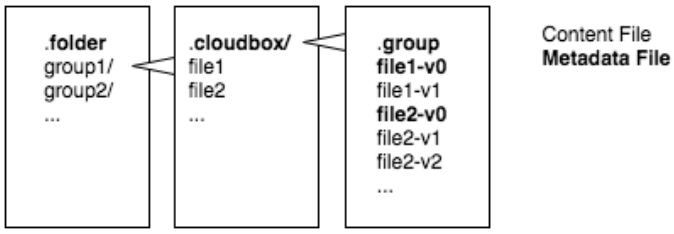


Figure 3.7: Metadata structure

Figure 3.7 shows the structure of the metadata: At the root level the metafile $.folder$ stores the keys for each group, policies and options. For each group there is a shadow folder $.cloudbox/$ that contains both the metadata for the group, in the $.group$ file, and all the files for the partial versions of the group files. The group metafile is the one storing the mapping for file keys and versions.

There are several reasons to use this hierarchic metadata structure: First of all, it obeys to a separation of concern principle that results in disclosing the minimum information required, given that groups don't have access to other groups meta information. Secondly, it simplifies the complexity of the metadata itself, as having it mixed together, would require more information to disambiguate the identity being referred, that eventually would have a greater cost in storage space used. Last of all, this separation is necessary, since the metadata needs to be in the underlying P2P network and otherwise it would not be possible to share group metadata separately.

There is no way for a file belonging to a group to be shared in another group. In order for a file to be shared with different groups, even if the members of one group are a subset of another, it must be individually added to every one of those groups. From the point of view of the system, the same file in different groups is a different file.

Secure Storage This component is depicted as separate from the file system because, even though it is implemented on top of the native file system, it will be an encrypted storage ciphered by the client application.

File System Adapter A component that will connect to the operating system and register the native file system events so that it can detect creation and modification of files.

Storage Manager This component is the one that has the ability to calculate the file modification deltas and reconstruct the file from the various chunks the file is split in.

In case of concurrent edits by different members, a first-come-first-served policy applies. The first to publish the file version identifier is able to keep the changes. The second one will abort and fail the update operation. Conflict management is out of scope for this solution, as it can be implemented on top of it, but it is guaranteed a consistent update operation.

Chapter 4

Implementation

This chapter presents the relevant details about the implementation of the Cloudbox system. The application itself can be described as two different modules: a client daemon, that encapsulates all the functionality described in chapter 3; and a User Interface (UI) that enables the end-users experience. For testing purposes, the implementation features a mocked direct peer communication protocol, as opposed to a full featured P2P overlay.

4.1 Client

The client daemon is responsible for monitoring the file system for changes and synchronizing remote changes. It is written in Scala, a language that mixes object-oriented with functional programming. Since Scala is a JVM language and is interoperable with Java, it is very easy to use the operating system abstractions defined by the Java API, as well as the existing cryptography libraries. Scala is meant to develop highly concurrent and distributed applications — hence the name, Scalable Language —, which is a nice fit for the requirements of Cloudbox.

The initialization process of the client makes a good use of Scala concurrency mechanism, as three parallel execution contexts are launched: an HTTP server, a file system monitor and a performance observer, that introspects the remaining modules for evaluation purposes.

Also during the initialization, the client creates a $Cloudbox$ folder in the user home folder, including a default group to which only the current user belongs, ready to start tracking files. Having a default group, comes at a cost of just the generation of its key. Rather than paying this performance cost when the user actually takes the action to create a group, doing it beforehand also simplifies the structure of the $.cloudbox$ shadow folder for files that would not otherwise belong to any group.

After this initial sequential execution the client is fully event-driven and is listening to the file system, the network and user input.

The decision to have a folder that automatically detects the changes applied to to the file system, rather than a mechanism, like a Command Line Interface (CLI), that would require the user to explicit indicate which files should be tracked, is based on which would be most user-friendly, given that both

require complex implementations. Even though that the first option is slightly more complex, given that it potentially introduces a code portability issue, the usability use case greatly out weights any disadvantage.

4.1.1 Metadata

In early versions of the Cloudbox prototype, given the multitude of executing threads, the metadata would sometimes get corrupted, due to the concurrent writes operations being made. Before implementing a simplistic lock system to regulate concurrent access to the metadata, investigating other alternatives revealed that using SQLite would be a better approach. SQLite is a transactional SQL engine that stores a consolidated state of the data model into a single file. Defining a data model for the metadata allowed for complete abstraction of the metadata implementation that besides solving the concurrency problems made it much easier to use.

Using an Object-relational mapping (ORM) library, makes it possible to access the metadata as a native Scala collection, instead of having to write SQL for reading from and writing to the metadata. It also allows to modify the structure of objects themselves. An interesting modification worth mentioning, is the addition of a special converter, that by annotating certain fields those can be stored ciphered in the metadata itself rather than in plaintext. All the client generated keys are stored this way. Each client is bundled with a unique symmetric key that is only used to be able to cipher those fields.

4.1.2 File System Monitor

Another relevant implementation detail is the choice for the file system monitoring. Since its version 7, that the Java language natively supports attaching to files and be notified of changes to them. Abiding by the concept of File System Adapter defined in chapter 3, the implementation, as fig. 4.1 shows, consists of a Publish-Subscribe (PubSub) system. In this system, the Java API emits events every time that a file is created, modified or deleted in the $Cloudbox$ folder. The $FileSystemNotifier$ class that corresponds to the channel in the PubSub schema, dispatches the events to subscribers that are registered. The $FileRules$ class is registered upon the client initialization.

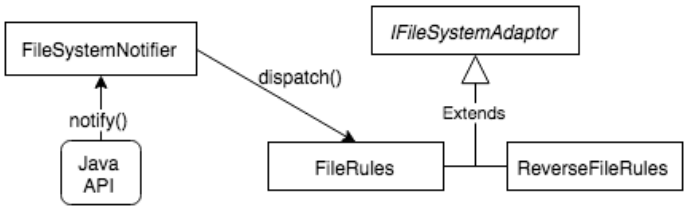


Figure 4.1: $FileSystemAdapter$ Publish-Subscribe

The $ReverseFileRules$ class also implements the $IFileSystemAdapter$ contract, in order to provide the same functionality when the events are generated from the network rather than from the file system. The operation implemented are the same, just slightly different, given that the input data and workflow state is different depending where the event is generated.

4.1.3 Networking

In place of the full featured P2P network, there is instead a direct communication protocol. This protocol is used as stub to provide an easier testing environment. Its interface is, nonetheless, very similar that of a complete P2P implementation and thus integration a full-fledged implementation should be relatively simple. The protocol is divided in two moments:

- **Bootstrap mode** An initial moment, when a joining node still is not connected to the network. In this case, the protocol dictates a bootstrap operation: The joining node searched for a well-known seed node, in order to obtain a manifest containing the identifier of the peers already in the network. If the seed node is not present, the protocol is aborted. Having found the seed node and obtained the address of the remaining connected nodes, the protocol switches to its operational mode.

- **Operational mode** The operational mode is responsible for maintaining the network structure up to date. In a repetitive interval, of ten seconds, each node pings every other node in its know node list. If any one fails to reply to its heartbeat it is removed from the other nodes list. The node remains in this mode for the rest of its execution.

After having the network set up and being constantly maintained, the stub DHT put and get operations are easy to implement: the put operation is a broadcast to all nodes currently in the network; the get operation simply fetches from the local cache of the node, given that all nodes contain all the data. As a result, in this case, we obtain a replication factor of N, being N the number of nodes, which is pretty inefficient from a storage space perspective. But this is just a stub implementation and it is not its objective to be efficient.

For this implementation, the keys exchange protocol is offline, and so, the join operation requires an invitation that includes the group key.

4.1.4 Updating Files

The mechanism for updating files is a parallel execution that, from time to time, tries to fetch for every file the next version to the one that is locally available.

Fetching the new versions is a recursive function:

- If it fails it updates an internal counter of failed updates.

- When it succeeds, it continues fetching updates until there are no more versions to be fetched, and then it fails.

The time that each file is updated depends on the number of previously failed updates and the time of the last modification of the file.

$$TSM(file) = CurrentTime - TimeOfLastModification(file) \tag{4.1}$$

$$TimeToNextUpdate(file) = min(\frac{max(2minutes, TSM(file)) + TSM(file) * \#FailedRetries}{2}, 6hours) \tag{4.2}$$

From eq. (4.2), all files are at best updated once per minute, but then, depending on their age they are checked for updates less often. In the worst case scenario it would take a file six hours to detect updates.

4.1.5 Standards

Some functionalities, as the diff algorithm, encryption algorithms or key generation were not reimplemented, but instead used available libraries, as there are already available handfuls of highly performant and tested libraries for these matters. Nevertheless, it is important to specify the algorithms that they use and configuration parameters that might be meaningful for the performance of the solution and its evaluation.

- The diff algorithm used is the Meyers algorithm [36], which belong to the insert/delete class, as explained in section 2.2.
- The group keys default settings generate a RSA asymmetric key pair of 1024 bits.
- File keys are 128 bits symmetric keys used in the AES encryption algorithm.
- The default chunk size in which file versions are split is 4MB.

4.2 User Interface

The UI module is an HTML and JavaScript application that in addition to being a way to visualize the internal state of Cloudbox, also triggers, via user command, core group operations like creating a group, or fetching the latest version of a group files.

Figure 4.2 shows the detail view of a group files. It essentially corresponds to the usual file system listing with the addition of being able to view the decomposition in versions.

Still, this is not the ultimate purpose of the UI, which is to allow for group operation: fig. 4.3 shows the possible operation for a given group: Create a new group; join an existing one; or leave the current one.

4.2.1 Sharing a group

The way to invite a new user to a group, is by clicking the “Share group” button of a group, as we see in fig. 4.2. Behind the scenes, this causes the consolidation of the group — if operating in a non history preserving mode — and generates an invite. An invite is composed of the group name and the group current key, which is all the necessary information the new member requires to setup his local copy. The

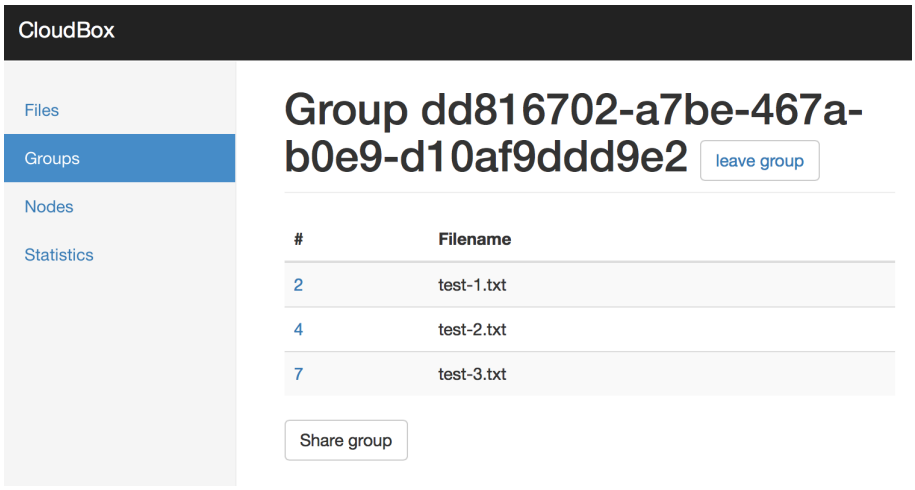


Figure 4.2: Group detail view

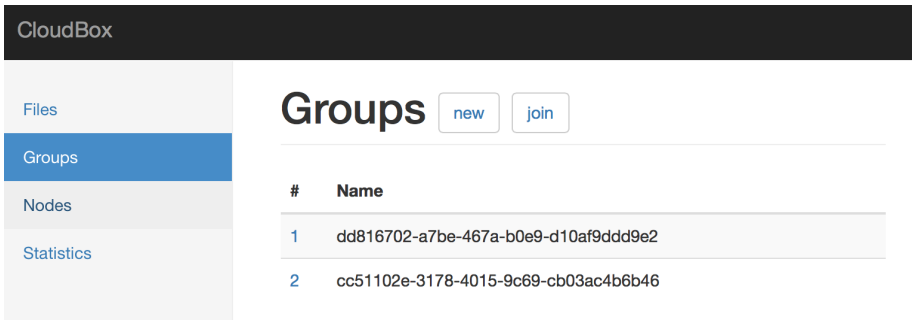


Figure 4.3: Group list view

invite is serialized as a Base64 string, in order to be shared with the joining member. The joining member uses the invite information to fetch the group metadata, which is in turn used to fetch file metadata, file keys and file contents.

Chapter 5

Evaluation

This chapter describes the detailed evaluation of our implementation of the Cloudbox system.

Section 5.1 aims at demonstrating the viability of the Cloudbox system as a secure storage platform. The metrics evaluated metrics are meant to be relevant to the overall performance and scalability of the system.

All the tests were ran on a 2,4 GHz Intel Core i5 with 2GB 1600 MHz of RAM; using the Java(TM) SE Runtime Environment (build 1.8.0_11-b12), Java HotSpot(TM) 64-Bit Server VM (build 25.11-b03, mixed mode) and Scala code runner version 2.11.7.

5.1 Operations Overhead

The data for this section is the result of micro-benchmarking the systems operations and collecting execution metrics, such as CPU, time and space. The protocol to execute this tests is described in appendix A.

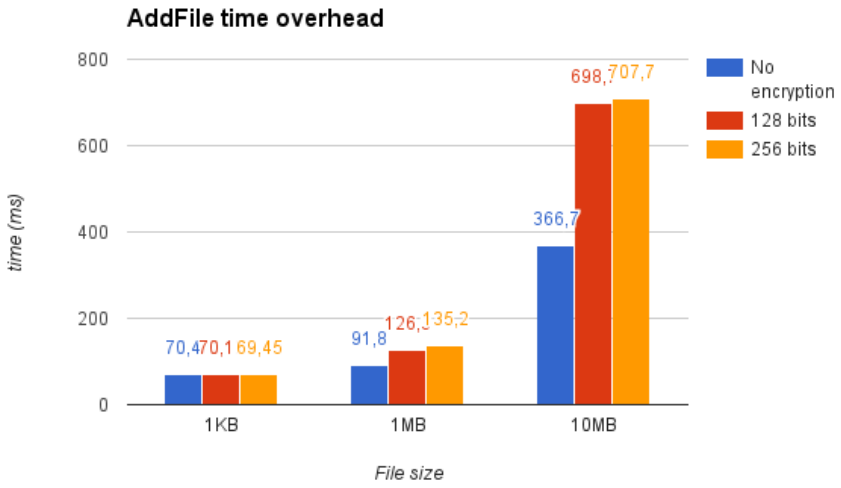


Figure 5.1: AddFile time overhead

Figure 5.1 is the result of the experimental data gather to analyze the impact of adding a group of files to the Cloudbox system. It condenses information from nine test cases, designed to evaluate the performance of the $AddFile$ operation under two variables: file size and file key size. Each test case consists of adding 10 files, simultaneously, to the Cloudbox system, under different operating conditions: encryption turned off, file keys with 128, 192 and 256 bits. The key size 192 is omitted in this graph as it revealed to show little difference from the 128 bits key.

Starting with the 1KB file group, we observe that the amount of data is insufficient to have an impact in the time taken by the ciphering operations, with all sets clocking in at about approximately 70ms, the same time as not having to cipher.

Going over the numbers of the 1MB file group, we observe that the file size starts to make a difference. For the 128 bits keys there is roughly a 37% increase and 47% increase with 256 bits keys.

Finally, the data from the test cases with large files—10MB—reveals a much more noticeable impact of the cipher operations, presenting a slowdown of 90% and 92% for 128 bits and 256 bits, respectively. The unexpected result is how close both ciphers with different key sizes fare. Doubling the key size takes little effect on the overall operation time. If we correlate this information with fig. 5.2, we understand that Cloudbox is able to maintain the ciphering time, regardless of the key size, at the cost of CPU usage. The test case for 256 bits key was able to perform in roughly the same average time, due to a more intense CPU activity.

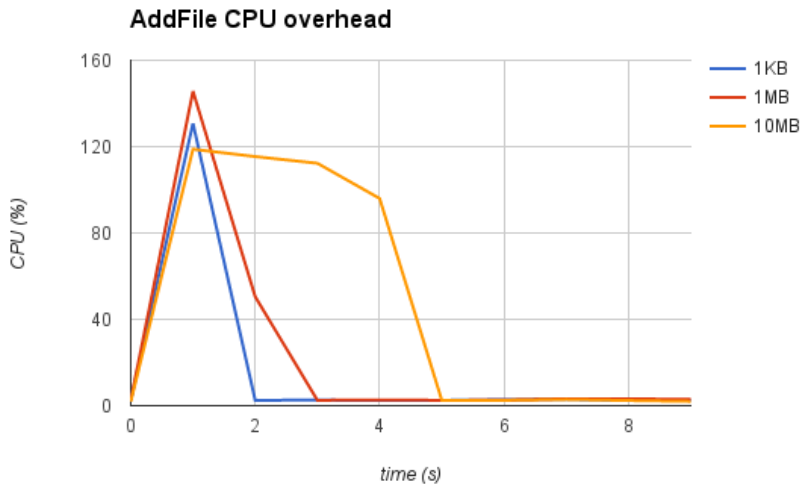
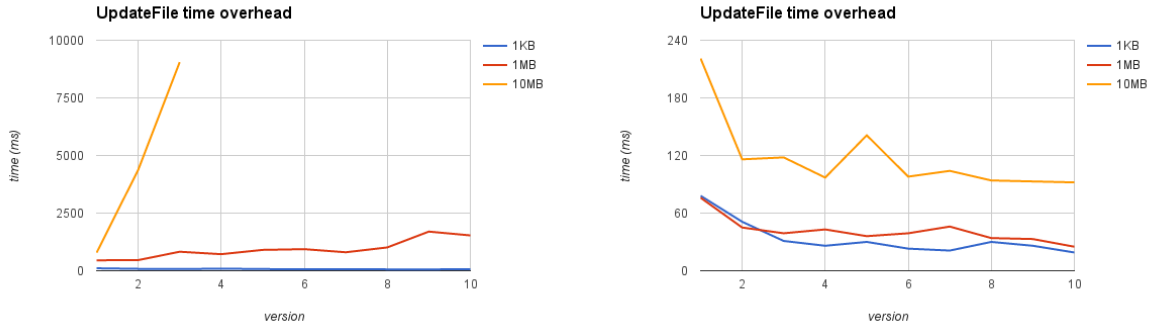


Figure 5.2: AddFile CPU utilization

The results of the test cases for the UpdateFile operation, are presented in two perspectives: one assuming the files are binary or multimedia files and another assuming that they are text based files. These test cases result of initial groups of files of 1KB, 1MB and 10MB, subjected to 10 updates of 1KB.

Considering first the binary files, in fig. 5.3a, for small sizes the implementation is able to handle the updates properly. Still, Increasing the file sizes causes great difficulties for Cloudbox to generate new versions. The time complexity is exponential. It is even unable to continue after the third version of the 10MB group due to insufficient memory, having that last one taken 10 seconds to generate. The problem



(a) Binary files

(b) Text files

Figure 5.3: UpdateFile time overhead

is the composition of the diff algorithm with the ciphering. For binary files, the diff algorithm is unable to generate the version efficiently, considering that the whole file was changed, in such a way that each version can double the size of the previous one. With versions getting exponentially large the cipher algorithm struggles to cipher the versions and thus its bad performance.

Considering text based files, of fig. 5.3b, the results are very different. Since the diff algorithm is able to generate efficient version files and the size of partial versions to be ciphered is reduced, the time overhead is constant with the evolution of the files.

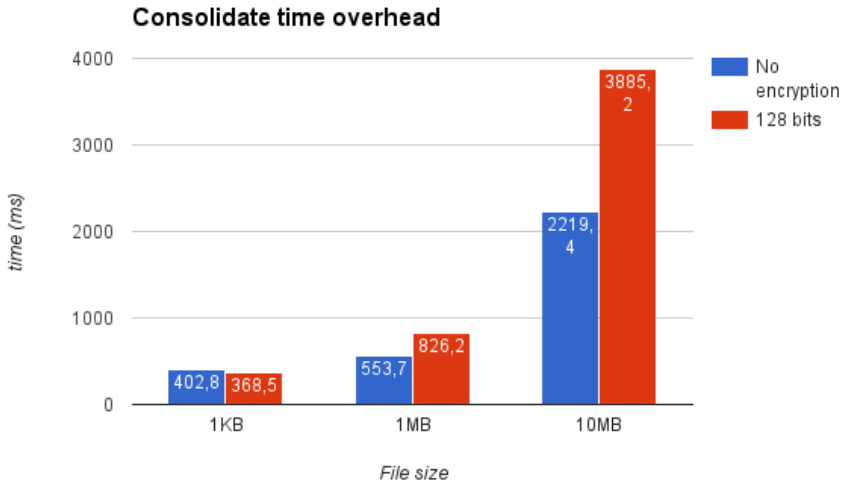
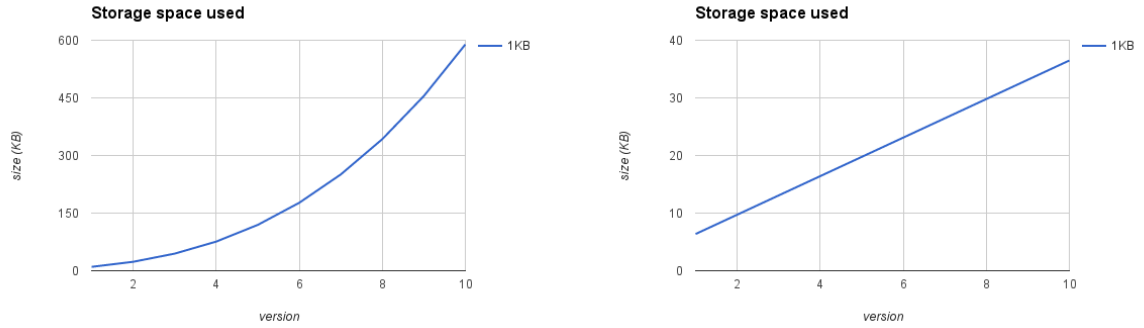


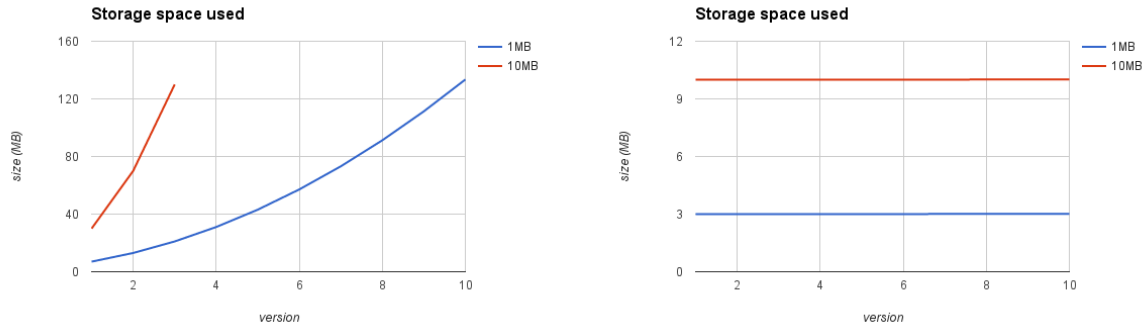
Figure 5.4: Consolidate time overhead

The consolidate operation consists in consolidation of the groups of files generated by the previous test cases: groups of files with 1KB, 1MB and 10MB with 10 updates each of 1KB. Figure 5.4 shows the time overhead of the consolidate operation. Similarly to the add operation, for groups of small files the overhead is low. As expected it grows with the size of the files, about 49% and 75% for 1MB and 10MB files. The consolidate operation is the first to surpass the one second threshold, after which the delay is noticeable to humans, but still, we must consider that the operation is being applied to all files in the



(a) Binary files (b) Text files

Figure 5.5: Storage space used - 1KB file



(a) Binary files (b) Text files

Figure 5.6: Storage space used - 1MB and 10MB files

group and its occurrence is very punctual in the lifecycle of a group.

5.2 Storage and Metadata

A critical measure to evaluate the Cloudbox system is its storage space requirements. The following results were extracted from the previous described file update experiment. The same groups of files—1KB, 1MB and 10MB—, and with set of binary and text based files.

Figure 5.5a and fig. 5.5b display the evolution of space used for a file of 1KB, through 10 updates. At its initial version a 1KB uses about 6KB of space on the local system. Subsequent version updates of 1KB add roughly 3KB to the used space total. The expansion is, obviously caused by the stored ciphertext. Like previously mentioned, the diff algorithm used is unable to produce efficient differences of the binary changes and fig. 5.5a shows the exponential growth in used space. Figure 5.6a and fig. 5.6b display the same results, separated for the sake of the charts scale. Here the versions updates were also of 1KB. We can grasp the efficiency of using file versioning, as updates to large files have almost no impact in the amount of space used.

The last result of the metadata evaluation shows how it evolves. Figure 5.7 demonstrates that

changes to any file are innocuous to the metadata size. Since metadata is only affected by the number of files or, to a lesser degree, by group changes, it was expected that changes to the files would not affect it.

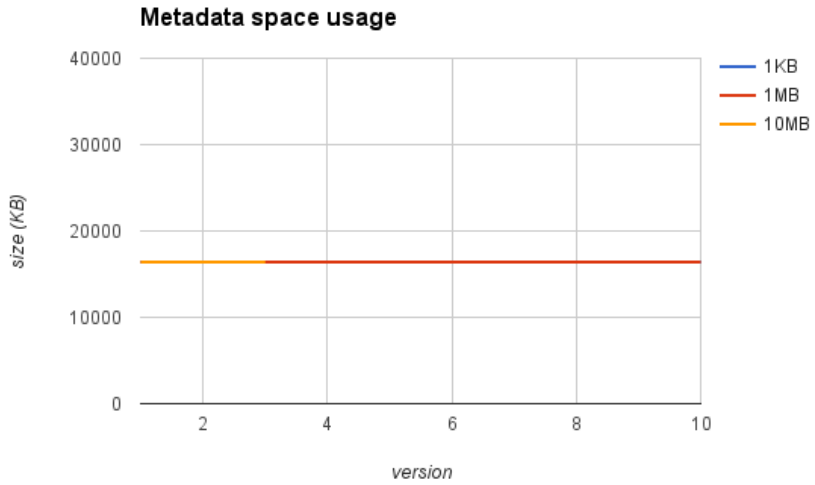
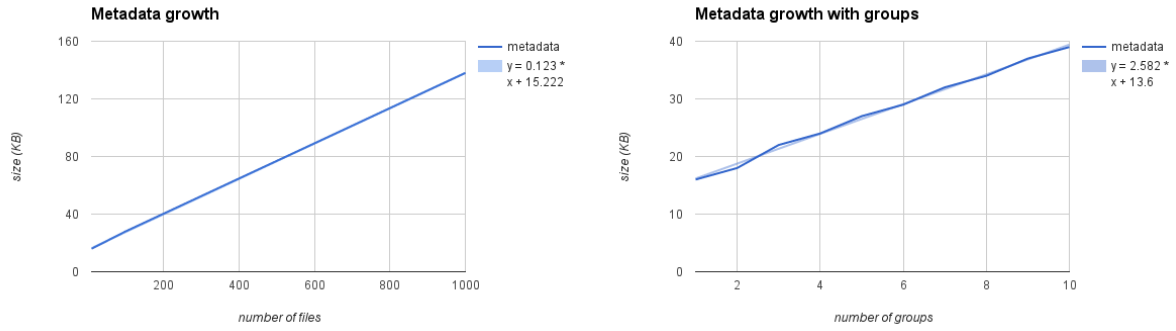


Figure 5.7: Metadata size over file evolution

Both fig. 5.8a and fig. 5.8b confirm just that, the factor by which metadata grows with the number of files tracked by the system, and with the number of groups present in the system. The growth is linear in both cases. The second case has a faster growth rate due to the larger amount of data that it has to track. Nonetheless, in any case, the size of the metadata is not bound to the size of file, and starts with a small overhead that grows linearly and is negligible considering the actual space used to store files.



(a) Metadata by number of files

(b) Metadata by number of groups

Figure 5.8: Metadata storage space usage

Chapter 6

Conclusion

Cloud services will play a very important role in the technological future. Its ubiquity is very exciting and is evermore enabling greater productivity and usability for its users. Still, cooperative cloud services are scarce, not that they are worse in any way, but simply because by being a more complex approach and harder to control, rather than the traditional centralized system, they dissuade cloud services providers.

In this thesis, we proposed a solution that combines pre-existing techniques to create a secure storage solution with the ability to share directories with other users. Our solution shows that security does not need to be a tradeoff for collaboration, as existing systems lead to believe. Both are attainable, even in a scenario where the network infrastructure is not trustworthy.

6.1 Concluding Remarks

This report presented a study of the state-of-the-art of peer-to-peer topologies and cryptography. The analysis of these topics allowed us to critically analyze the existing systems and identify aspects that can be improved in order to provide a better level of security: centralized infrastructure; server-side encryption and no forward secrecy.

Once identified the shortcomings, we proposed a solution that considers the files and the groups themselves in epochs, in order to achieve efficient group membership, and using versioning to efficiently support file ciphering operations.

In our evaluation, the results showed marginal overhead on the critical operations, good scalability when dealing with text-based files, and all of that in sub-second time for almost all scenarios.

6.2 Future Work

There is much work that can be done to improve our solution. By conducting research and further experiments on specific topics of the solution we can get great optimizations. We can also find ways to make the solution easier to use by adding new functionalities. Here are some examples:

- **Ahead of time consolidate** Consolidating ahead of time can save a lot of time and network traffic depending on when it is executed. However, when to consolidate is a decision that would require large usage datasets analysis, in order to be sure of the appropriate moment. For that reason, we think that finding that sweet spot is worth a research of its own.

- **Adaptative diff algorithm** From our evaluation, we have observed that the diff algorithm used in the implementation results in subpar performance for binary files. It would be nice to have a mechanism that would scan the files when they are added to a group and detect the most appropriate diff algorithm to apply to each file, and store that information in the metadata. This way, all kind of files would have efficient versioning.

- **Conflict resolution** Solving concurrent updates was always out of the scope of this work, but it is a very interesting, and well studied, problem to be addressed. Specially considering that when a conflict occurs, both updates are the calculated differences from the same source file. There are already many version control systems that can solve this problem without the need for human intervention. It would be interesting to see how their techniques can be applied and integrated in Cloudbox.

Bibliography

[1] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, S. Lim, *et al.*, "A survey and comparison of peer-to-peer overlay network schemes.," *IEEE Communications Surveys and Tutorials*, vol. 7, no. 1-4, pp. 72–93, 2005.

[2] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM Computing Surveys (CSUR)*, vol. 36, no. 4, pp. 335–371, 2004.

[3] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[4] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Looking up data in p2p systems," *Communications of the ACM*, vol. 46, no. 2, pp. 43–48, 2003.

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A scalable content-addressable network*, vol. 31. ACM, 2001.

[6] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao, "Distributed object location in a dynamic network," *Theory of Computing Systems*, vol. 37, no. 3, pp. 405–440, 2004.

[7] C. G. Plaxton, R. Rajaraman, and A. W. Richa, "Accessing nearby copies of replicated objects in a distributed environment," *Theory of Computing Systems*, vol. 32, no. 3, pp. 241–280, 1999.

[8] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware 2001*, pp. 329–350, Springer, 2001.

[9] D. Malkhi, M. Naor, and D. Ratajczak, "Viceroy: A scalable and dynamic emulation of the butterfly," in *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pp. 183–192, ACM, 2002.

[10] M. Ripeanu, I. Foster, and A. Iamnitchi, "Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design," *arXiv preprint cs/0209028*, 2002.

[11] B. Cohen, "Incentives build robustness in bittorrent," in *Workshop on Economics of Peer-to-Peer systems*, vol. 6, pp. 68–72, 2003.

[12] S. Hand and T. Roscoe, “Mnemosyne: Peer-to-peer steganographic storage,” in *Peer-to-Peer Systems*, pp. 130–140, Springer, 2002.

[13] E. A. Brewer, “Towards robust distributed systems,” in *PODC*, p. 7, 2000.

[14] K. Church, A. G. Greenberg, and J. R. Hamilton, “On delivering embarrassingly distributed cloud services.,” in *HotNets*, pp. 55–60, Citeseer, 2008.

[15] P. Kulkarni, F. Douglis, J. D. LaVoie, and J. M. Tracey, “Redundancy elimination within large collections of files.,” in *USENIX Annual Technical Conference, General Track*, pp. 59–72, 2004.

[16] M. W. Storer, K. Greenan, D. D. Long, and E. L. Miller, “Secure data deduplication,” in *Proceedings of the 4th ACM international workshop on Storage security and survivability*, pp. 1–10, ACM, 2008.

[17] J. R. Douceur, A. Adya, W. J. Bolosky, P. Simon, and M. Theimer, “Reclaiming space from duplicate files in a serverless distributed file system,” in *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pp. 617–624, IEEE, 2002.

[18] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, *et al.*, “Oceanstore: An architecture for global-scale persistent storage,” *ACM Sigplan Notices*, vol. 35, no. 11, pp. 190–201, 2000.

[19] R. Rodrigues and B. Liskov, “High availability in dhts: Erasure coding vs. replication,” in *Peer-to-Peer Systems IV*, pp. 226–239, Springer, 2005.

[20] H. Weatherspoon and J. D. Kubiatowicz, “Erasure coding vs. replication: A quantitative comparison,” in *Peer-to-Peer Systems*, pp. 328–337, Springer, 2002.

[21] T. Suel and N. Memon, “Algorithms for delta compression and remote file synchronization,” 2002.

[22] J. MacDonald, *File system support for delta compression*. PhD thesis, Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[23] S. M. Diesburg and A.-I. A. Wang, “A survey of confidential data storage and deletion methods,” *ACM Computing Surveys (CSUR)*, vol. 43, no. 1, p. 2, 2010.

[24] C. P. Wright, J. Dave, and E. Zadok, “Cryptographic file systems performance: What you don’t know can hurt you,” in *Security in Storage Workshop, 2003. SISW’03. Proceedings of the Second IEEE International*, pp. 47–47, IEEE, 2003.

[25] T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann, and L. Uhsadel, “A survey of lightweight-cryptography implementations,” *IEEE Design & Test of Computers*, vol. 24, no. 6, pp. 522–533, 2007.

[26] G. Ateniese, M. Steiner, and G. Tsudik, “New multiparty authentication services and key agreement protocols,” *Selected Areas in Communications, IEEE Journal on*, vol. 18, no. 4, pp. 628–639, 2000.

[27] W. Diffie, P. C. Van Oorschot, and M. J. Wiener, “Authentication and authenticated key exchanges,” *Designs, codes and cryptography*, vol. 2, no. 2, pp. 107–125, 1992.

[28] S. M. Bellovin and M. Merritt, “Encrypted key exchange: Password-based protocols secure against dictionary attacks,” in *Research in Security and Privacy, 1992. Proceedings., 1992 IEEE Computer Society Symposium on*, pp. 72–84, IEEE, 1992.

[29] M. Steiner, G. Tsudik, and M. Waidner, “Diffie-hellman key distribution extended to group communication,” in *Proceedings of the 3rd ACM conference on Computer and communications security*, pp. 31–37, ACM, 1996.

[30] Y. Kim, A. Perrig, and G. Tsudik, “Group key agreement efficient in communication,” *Computers, IEEE Transactions on*, vol. 53, no. 7, pp. 905–921, 2004.

[31] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras, “Inside dropbox: understanding personal cloud storage services,” in *Proceedings of the 2012 ACM conference on Internet measurement conference*, pp. 481–494, ACM, 2012.

[32] “Your stuff is safe with dropbox.” `https://www.dropbox.com/security#protection`. Accessed: 2014-11-30.

[33] I. LogMeIn, “Cubby: A secure solution,” 2014.

[34] K. Lissounov, “Bittorrent sync: Security is our highest priority,” 2014.

[35] “When are my files safe?.” `http://support.spacemonkey.com/customer/portal/articles/1384011-when-are-my-files-safe-`. Accessed: 2014-11-30.

[36] E. W. Myers, “Ano (nd) difference algorithm and its variations,” *Algorithmica*, vol. 1, no. 1-4, pp. 251–266, 1986.

Appendix A

Micro-Benchmarmink Test Protocol

To run the benchmark for the Cloudbox system follow these steps:

1. Consult the $CloudboxTests$ class for the desired test case name
2. Export an environment variable named $CLOUDBOX\_TEST$ with the value being the desired test case name
3. Make sure that the system is starting in a clean state by running the script $./script/clean.sh$
4. Run the application with $sbt\ run$
5. After the tests finished message is presented press $Ctrl + D$ to obtain the metrics captured.