Cloudbox
Private, Reliable and Distributed Storage

Rafael Cortês
rafael.cortes@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2015

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

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.

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 phones, digital cameras and other multimedia devices, that generate new forms of media. Each year new 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 system is thus a critical problem if we intend to keep the innovation rate of multimedia peripherals and sharing 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 systems' scalability.

2 Related Work

For this work we surveyed the state of the art for Peer-to-peer (P2P) [10] [3] in order to understand how their organization and structure would affect the solution's performance.

We have also analyzed different optimization techniques such as Geo-replication [4], Data deduplication [7], Erasure-coding[6] and delta enconding[12], to

evaluate how they could help in building an efficient solution.

In the cryptography area we surveyed secure storage mechanism [5] [13] to have an understanding of the requirements for secure storage system and their organization, so that we could develop our solution in the most appropriate way.

Finally, as for existing systems, Dropbox[2] is the most popular cloud storage service. It distinguishes itself from its competitors by offering easy to use sharing features. Other reviewed systems, such as Cubby[9], BitTorrent Sync [8] and [1] are the major existing solutions in this area. There are countless more services for online storage, but they do not differ in the technologies used from those already mentioned. Table 1 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.

These systems have problems that prevent them from being a truly secure storage service. 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. 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.
- **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 unauthorized 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.

3 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 File-level Operations

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, 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.

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 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*.

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. To guarantee the anonymity of the content stored, the file identifier is itself ciphered with the owner's key, in a way that

	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 1: Technologies employed by the described solutions

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: end procedure
6: procedure CreateVersion(content, K_F)
7:     cipheredContent ← Cipher(content, K_F)
8:     PutToDHT(VersionIdentifier(K_F), cipheredContent)
9: end procedure
```

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.

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.

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)
5: end procedure
```

Consolidate Finally, the consolidate operation, at file level, is a combination of the create and update operations. It can be defined as presented in algorithm 3.

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})
10: end procedure
```

3.2 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. The second mode, showcased in fig. 1, hides the history of a file for new members at the cost of consolidating the file 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.

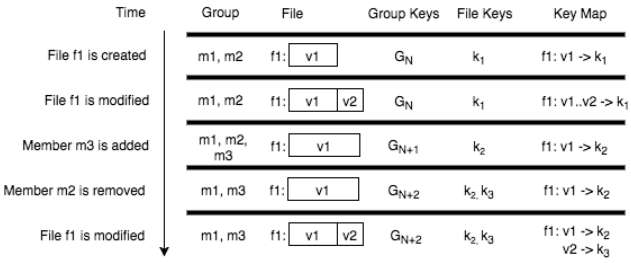

Figure 1: File versioning evolution without history

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. alg:group-add-no-history detail the add operation, in the mode that does nit expose any previous group history. For that, 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. In the mode that preserves history, no action needs to be taken other than distribute the current group key to the new member.

Algorithm 4 Add user to group operation preserving history

```
1: procedure AddUser(G)
2:     Distribute(K_G)
3: end procedure
```

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)
12: end procedure
```

Remove User/Leave Group 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)
10: end procedure
```

3.3 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. 2.

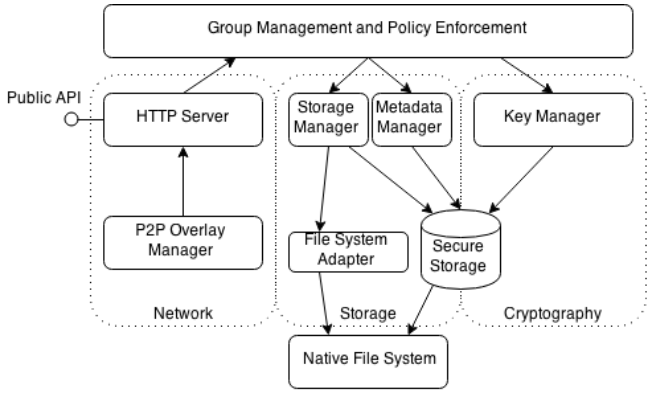

Figure 2: Client architecture components

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.

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 Application Programming Interface (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.

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.

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.

4 Implementation

This section 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 section 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 section 3, the implementation, as fig. 3 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.

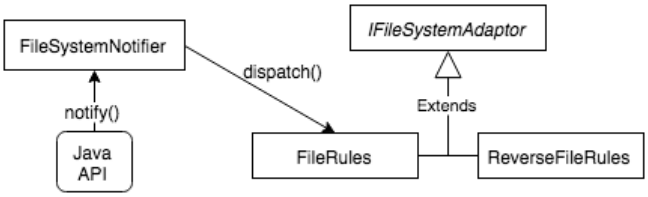

Figure 3: $FileSystemAdapter$ Publish-Subscribe

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 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 [11], which belong to the insert/delete class.
- 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.

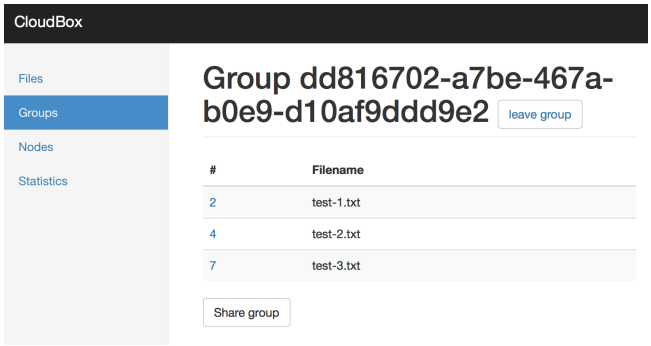

Figure 4: Group detail view

Figure 4 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. 5 shows the possible operation for a given group: Create a new group; join an existing one; or leave the current one.

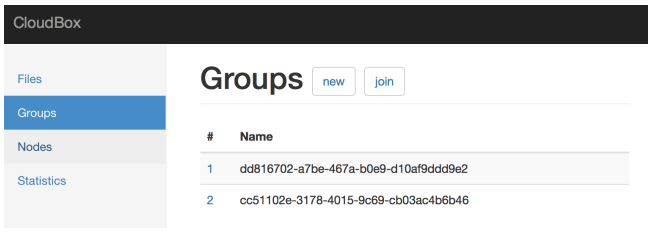

Figure 5: Group list view

5 Evaluation

This section 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.

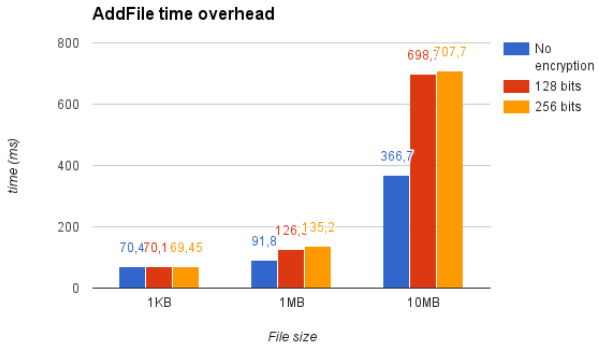

Figure 6: AddFile time overhead

Figure 6 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 perform. Doubling the key size takes little effect on the overall operation time. If we correlate this information with fig. 7, 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.

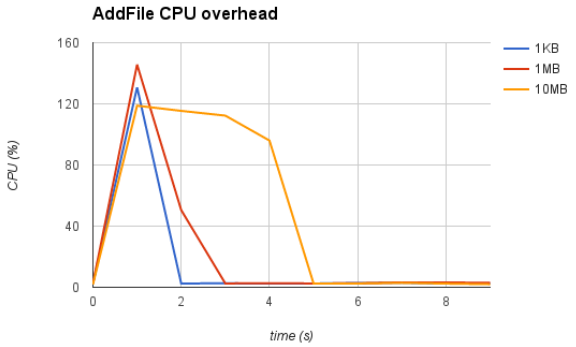

Figure 7: AddFile CPU utilization

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 section 5.1, 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 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 section 5.1, 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.

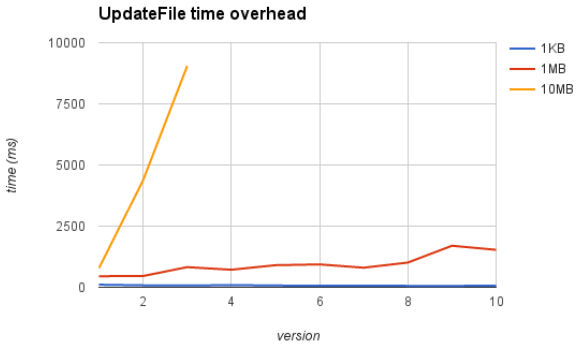

Figure 8: UpdateFile time overhead for binary files

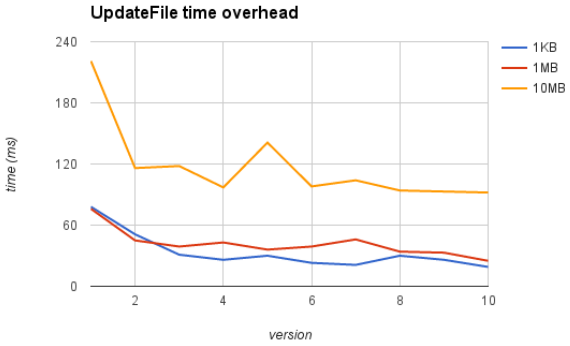

Figure 9: UpdateFile time overhead for text files

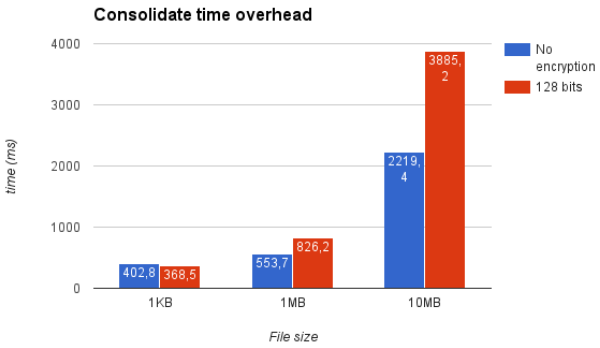

Figure 10: Consolidate time overhead

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 10 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 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.

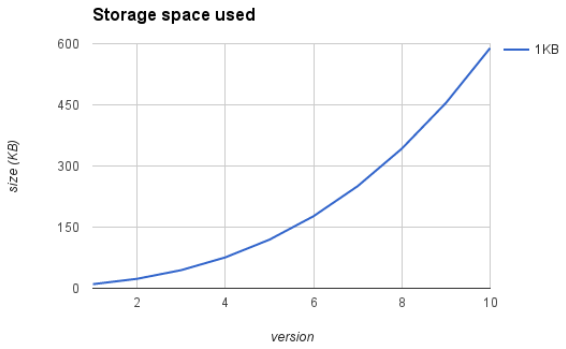

Figure 11: Storage space used for binary files - 1KB file

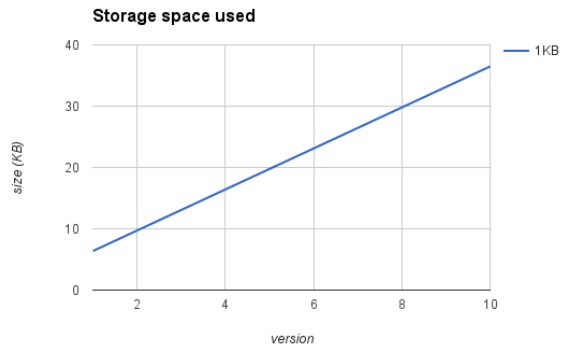

Figure 12: Storage space used for text files - 1KB file

Figure 11 and fig. 12 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 cipher-text. Like previously mentioned, the diff algorithm used is unable to produce efficient differences of the binary changes and fig. 11 shows the exponential growth in used space. The results for the 1MB and 10MB groups lead to the same conclusions, omitted here for the sake of the charts scale. 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. Section 5.2 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. Actually, the factor by which metadata grows with is the number of files tracked by the system or changes to the group membership.

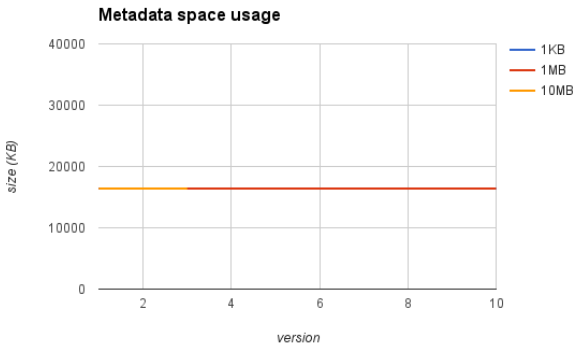

Figure 13: Metadata storage space usage over file evolution

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 report, 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 subsecond 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.

References

[1] When are my files safe? `http://support.spacemonkey.com/customer/portal/articles/1384011-when-are-my-files-safe-`, accessed: 2014-11-30

[2] Your stuff is safe with dropbox. `https://www.dropbox.com/security#protection`, accessed: 2014-11-30

[3] Androutsellis-Theotokis, S., Spinellis, D.: A survey of peer-to-peer content distribution technologies. ACM Computing Surveys (CSUR) 36(4), 335–371 (2004)

[4] Church, K., Greenberg, A.G., Hamilton, J.R.: On delivering embarrassingly distributed cloud services. In: HotNets. pp. 55–60. Citeseer (2008)

[5] Diesburg, S.M., Wang, A.I.A.: A survey of confidential data storage and deletion methods. ACM Computing Surveys (CSUR) 43(1), 2 (2010)

[6] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., et al.: Oceanstore: An architecture for global-scale persistent storage. ACM Sigplan Notices 35(11), 190–201 (2000)

[7] Kulkarni, P., Douglis, F., LaVoie, J.D., Tracey, J.M.: Redundancy elimination within large collections of files. In: USENIX Annual Technical Conference, General Track. pp. 59–72 (2004)

[8] Lissounov, K.: Bittorrent sync: Security is our highest priority (2014), `http://forum.bittorrent.com/topic/32592/`

[9] LogMeIn, I.: Cubby: A secure solution (2014), `https://www.cubby.com/welcome/common/resources/Cubby_Security_Whitepaper.pdf`

[10] Lua, E.K., Crowcroft, J., Pias, M., Sharma, R., Lim, S., et al.: A survey and comparison of peer-to-peer overlay network schemes. IEEE Communications Surveys and Tutorials 7(1-4), 72–93 (2005)

[11] Myers, E.W.: Ano (nd) difference algorithm and its variations. Algorithmica 1(1-4), 251–266 (1986)

[12] Suel, T., Memon, N.: Algorithms for delta compression and remote file synchronization (2002)

[13] Wright, C.P., Dave, J., Zadok, E.: 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)