LIBRARY DEVELOPMENT FOR STORJ CLOUD CLIENTS IN UNSUPPORTED ENVIRONMENTS

Based on experiences in an Android environment

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Bachelor's thesis

Degree Programme in Business Information Technology

Visamäki – Hämeenlinna – Finland

Spring 2017

Gabriel Comte

ABSTRACT

Degree Programme in Business Information Technology
Visamäki – Hämeenlinna – Finland

Author Gabriel Comte **Year** 2017

Subject Library development for Storj cloud clients in unsupported environments

Supervisor Lasse Seppänen

ABSTRACT

The thesis illustrates the differences between presently common cloud architectures which are traditionally of a centralized form and decentralized cloud architectures. The latter particularly pays attention to the decentralized cloud provided by Storj Labs. Researching the Storj cloud further, it explains the advantages that its architecture entails and presents some of the difficulties coming with it.

The main aim of the thesis is providing information for developers on how to programmatically access the Storj cloud for building client software, especially when working in an environment, for which there is no supporting library provided by Storj Labs or any third party. The thesis furthermore reveals, that many processes of a Storj cloud client are radically different from what a traditional cloud client is like, as well as how they differ from the traditional implementations.

The main topics approached in this thesis are the authentication against the cloud, the up- and download of data to respectively from the cloud with all the various steps it includes, encrypting and decrypting this data and a detailed description of the concept of *sharding*, which is fundamental to the concept of Storj Labs' cloud service. These topics are approached in a way, which provides compatibility for the resulting libraries against the libraries provided by Storj Labs, as this is a necessary measure to provide data portability over different systems.

The provided information is acquired and validated by a reference implementation that had been developed as a part of this thesis. As this implementation is an Android app, the libraries and the code that is provided in the thesis are specifically applicable for Android projects.

Keywords Decentralized clouds, Storj, cryptography, trustless software solutions

Pages 67 pages including appendices 10 pages

TERMINOLOGY

Client	Whenever used in this thesis, the term ought to be understood in the IT-related meaning: “software that accesses a remote service”
Trustless systems	Systems that work in ways that do not require any trust between the various actors on the system
Storj Labs	A Company providing decentralized cloud storage, incorporated in 2015 and based in Atlanta, Georgia
Storj cloud	The decentralized cloud, maintained by Storj Labs
Storj network	The network the Storj cloud is based on
Farmer	Devices that offer storage inside of the Storj network
Renter	Devices that consume storage provided by the farmers
Bridge	A component of the Storj network, which supports the clients in the client-side tasks
Shard	A fraction of a file, result of *sharding* a file
Mirroring	Duplicating shards from one farmer to multiple farmers in order to establish redundancy for each shard
Data audit	are used to examine, whether farmers do actually do store the shards they agreed to store (in automated contracts)
Challenges	The input used to generate audits. 32 bytes of random data
Exchange reports	Are reports which the clients send to the Bridge, after each shard upload or download, providing information about the process to the Bridge.
Symmetric key	Key which is used for both encrypting and decrypting data. Unlike asymmetric encryption, where there is one key used to encrypt data and another one the decrypt the resulting data
Initialization Vector	A cryptographic primitive, providing a measure to secure encryption algorithms from attacks. Is random data
Nonce	Abbreviation for ***n**umber used **once***. Is random data. Used in cryptography to secure authentications. Must only be used once
Entropy	Collected randomness. In this thesis used for random data
Mnemonic	Representation of an entropy in multiple human understandable words. Comes in sizes from 12 to 24 words, depending on the entropy’s size
Concurrency	A property of a system or an algorithm allowing multiple actions to be executed at the same time

CONTENTS

1 INTRODUCTION

In the last few years, cloud computing has turned into a vast and ever-growing market, which has attracted various enterprises to establish themselves in this new business field. But while these competing enterprises' cloud services might differ on many levels, they also have one key aspect in common. The fundamental system designs of their clouds are all similar; they are always based on a centralized IT architecture.

A start-up called *Storj Labs* has a different vision on how cloud computing should be designed. Since 2015 they are building up a cloud service with a radically different system architecture, which could bring benefits over traditional clouds in many aspects. In a nutshell, Storj Labs is trying to create a cloud storage service without running any storage server themselves. Instead, they are creating a network, in which private individuals may provide their disk space and thus *be the cloud*[1] themselves.

Storj Labs is however operating on a limited scope. It has its focus on providing a cloud storage service as a resource for cloud developers, not on building any clients nor any services for end-users.
As this is a very recent project and Storj Labs does not intend to provide any client software itself, there is a resulting lack of possibilities to access the Storj cloud for end-users. This creates new opportunities for third party developers. Nevertheless, these opportunities are to be enjoyed carefully; as the project is very young, there are only a few people with the relevant experience and even fewer documentation. These problems are aggravated by the fact that there are (still) many environments for which Storj Labs does not provide any library.
This thesis is an attempt to fill this gap by giving further insights on the details to be considered when connecting client systems to the Storj cloud, particularly for the cases in which a client library implementation is necessary. The insights are proven by a reference implementation of a Storj cloud client Android library and an Android app that integrates this library.

Within the framework of these investigations, the key objective is addressing the subsequently mentioned research questions.
What are the main obstacles in developing a Storj cloud client library?
What aspects must be considered to ensure compatibility with the libraries provided by Storj Labs?
Is the Storj environment already stable enough for developing a sustainable library?

[1] Slogan of Storj Labs

2 INITIAL SITUATION

2.1 Cloud computing

Cloud computing has rapidly grown into a vast market over the past years. Yet, the process of growth is still on-going; according to Gartner, the movement into the cloud or “Cloud Shift” as they call it, will surpass US$ 1 trillion in global IT spending by the year 2020 (Gartner Inc, 2016). Cloud computing is penetrating economies on a global scale and the term “cloud” has resounded throughout the land, as it appears in many different sectors, way beyond the IT sector alone.

But what is “the cloud” from a technical point of view? The US American *National Institute of Standards and Technology NIST* defines clouds among other characteristics as a model for "on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications and services)" (P. Mell, 2011).
This thesis focuses particularly on the resource *storage,* which is why in this thesis, the term cloud is mainly referring to space provided somewhere in the internet, which allows clients of a given cloud to upload and download data from respectively to it. The actors behind these clients might range from entire cloud solutions up to users who directly access the cloud storage.

From this perspective, taking Dropbox as an example could help to illustrate an answer to the question about what cloud storage really is. Dropbox stores all its customers’ data on servers in the United States of America (Dropbox, 2017). Their cloud is thus substantially nothing else than a large cluster of servers. Dropbox’ structure is hence inherently of a centralized architecture, with its customers’ data gathered on centralized servers. But this structure is not only used by Dropbox. In fact, quite the opposite is the case. Dropbox’ cloud model is indeed representative of how cloud computing is factually done at present (Jaeger et al., 2009, S. Chaper 2: What is the cloud?).
In this thesis, this traditional architecture of current cloud systems is further on referred to as a *centralized cloud architecture*. Figure 1 illustrates this kind of architecture.

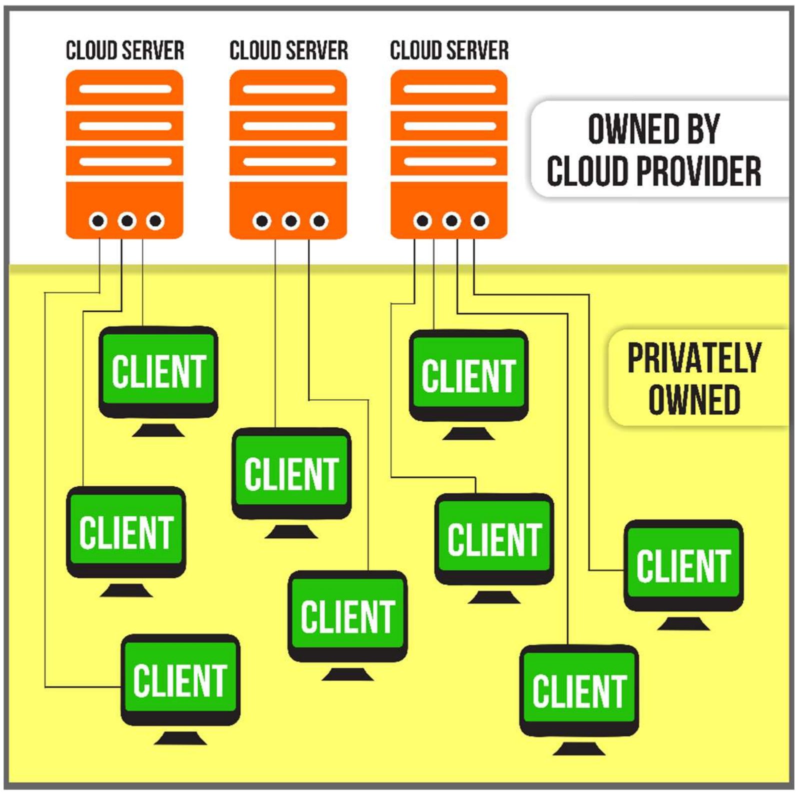


Figure 1: a centralized cloud architecture

Please Note: Whenever this thesis mentions the term *cloud clients*, this involves a broader spectrum of applications than just the Dropbox equivalent of an online file hosting service, which enables its users to upload and download data. The spectrum involves all use cases where cloud storage is needed as a resource. This involves all kinds of applications such as image hoster, social networks, streaming services, content management systems, email providers and many more.

2.2 **Weaknesses of centralized cloud architectures**

There are various inherent problems stemming from such a centralized model. This chapter covers the most common ones. There are certainly various sets of solutions used by various cloud system providers, to curb the extent of these weaknesses. One could however argue, that these solutions are just attempted fixes, to reduce the drawbacks created by the unfavourable underlying system architecture.

2.2.1 Privacy

Customers' privacies depend on their trust towards the respective cloud hoster; the hosting company can fully access all of its users' data which gives the hoster the power to do whatever it wants to with the data. Privacy abuse is hard to detect for a user, as the data is fully out of the users control and often even in another country than the user himself / herself.

In addition to this, many well-established cloud hosters have their centralized data centres in the United States of America, which makes them subject to the PATRIOT act (United States Department of Justice, 2001) as well as the Homeland Security Act (Public Law of USA, 2002) and other existing and coming laws of the US government. (Jaeger, P., Lin, J., Grimes, J., Simmons, S., 2009, S. Chapter 5: What rules govern the cloud?) Client-side encryption is an established measure to address this issue, but is in practice often not applied.

Another threat to a user's privacy are all kinds of hacker attacks. What makes this problem even worse on centralized cloud systems, is that the location of centralization – whether digital or physical – is publicly known. Thus, whenever attackers know on what cloud service somebody's data is stored, they also know where to attack.

2.2.2 Availability and data loss

Redundancy is a common measure for increasing both data availability and data protection against permanent loss. In a centralized cloud architecture, the level of redundancy to be provided is rather limited, as redundancy should include geographical distribution, which causes high costs.
Amazon S3 provides an availability of 99.99% for their standard storage (Amazon webservices, 2017). This means that the data may still be inaccessible for 52.56 minutes in total per year. Even though this availability seems to be impressive at first, it is revealed as actually not that big, if you consider that only 4 independent servers with only 90% uptime each would already provide this availability:

$$data\ availability = 1 - 0.1^4 = 0.9999 = 99.99\%$$

An architecture making it possible to involve significantly more servers could massively increase the data availability and durability, even without the particular servers being very reliable themselves.

2.2.3 Performance

The performance of a centralized cloud is usually low by design, since all traffic is going to and coming from a centralized cluster; the possible traffic rate is used by many users at the same time and therefore divided between them.
This problem is further increased by a geographical problem; if a cloud with global customers is centralized, this inherently means that some of the clients will be geographically far away from this cloud. These users would consequently have a slower communication with the cloud, due to the big distances their data must travel.

2.2.4 Price

The clients of a centralized cloud system are privately owned and automatically added to the system by the users without any further action from the cloud providers being necessary. All the costs for the clients are paid by the users. This involves costs for the clients' hardware (computer, smartphone, …), price of the internet access, electricity expenses, Software costs, maintaining, and support.
On the contrary, this very same fact is not true for the system part that stores the uploaded data respectively the centralized part of the system. The centralized data centres must be built up and maintained by the cloud provider, which generates high costs for them.

3 STORJ LABS' VISION OF A DECENTRALIZED CLOUD

3.1 Preface

The approach Storj Labs follows for building their cloud system involves many interesting questions, discussions, solution approaches and new concepts and technologies. The Storj project is in many ways related to the Blockchain sector, as many technologies used by Storj Labs are inspired by other projects in this environment. Yet many parts of the Storj project are deliberately left out in this thesis, as they are irrelevant to the core issues of the thesis' topic. While a fair amount of information about the Storj cloud can be found by searching the internet, this thesis is meant to focus on information that is more difficult to access.
If more information about the Storj cloud itself is desired, a recommendable source to start with is the white paper as well as the Storj community chat.
White paper: https://storj.io/storj.pdf
Community chat: https://community.storj.io/

3.2 The company and its mission

Storj Labs Inc. was founded in 2015 and is based in Atlanta, Georgia (Bloomberg, 2017). Its mission is to tackle the problems of traditional, centralized cloud systems, by creating a decentralized cloud. Presently, the Storj network is essentially designed to be a solution for cloud *storage*. (Hoyes, 2014)
Figure 2 illustrates the differences between centralized and decentralized cloud systems and may reveal how the weaknesses of centralized cloud systems disappear or diminish in decentralized clouds. It also illustrates, that the possibility for everyone to be part of the cloud, leads to the situation that some of the machines are both server and client at the same time. This decentralized architecture is however not exactly matching the system architecture of the Storj network.

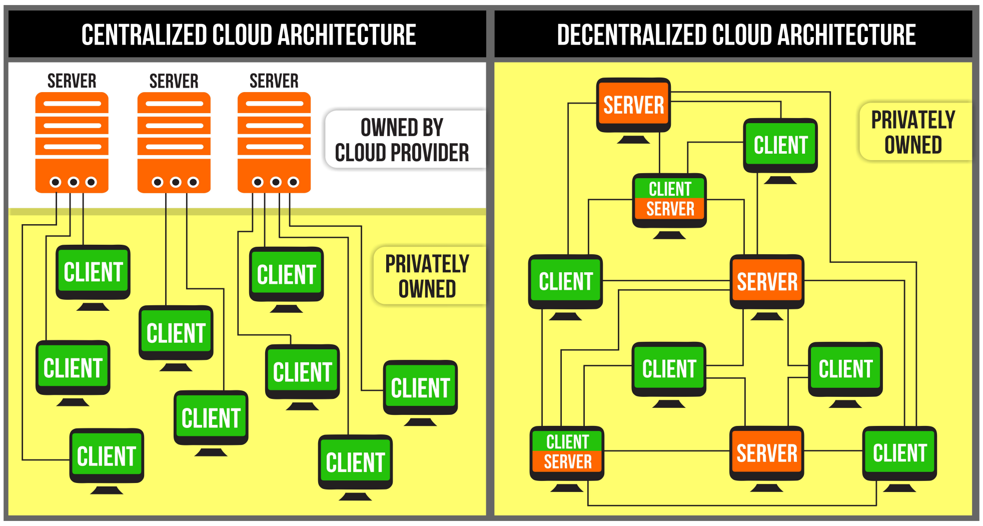


Figure 2: centralized vs decentralized cloud architectures

3.3 The data servers in the Storj cloud

Storj Labs' decentralized cloud allows everyone with the respective intent to contribute storage to the network. Consequently, every device on this planet meeting three basic requirements can be used to earn money by being a data server, respectively being a *farmer*. Table 1 lists these requirements.

Table 1: Requirements for devices to be able to rent storage to the Storj network

Requirement	Details
Being connected to the internet	There is no limitation to neither the internet connections bandwith nor the uptime of a device, but favourable parameters are rewarded by the network.
Having free disk space available	There is no limitation to the amount of disk space user wants to provide to the network.
Being able to run Storj Labs' application *Storj Share*	As of March 2017, *Storj Share* is available on Windows and on Linux computers (Storj Labs, 2017d).

Since neither the device's location, nor its ownership are subject to these requirements, the cloud can consequently be truly decentralized and distributed. (Rawle, 2016)

With this new possibility, Storj Labs creates a market for cloud storage on a lower level than it has existed so far; in the Storj network, entering the cloud storage market is no longer restricted to market participants who possess the means to maintain a complex data centre, but open to every individual with any connected, storage capable device. (Wilkinson, S., Boshevski, T., Brandoff, J., Prestwich, J., Hall, G., Gerbes, P., Hutchins, P., Pollard, C., 2016, p. 2)

3.4 Often addressed issues

Insights on how the Storj cloud technically works and how it addresses issues coming with a new architecture in cloud computing are beyond the scope of this thesis. However, there is a minimum of information that should be covered to convince people about the potential of a decentralized cloud. People who are new to this topic often take a critical attitude towards it, which is usually due to two particular issues. Hence the information about how Storj Labs addresses these two issues seems crucial for the understanding of the greater picture, which is why they are covered in this chapter.

"Why would I trust a random person to store my data on his/her computer? What if he or she spies on it?"

The key point to this issue is that the Storj Labs' cloud system is designed to be *trustless*. There is no trust needed between the renter and the farmer because the system itself provides trust based on various algorithms. The solution to the question of privacy as asked above, has the two following measures to it (Wilkinson et al., 2016, pp. 2-3).

Files are usually *sharded*. This means that none of the farmers receives any entire file, but instead just a chunk of it. The concept of sharding is further explained in chapter 7.4 . The more significant measure concerns the client-sided data encryption. All files that are uploaded to the cloud ought to be encrypted, before they are uploaded to any farmer. Therefore, they are unreadable to the farmers that are housing the data.

Both these measures are client-sided tasks and not mandatory for client implementations. Yet they are strongly recommended by Storj Labs, fully integrated in their libraries and thoroughly supported by the Storj network.

For Shawn Wilkinson, the CEO of Storj Labs, the privacy of the Storj cloud is not only as good as the one provided by traditional clouds, but even better. Concerning this matter he points out, that Storj Labs as a cloud storage provider "does not need to know anything about the data stored on the network in order to be able to operate." This gives the user additional privacy, as there is no third party who could access the uploaded data. (Wilkinson, Storj Master Plan, 2016)

"What happens to my files, if the person who stores my files turns his or her computer off? I would lose any access to it, wouldn't I?"

This problem of availability is addressed in the traditional way, by using redundancy. The decentralized cloud is very suitable for redundancy: its number of possible nodes to be used for creating redundancy is theoretically as big as the sum of all farmers. There are various concepts to be used for creating this redundancy. (Wilkinson et al., 2016, pp. 11-12) The main idea however stays the same for all of these concepts. It is to replicate the data of any farmer, as soon as the farmer went offline (by using the remaining data), and distribute it to a new farmer. This way, the network is behaving in a self-healing way, and the state of the redundancy stays the same over time. Theoretically this behaviour even increases the data availability, since the data is being placed onto farmers with higher uptime as the replacement is being repeated.

3.5 A storage provider for developers

Storj Labs' focus is on generating a cloud storage system, which can then be used by third party developers. Consequently, their focus is not directly on the end-users, which also means that they are not focused on building client software for different systems like for example Android devices, but merely about providing interfaces and libraries for third party developers, who want to use Storj as a cloud storage layer for their application. (Storj Labs, 2016a)

3.6 Development state

The Storj network is still under development. This concerns, for example, the payment system, which has not yet been implemented (Wilkinson et al., 2016, p. 9). Until its introduction traffic and storage remains free for the end-users.
This state of development involves recurring changes in the functioning of the network and is as such a considerable challenge for the development of any client.

3.7 Known weaknesses of the Storj cloud

After comparing the performance of the Storj cloud with the performances of traditional cloud providers, Holloh (2017) criticises it as being non-competitive. He bases his statements in two main factors. On one hand he mentions that the network would still be too small and therefore not at its full potential yet. On the other hand, he speaks about an issue coming with Storjs sharding and encryption practice. It prevents the network from making delta uploads. These are upload processes, in which a client just transfers the data that actually changed inside of a file, instead of uploading the whole file. The same principle is equally valid for downloads. (Holloh, 2017, p. 69)

3.8 Storj cloud clients

As the Storj project is very new and Storj Labs itself does not focus on cloud clients for end-users, there are right now only three different clients publicly available. Table 2 gives a concise overview about these clients, and states the responsible party behind each of them. To attentive readers, it may furthermore implicitly reveal, that there is still a lot of work to be done for end-users to have a valuable experience with the Storj cloud.

Table 2: Publicly available Storj cloud clients

Node.js client	**Storj web interface**	**Third party Java client**
Storj Labs	Storj Labs	Stephen Nutbrown
The Node.js client does not have any GUI and can therefore only be interacted with as a command-line interface.	Storj Labs' web interface can only be used to browse and manage buckets, but not to actually up- / download data to / from the network.	The java client is currently having errors due to changes that were made on the Storj network, that have not yet been addressed by him.
https://docs.storj.io/docs/getting-started	https://app.storj.io/	https://github.com/NutterzUK/Storj-Java

Looking at the available clients, it becomes clear that there is currently no possibility for mobile device users to access the Storj network for transferring data at all. Moreover, there is not only a lack of client implementations, but there is currently no Storj library available, which could be used for developing apps for mobile devices.

3.9 Conclusion

The weaknesses of common cloud systems, caused by their system design of a centralized architecture, are evidently significant. Storj Labs on the other hand has various measures on how they approach these issues, the key concept being the decentralized architecture of their cloud, which itself entails a new set of issues.

The biggest obstacles of developing a Storj cloud client are the lack of documentation about the Storj cloud, a weak developer support, the state of the Storj cloud – being a system that's still under development – and the fact that there are no similar projects so far that could be used as a reference implementation.

4 STORJ LABS CLOUD ARCHITECTURE

4.1 Complex client-sided tasks and the Bridge

In the Storj network the cloud clients are responsible for many tasks like encrypting and sharding files, finding farmers and creating contracts with them, issuing audits and verifying them (to make sure that the farmers are still online and still have the data the client stored on them), and paying the farmers. All these requirements for cloud client software result in a need for complex client libraries and comprehensive clients. In addition to the complexity, there is another issue coming with these requirements; some of the given tasks demand high uptime from the clients, which is a serious drawback for many customers, as the devices they connect to the cloud might not mean to be continuously turned on and connected. (Wilkinson et al., 2016, p. 17)
In order to lighten the burden on the clients, Storj Labs has introduced a supporting component to the network: The so-called Bridge. The Bridge is designed to be run on a centralized server, to take responsibility over some of the client-sided tasks, while delivering theoretical full uptime. (Wilkinson et al., 2016, p. 18)
The software of the Bridge is open source, just like any other software from Storj Labs (Storj Labs, 2017a), and may therefore be setup on any server of an individual or an enterprise. There is however also a Bridge hosted by Storj Labs themselves, which is open for any public use. This Bridge is accessible over *https://api.storj.io* and is also the standard setup, as currently used by the majority of the clients accessing the cloud. Figure 3 illustrates the respective network model as it is used in this thesis.

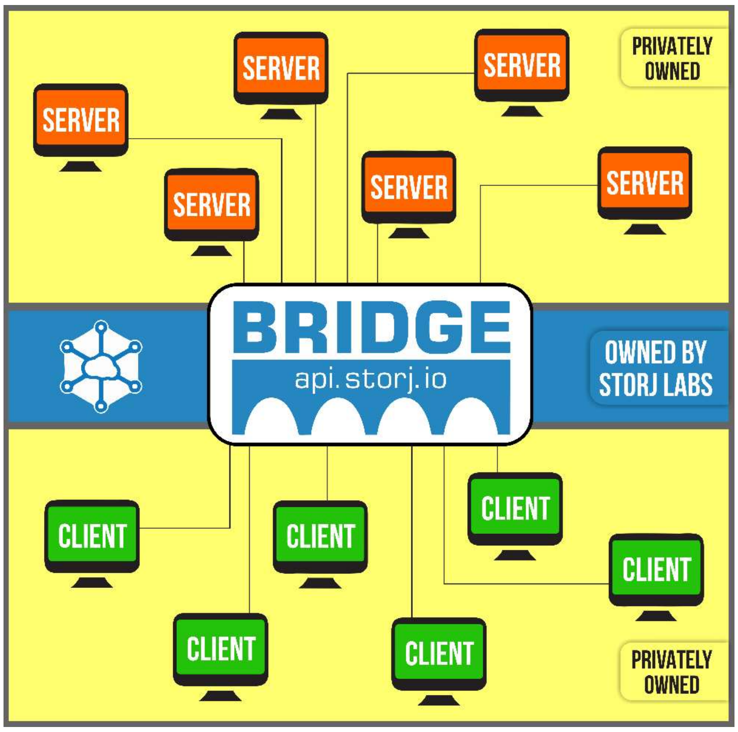


Figure 3: the network model as used for this thesis

Please Note: For the simplicity of the model shown in figure 3, all computers involved are either servers or clients. In the real world, it is also perfectly possible for devices to be both client and server at the same time.

The Bridge software is continuously being updated and expanded by Storj Labs. It is accessible from all HTTP capable devices, as it is a RESTful web service providing data over HTTP calls. This means that it is accessible from all common and modern operating systems which quite obviously also includes Android devices.

The source code of the Bridge service is available on Github:
https://github.com/Storj/bridge

4.2 The centralization problem of the Bridge architecture

While the solution with the centralized Bridge is indeed convenient for third party developers, it is rather obvious that it also brings back the problems of centralized clouds inside of Storj Labs´ cloud solution. It is however not the same degree of delegated trust towards the cloud hoster, as it is found in the clouds of traditional cloud providers. The Bridge does for example neither store data (but just metadata about the data) nor ever receive any unencrypted data (with an exception for public buckets) (Wilkinson et al., 2016, p. 18).
Furthermore, the Storj whitepaper states that "it is possible to envision Bridge upgrades that allow for different levels of delegated trust" (Wilkinson et al., 2016, p. 18).
Another measure for Bridge clients to minimize the level of required trust to be delegated to any Bridge, is to use multiple Bridge servers and hence split the workload between them (Wilkinson et al., 2016, p. 18).
Furthermore, in March 2017, Wilkinson announced, that Storj Labs would aim to strengthen the availability of the network, by giving the opportunity for clients to store copies of network locations and authorization keys. With this solution, clients could still access their files for a certain time, even if the Bridge would completely fail. (Wilkinson, 2017)

4.3 The architecture adopted in this thesis

For this thesis, the most common way of connecting clients to the farmers is chosen, i.e. using the Bridge provided by Storj Labs. This decision is taken because it allows giving insights on how to connect to Storj from Android devices in the most common way, but also because at this early stage of Storj, this is simply the most stable and most tested way to access the Storj network. Table 3 shows how the workload is divided between the Android app and the Bridge in such architecture. (Wilkinson et al., 2016, p. 18).

Table 3: Responsibilities of the clients vs. the Bridge

Responsibility of the client	**Responsibility of the Bridge**
Sharding files	Contact negotiation with farmers
Encrypting and decrypting files	Providing (micro-)payments to farmers
Managing file encryption keys	Issuing and verifying data audits
Generating data audits	Managing file state by making sure every shard is always available.

In addition to the mentioned tasks, the Bridge offers a one-level hierarchy composed of so-called Buckets, as well as a user management backend. A library might as well provide these functionalities. Both functionalities are not related with the farmers in any way, but just utilities that the Bridge provides to the clients.

5 CURRENT STORJ CLOUD CLIENT LIBRARIES

There are a few libraries that are already developed or still in progress, which may be used for creating client software, or serve as a reference when implementing one's own library. As to this point there are only very few, they might not be an option for every system, as this is for example the case for all modern mobile device operating systems. This chapter demonstrates these different developments as well as their development status.

5.1 Node.js library

There is a Node.js library from Storj Labs, which is the first library there was and is also the one used in Storj Labs' client command line interface. This library is the most extensive there is and it is continuously being updated and expanded. Being a Node.js project, it is based on a JavaScript runtime environment which is probably also its greatest weakness, as Node.js does currently not support any operating system used on mobile devices (Node.js, 2017). A remarkable advantage of this library, is that due to the broad support of JavaScript inside of all common browsers, it may be used inside of browsers too. (Storj Labs, 2017e)

The library and a respective installation guide are available on Github: https://github.com/Storj/storj.js

5.2 Libraries for C, Python and Java

Storj Labs affirms that "Implementations in C, Python, and Java are in progress" (Wilkinson et al., 2016, p. 19).
At the time of this thesis' realisation however, only a beta version of the C library had been released. There was not a lot to be seen from the two other libraries at the time. Due to this fact, these libraries are not covered any further in this thesis, even though especially the Java library could have been very helpful for the development of the reference implementation.
The beta version of the C library is available on Github:
https://github.com/Storj/libstorj

5.3 Third-party Java library

There is a third-party Java library, written by Stephen Nutbrown, which unfortunately is not up to date. Uploads and downloads do not work anymore, as Storj Labs switched from using websockets to HTTP after this library was implemented. Nevertheless, the library is a great reference implementation, since many ideas and even a lot of code, may be adopted to an own implementation, as the library comprises an outstanding code quality.

This library was a great help for developing the reference library, even though there are various differences between the Java Virtual Machine which the library was programmed for, and the Android Runtime which the reference library is running on. Nevertheless, a significant amount of code could be reused in the reference implementation and a substantial part of it was indeed reused.

The source code of the library is accessible on Github:
https://github.com/NutterzUK/Storj-Java

6 METHODOLOGY

6.1 Thesis

The thesis is carried out as part of a bachelor's degree programme in Business IT, within the framework of the Double Degree Program which is based on a collaboration between the Häme University of Applied Sciences (HAMK) and the Bern University of Applied Sciences (BFH).

This thesis follows the guidelines of a practice-based thesis as described by HAMK. Hence it consists of a practice-based part being the reference implementation, and a documentation to it, which is provided by this thesis. (HAMK, 2017, pp. 5-6)

6.2 Purpose

Initially the idea of this thesis was to create an Android app which would allow to transfer data from and to the Storj cloud for all devices running on Android. With growing experience about Android and the Storj cloud, it became obvious, that it was not as easy to develop this app as it appeared at the beginning. This was mainly due to the reason, that Storj Labs does not provide any library for any mobile devices operating system (yet). This need for an Android library, led to the development of the reference implementation.
In the process of this development, the author experienced how complex and interesting this distributed system was, but also how difficult it was to find any documentation, information or help. For this reason, there seemed to be a more valuable contribution to make than just developing an app. The purpose of this thesis is therefore to investigate and to collect information, in order to provide it in a way that it would be easier for future developers to find documentation, whenever their idea was to integrate Storj into an environment, which is not supported by Storj Labs.

6.3 Knowledge base

The author disposes of a 10-year-long experience in software development, which applies particularly for Java development. As Android was the chosen development environment, the author’s experience was fundamental to the creation of this thesis, since Android software development bases on an adapted and limited Java-implementation. Furthermore, Android development is part of the authors study programme at HAMK.

The Storj-related knowledge is based on many different sources. An important part of the sources are all kind of articles and blog entries that are available on the internet, yet do need quite a bit of investigation to be discovered. Regarding these sources, the Storj whitepaper must be particularly highlighted. Another important source is the Storj community chat. This chat is open to anyone who wants to discuss Storj-related topics. Finally, a considerable amount of insights is based on the analysis of source code from the Storj repositories on Github. This is possible since Storj Labs publishes its code as open source.

Many of the technologies used by Storj Labs are inspired by the Bitcoin / Blockchain / Cryptocurrency sector. Hence investigations in this field deliver further insights, and especially deeper understanding about the functionality of some of the technologies.

6.4 Development environment

The choice of Android as a development environment for developing a Storj library is mainly based on the author's greater intention to build a Storj cloud client for Android devices. As a result of this purpose, there was a premature version of both a library and an Android client existing already prior to the realization of this thesis. This version had been developed by the author in collaboration with his fellow student Juho Puoliväli as a school project during their Business IT studies at HAMK. The project served as a base for further development. Table 4 shows the characteristics of the projects version at the start of the thesis' realization. The existence of this premature version together with the already acquired Android development know-how further influenced the choice of Android as a target environment.

Table 4: characteristics of the library development prior to the thesis

Authentication	**Only basic authentication implemented**
Up- and downloads	Only partly implemented and due to changes of used technologies inside of the Storj cloud not working anymore. Including potential for various improvements in many different areas
Encryption	Not meeting modern security standards and not compatible with libraries provided by Storj Labs
Buckets	Fully implemented, except for the feature *public buckets*
General	Many aspects of the app in a pre-alpha state

The reference implementation is developed in the Android Studio, using a Motorola Nexus 6 with Android 7.0 (Nougat) to execute the software on.

6.5 Validating the acquired information

As the investigation field of this thesis is rather complex, and information is not only difficult to obtain, and furthermore sometimes controversial, sometimes non-existent and sometimes even wrong, the error rate of the elaborated information for this thesis is rather high. Thus, it is important to prove the obtained information. For this purpose, it is turned into code whenever possible. In such way, the functionalities can be tested against the Storj network, which proves them right or wrong by simply being functional respectively failing. This is the main purpose of the reference implementation. Trial and error is a substantial part of this thesis' investigation process.

As the complexity of the procedures discussed in this thesis is also an issue for the reader, these procedures are broken down to small steps and accompanied by sample data. This is especially helpful when it comes to hashing and encryption procedures, as from a human point of view the outcome is just structureless random data.
As the reader is given sample input data together with the corresponding output data, developing an own solution should be significantly eased. For such case, the reader may verify each single hashing and encryption function, whether the right algorithm with the correct settings is being used. As a result of this, the code presented in this thesis is not identical to the code of the reference implementation but instead edited in a way, that is helpful and easy to understand for the reader.

All code is represented in Java syntax, in a form that it would be executable when running in an Android environment.

7 CLIENT LIBRARY DEVELOPMENT

7.1 Forms of Authentication

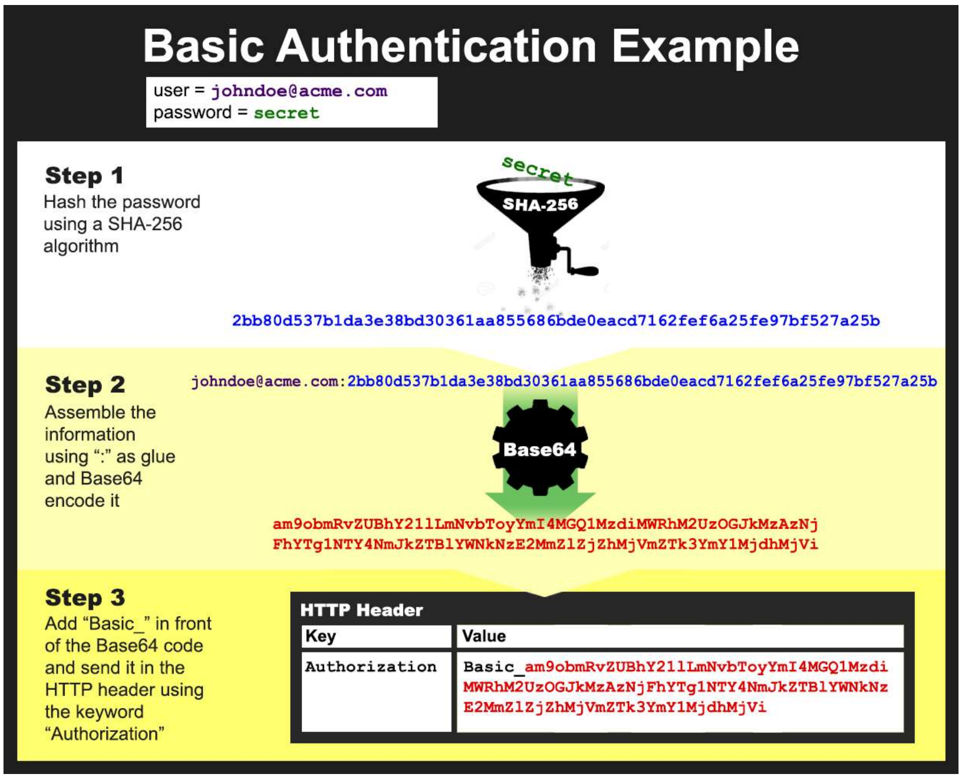
To be able to access Storj Labs' Bridge, an account is needed. It can be created on https://app.storj.io but it is also possible, to create new accounts using the Bridge itself by sending a POST request to https://api.storj.io/users. In both cases a user would have to confirm the email address before being able to use the account. (Storj Labs, 2016e)

Once a user account is available, there are two different ways to authenticate Bridge-users on the Bridge. When accessing the farmers, there is even a third authentication type used. This chapter gives further insights about the used techniques. (Storj Labs, 2017b)

7.1.1 Basic auth

Basic authentication is the term for the traditional way of authentication – using a user account and a password.

The user account is transmitted as plaintext, while the password is not directly being sent to the Bridge, but instead just a SHA-256 hash of it.
(Storj Labs, 2017b)
These two variables are then assembled together and transformed to a Base64 string. The resulting string needs to be sent in the HTTP header, using the tag "Authorization". It is crucial, that this whole procedure is realised using text that is based on the UTF-8-character encoding. Figure 4 shows a detailed visualization of this procedure using the example-user "*johndoe@acme.com*" with the password *"secret".*



Basic Authentication Example

user = johndoe@acme.com
password = secret

Step 1
Hash the password using a SHA-256 algorithm

2bb80d537b1da3e38bd30361aa855686bde0eacd7162fef6a25fe97bf527a25b

Step 2
Assemble the information using ":" as glue and Base64 encode it

johndoe@acme.com:2bb80d537b1da3e38bd30361aa855686bde0eacd7162fef6a25fe97bf527a25b

am9obmRvZUBhY21lLmNvbToyYmI4MGQ1MzdiMWRhM2UzOGJkMzAzNj
FhYTg1NTY4NmJkZTBlYWNkNzE2MmZlZjZhMjVmZTk3YmY1MjdhMjVi

Step 3
Add "Basic_" in front of the Base64 code and send it in the HTTP header using the keyword "Authorization"

HTTP Header

Key	Value
Authorization	Basic_am9obmRvZUBhY21lLmNvbToyYmI4MGQ1Mzdi MWRhM2UzOGJkMzAzNjFhYTg1NTY4NmJkZTBlYWNkNz E2MmZlZjZhMjVmZTk3YmY1MjdhMjVi

Figure 4: basic authentication illustration using an example

Code snippet 1 shows the preparation for the basic authentication data in the form it should be added to the HTTP header and using the Hashing class from `com.google.common.hash`.

```
// sample data
String uname = "john.doe@acme.com";
String pass = "secret";

// actual procedure
String hash = Hashing.sha256().hashString(pass,
                         StandardCharsets.UTF_8).toString();
String concat = base64Encode(uname + ":" + sha256(pass));
String authHeader = "Basic " + concat;
headerParams = new HashMap<String, String>();
headerParams.put("Authorization", authHeader);
```

Code snippet 1: basic auth for HTTP calls as implemented in the reference implementation

7.1.2 Signature based authentication

The Bridge supports an authentication, that uses signatures which are based on public-key cryptography. Instead of sending a password with every request, the client signs each request and sends the resulting digital signature together with the request. A fundamental requirement for this method is that every single signature that is sent is unique. This uniqueness is achieved by adding a *nonce (=* ***n****umber used* ***once****)* to the data that is signed. What data the nonce contains is irrelevant; the nonce's only requirement is, that it is unique for every request.

```
// generating a nonce
String nonce = UUID.randomUUID().toString();
```

Code snippet 2: automated nonce generation in Java

Storj's signature solution is based *on ECDSA (Elliptic Curve Digital Signature Algorithm)*. As the name suggests, ECDSA is an algorithm, which is based on elliptic curves. While there are many different defined elliptic curves, Storj uses the curve called *Secp256k1*. This specific curve is most known for being used in Bitcoin. (Antonopoulus, 2014, p. 66)

The basis for this method is a key pair generated by the client. This key pair contains a private and a public key. The private key is used to generate the signature and must be kept secret. It must never be transmitted to the Storj network in any way, as this would put a user's security at enormous risk. The public key on the other hand is used for verifying the signature and must therefore be uploaded to the Bridge. (D. Johnson, A. Menezes, S. Vanstone, 2016, pp. 3, 24)

```
public static KeyPair generateKeyPair() throws Exception {
        // in Android, this is actually the spongy castle
        // provider, even though it's called bouncy castle
        Security.insertProviderAt(new BouncyCastleProvider(, 1);

        // Storj signatures use the ellipic curve called secp256k1
        ECGenParameterSpec genSpec;
        genSpec = new ECGenParameterSpec("secp256k1");

        KeyPairGenerator gen;
        // SC stands for SpongyCastle
        gen = KeyPairGenerator.getInstance("ECDSA", "SC");
        gen.initialize(genSpec, new SecureRandom());

        return gen.generateKeyPair();
}
```

Code snippet 3: key pair generation method from the reference implementation

From a technical perspective, the public key is a point on the elliptic curve. Therefore, it must get encoded in order to get into a form that it may be registered on the Bridge. The encoded public key must then be of the following form to be accepted by the Bridge: it might be either compressed (chosen approach of the Storj Node.js library) or uncompressed and must be transformed to a hexadecimal String.

Table 5 shows examples of the two forms that are accepted by the Bridge.

Table 5: Examples of public keys readable for the Bridge

Compression	Key Example	Size
compressed	`0366da51dd4fcd758eedabb1a79ba9c885c657ce3fa13c2e06e7a3e203248c8735`	66 characters (33 Bytes)

uncompressed	`04d237a70804daddcfcaf309925aeaa105d8b7a2121094d8051a8739e6703282e09c8972d5e19ec18434cb97f08d8e0dced7ad9944f1382a76d2c967d1b1bc4ef2`	130 characters (65 Bytes)

Code snippet 4 shows the conversion of the public key into a hexadecimal encoding as a string, which may then be sent to the Bridge as a string.

```
public static String convertPubKeyToBridgeFormat(PublicKey pk){
    BCECPublicKey publicKeySC = (BCECPublicKey) pk;
    byte[] pubKeyBinary = publicKeySC.getQ().getEncoded(true);
    return bytesToHex(pubKeyBinary);
}
```

Code snippet 4: converting the public key to a hexadecimal encoding

Once the public key is registered on the Bridge, the client may use the signature based authentication, using the private key to sign its requests. For this procedure of signing, the algorithm *SHA256 with ECDSA* must be used. This means that the data to be signed is first hashed using the algorithm SHA256 and then the resulting hash is signed using the algorithm ECDSA.

What data will be signed depends on the request. There are two patterns that are followed, depending on whether a request submits data, or just requests data. In either case, a nonce must be part of the data to be signed, as this makes the signature unique. Table 6 further explains the two patterns.

Table 6: Data to be signed for Requests that transmit data vs Requests that do not

Request NOT transmitting data	Request transmitting data

The data to be signed consists of the HTTP-method of the request, the path-part of the URL, starting with a slash but NOT ending with slash, and the nonce. These three parts must be separated by newlines (two in total).
Example
`GET`
`/buckets`
`__nonce=800a-4eb2-9afc`
Additionally, the nonce must be added to the URL of the request as a GET-parameter.
	The data to be signed consists of the HTTP-method of the request, the path-part of the URL, starting with a slash but NOT ending with slash, as well as all the data submitted to the server in JSON format. This does also contain the nonce.
	Example
	`POST`
	`/buckets`
	`{"name":"test","user":"joh`
	`n.doe@acme.com","status":"`
	`Active","__nonce":"800a-`
	`4eb2-9afc"}`

Code snippet 5 shows the function of the reference implementation which signs this data with the given private key.

```
public static String signData(PrivateKey privateKey, String data)
                                            throws Exception {
    Signature sig = Signature.getInstance("SHA256withECDSA");
    sig.initSign(privateKey);
    sig.update(data.getBytes());
    byte[] signature = sig.sign();

    return bytesToHex(signature);
}
```

Code snippet 5: signing data with SHA-256 and ECDSA

The created signature as well as the public key then need to be added to the header information of the HTTP-request, using the keyword *x-signature* for the signature respectively *x-pubkey* for the public key.

7.1.3 Tokens

A token based authentication is available only for the direct communication between renter (client) and farmer in which use case it is furthermore the only authentication method available (Wilkinson et al., 2016, p. 17). The tokens must be requested by the Bridge. These token requests trigger the Bridge to create contracts with the farmer, which are the basis for the interaction between the farmer and the renter. After successful contracting the Bridge then returns a token for each token request, which can then be used by the client, to either upload or download data to respectively from a farmer. Depending on whether a client wants to upload or download data, it needs to declare a different *operation* in the request: “PUSH” for uploading, “PULL” for downloading.

Code snippet 6 shows an example for a token request of the type PUSH, as it would be sent to the Bridge as a HTTP POST call.

```
// b = sample of a bucket id
String b = "a4b3a6872bfea510bbd995a1";
String url = "https://api.storj.io/buckets/" + b + "/tokens";
Map<String, Object> postBody = new HashMap<String, Object>();
postBody.put("operation", "PUSH");
String postBodyJson = gson.toJson(postBody);
```

Code snippet 6: An example PUSH-token request

As of February 2017, a Storj Improvement Proposal has been submitted, which could alter the way renters authenticate themselves against farmers. The proposal involves the possibility to add public keys to contracts, which would enable a signature based solution just as the one for the communication between clients and the Bridge. (Fuller, 2016)

7.1.4 Conclusion

While the only possibility to authenticate a client against its farmers is to use tokens, its authentication against the Bridge leaves a choice of two different methods, *basic auth* and *signature based auth*.
There are various advantages respectively disadvantages between these two methods. However, the most significant one might be the following security issue coming with basic auth: whoever is able to spy on the credentials a specific client uses for its authentication, is able to steal them and do whatever transaction he/she wants to do, using the identity of its victim. Even though the SHA-256 hash protects the user from anybody figuring out his / her password, an authentication is perfectly possible without knowing the cleartext password, but only knowing its SHA-256 hash.
The signature based authentication in return is the safer method, since the only way to steal the credentials of a user would be to steal his / her private key – which is never transmitted to the network. The disadvantage of the signature based authentication is that it seems to be more difficult to implement.

A reasonable usage of the two methods is to use basic auth to enable signature based auth, which involves uploading a public key to the Bridge. Once this initial step has taken place, the use of signature based auth is recommendable. Following this procedure, it is not necessary, to store any user's password on any device, but in each case just the created key pair.

7.2 **Data transfer procedures**

Transferring data from or to the Storj cloud using the Bridge involves a sequence of different steps on behalf of the different network participants. These working steps vary for up- respectively downloads. This chapter is further explaining these required steps. (Wilkinson et al., 2016, p. 19)

7.2.1 Uncertainties about the upload procedure

There is a step based instruction on how the file upload works in the Storj whitepaper (https://storj.io/storj.pdf). It does however slightly vary from the instruction given in this thesis. On one hand the description here is a bit more extensive, on the other hand some steps differ in their order. This is due to experiences, the author made while developing the reference implementation for the thesis.
The main inconsistency has emerged, because the issuing of audits by the Bridge has not entirely been implemented so far. The audits are the base for the payment system, which is just being developed while this thesis is written (Wilkinson et al., 2016, p. 8).
It is yet to be found out, by which event the first audit issuance of the Bridge is triggered and how the Bridge reports the answer to the client. The latter point is a question about what response the audit verification result is added to, which depends on the first point: the question about which client request triggers the audit issuance. In the whitepapers instruction, it is explained as triggered by the client transferring the audit information. However, the thesis' reference implementation sends this information already before the data has actually been uploaded to the farmer. The bridge would therefore neither know, when the upload has finished, nor have any request available to answer to.
When researching this issue further, a Storj-developer told the author, that the issuing of audits was not activated at the time and that the issuance of the *exchange reports* would trigger the audits once they are fully implemented (littleskunk, 2017). This is the information this step by step instruction is based on. In this scenario, the audit results would be communicated in the response to the exchange reports issuing request of the client.
As a conclusion of this, the author recommends any developer referring to this instruction to be mindful of this situation and to be attentive to the further implementation of the Bridge, which will eventually lead to a clearer notion of this sub process. (Wilkinson et al., 2016, p. 19)

7.2.2 Upload procedure

The following list shows the sequence of steps needed to upload a file to the Storj network. (Wilkinson et al., 2016, p. 19)
Figure 5 shows a graphical representation of these steps.

1) The client prepares data to be uploaded. This step results in one or multiple encrypted shards.

2) The client informs the Bridge, that it wants to upload data. For this purpose, it creates a frame on the Bridge. A frame contains metadata used for the upload of a file.

3) The client adds shards to the frame. More precisely, these are not the actual shards, but only metadata about them, like for example their file sizes. They also contain the information needed for issuing audits.

4) Each shard [metadata] that is added to the frame, triggers the Bridge to generate a contract with an available farmer.

5) As a response to the shard adding request (step 3), the Bridge responds with contact information about the farmer it generated a contract with. This contains information such as the farmer's address (IP or domain), its port, the farmer's id (node id) inside the network, or the protocol version it uses. In addition to this, the Bridge also sends an authentication token for the client to authenticate against the farmer.

6) The client directly uploads its shards data to the farmer, using the information and token it received in step 5. This is done over HTTP.

7) The client uploads *exchange reports* for each shard-upload whether they were successful or unsuccessful. These reports contain data about the uploads which are then used by the bridge to maintain its internal, farmer-related reputation system. For the client itself, this step is also important because it will probably [see chapter 7.2.1] be implemented to serve as a trigger for the Bridge to start step 8.

8) The Bridge proves whether the data was transferred correctly by issuing an audit for the farmer and verifying the response. At the time this thesis is written, it is not yet clear, how the Bridge will transmit the result of the audit to the client (as an answer to which request). If the result were negative however, it seems evident that the shard would have to be transferred again. After having verified the upload, the Bridge is ready to start the mirroring process.

9) The client uploads metadata about the files to the bridge, and associates it to the frame that was used for uploading the shards. This metadata contains information about the file, namely its filename, file size and mimetype, as well as organizational information for the bridge like an id and the id of the bucket and the frame it belongs to.

10) As long as no other setting is implemented, the Bridge will assume to be responsible for the tasks needed to be completed during the time the shard is online. Concerning farmer-side communication, this involves issuing audits, paying farmers, and managing file state. Concerning the client-side communication, it involves exposing the file metadata for the client.

Figure 5 shows an illustration of these steps using an example, in which there is a file uploaded, which is split up into 3 shards in step 1. Therefore, the data is uploaded to 3 farmers. In the given example one farmer would have an error, which would lead to the Bridge searching for another farmer. For an increased simplicity of the model, the mirroring process is not shown. As it is only relevant for the communication between the Bridge and the Farmer, it is of little interest for a client library developer anyways.

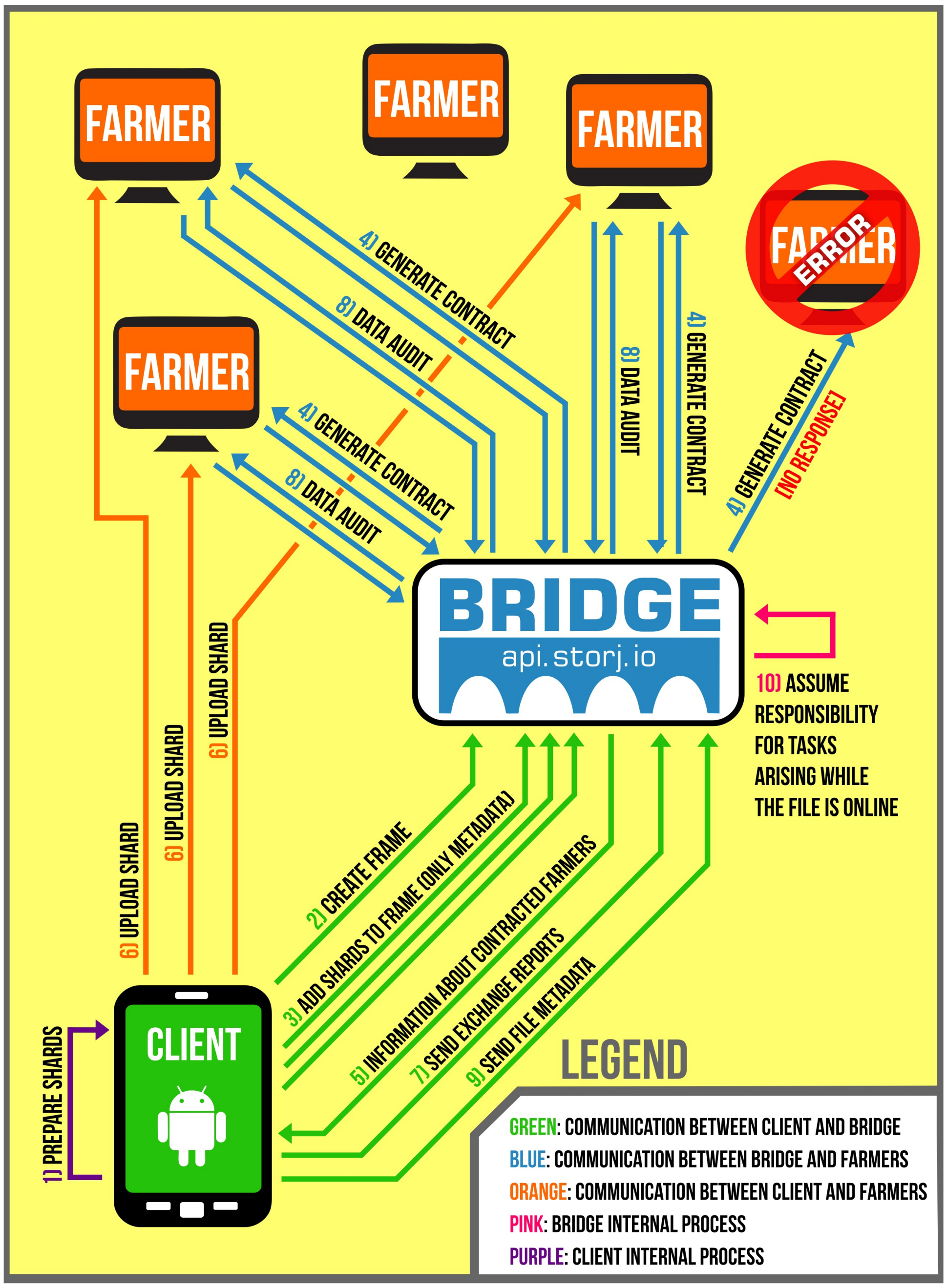

FARMER
FARMER
FARMER
FARMER
FARMER
ERROR
BRIDGE
api.storj.io
CLIENT
1) PREPARE SHARDS
2) CREATE FRAME
3) ADD SHARDS TO FRAME (ONLY METADATA)
4) GENERATE CONTRACT
4) GENERATE CONTRACT [NO RESPONSE]
5) INFORMATION ABOUT CONTRACTED FARMERS
6) UPLOAD SHARD
7) SEND EXCHANGE REPORTS
8) DATA AUDIT
9) SEND FILE METADATA
10) ASSUME RESPONSIBILITY FOR TASKS ARISING WHILE THE FILE IS ONLINE
LEGEND
GREEN: COMMUNICATION BETWEEN CLIENT AND BRIDGE
BLUE: COMMUNICATION BETWEEN BRIDGE AND FARMERS
ORANGE: COMMUNICATION BETWEEN CLIENT AND FARMERS
PINK: BRIDGE INTERNAL PROCESS
PURPLE: CLIENT INTERNAL PROCESS

Figure 5: illustration of the upload process

7.2.3 Download procedure

The following list shows the sequence of steps needed to download a file from the Storj network. (Wilkinson et al., 2016, p. 19)
Figure 6 shows a graphical representation of these steps from the clients point of view.

1) The client requests a file from the Bridge, using the files id.

2) The Bridge sends the client information about the farmers, where it can download the shards that make up the file. This contains among other information the farmers IP-address or its domain name, and a token which the client uses to authenticate against the farmer in step 3.

3) The client downloads the shards needed from the farmers over HTTP, using the farmer information and tokens received in step 2.

4) The client uploads *exchange reports* for each shard-download that was *unsuccessful*. These reports contain data about the downloads which are then used by the bridge to maintain its internal, farmer-related reputation system.

The author expects a future release of the Bridge to respond to the exchange report with information about a new farmer for the client to download the shard from. At the time this thesis is written however, there is no such response implemented yet.

5) The client converts the encrypted shards into the unencrypted file.

Figure 6 illustrates these steps using the following scenario:
The client wants to download a file, that consists of three shards. One of the farmers however has an error and does therefore not deliver the shard. Therefore, the client sends a negative exchange report to the Bridge to which the Bridge responds by sending the access data to another farmer. The client then starts a new download of the shard from the new farmer.

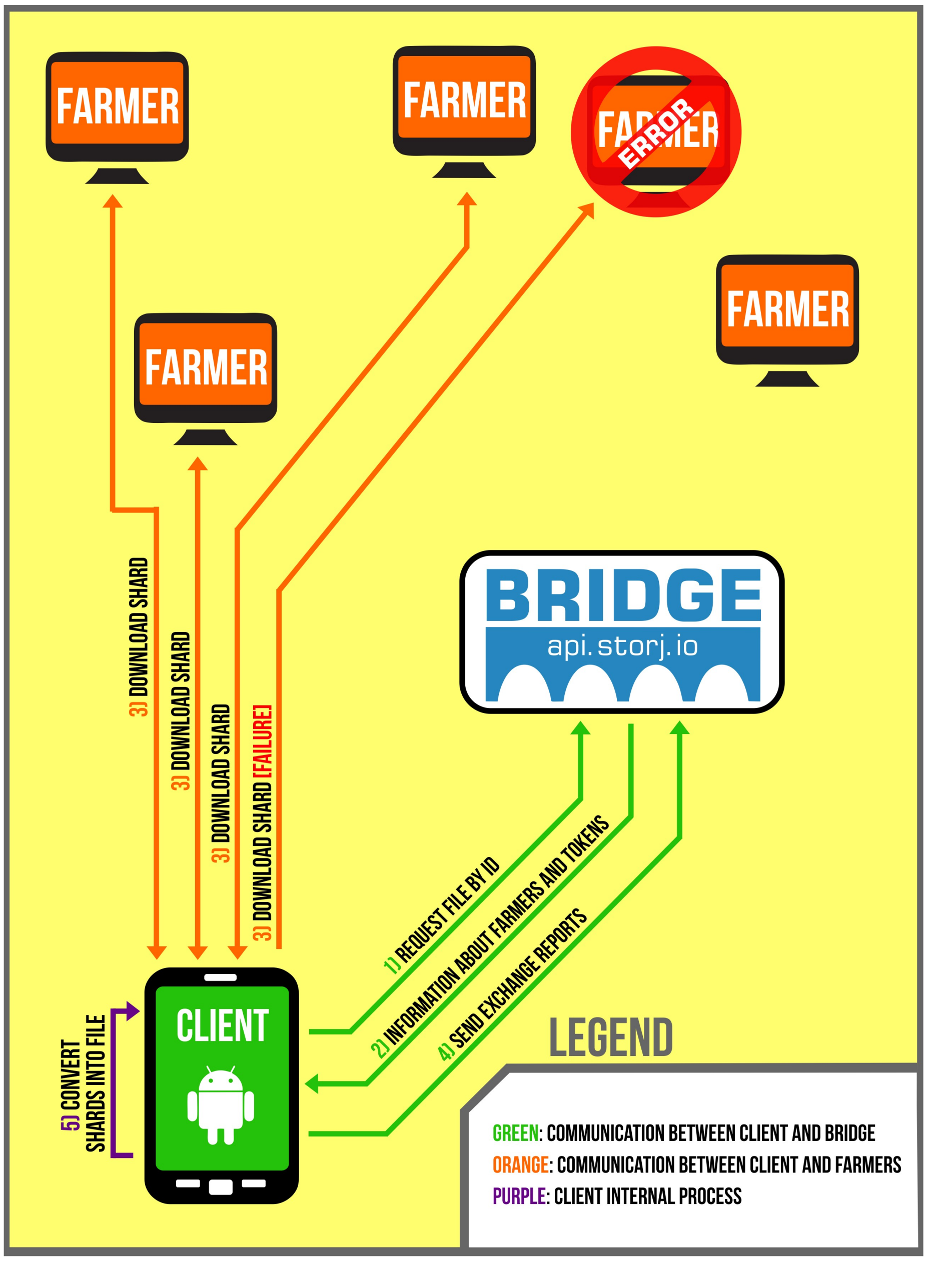

FARMER
FARMER
FARMER
ERROR
FARMER
FARMER
BRIDGE
api.storj.io
3) DOWNLOAD SHARD
3) DOWNLOAD SHARD
3) DOWNLOAD SHARD
3) DOWNLOAD SHARD [FAILURE]
1) REQUEST FILE BY ID
2) INFORMATION ABOUT FARMERS AND TOKENS
4) SEND EXCHANGE REPORTS
CLIENT
5) CONVERT SHARDS INTO FILE
LEGEND
GREEN: COMMUNICATION BETWEEN CLIENT AND BRIDGE
ORANGE: COMMUNICATION BETWEEN CLIENT AND FARMERS
PURPLE: CLIENT INTERNAL PROCESS

Figure 6: illustration of the download process

7.2.4 Shard transfers

The shard upload and download to respectively from a farmer is done over HTTP. The following cohesive code snippets show the implementation of the shard upload from the reference implementation.

Code snippet 7 shows the variables needed for the HTTP call. All of these variables are provided by the Bridge, as a response to the token request.

```
URL url = new URL(strUrl);
HttpURLConnection conn = (HttpURLConnection)url.openConnection();

conn.setRequestMethod("POST"); //"POST" for upl. | "GET" for downl.
conn.setDoOutput(true); // true for POST requests, false for GET
conn.addRequestProperty("x-storj-node-id", nodeId);
conn.setRequestProperty("Content-Type","application/octet-stream");
conn.setConnectTimeout(240000); // optimal parameter unknown
```

Code snippet 7: configuring the connection

Code snippet 9 shows the settings of the connection to be set before starting the HTTP call. The method is always POST for uploading and GET for downloading.

```
// sample data [usually delivered by the Bridge]
String farmerIp = "123.123.123.123";
int farmerPort = 9876;
String token = "1591756dd4a1847997c4cf9a8aba9040f28cd2f3";
String shardHash = "41007cfb958ebe453c4f8e53f85cb9f007051efa";
String nodeId = "6d825b7dc6ce3dc0aa0225cccfd66b49fcf9e032";

String farmerAddress = "http://" + farmerIp + ":" + farmerPort;
String params = "/shards/" + shardHash + "?token=" + token;
String strUrl = farmerAddress + params;
```

Code snippet 9: sample variables and the composing the URL to be called

Code snippet 8 shows how the data finally is sent to the farmer and subsequently the connection is closed.

```
// send request
OutputStream os = conn.getOutputStream();
DataOutputStream wr = new DataOutputStream (os);
File shardFile = new File(shard.getPath());
// send shard data as binary
wr.write(readAllBytes(shard.getPath()));

// close connection
wr.flush();
wr.close();
conn.disconnect();
```

Code snippet 8: transferring the data and closing the connection

7.3 Buckets

Buckets on Storj are like folders in a file system, with the difference that they only allow one single level of hierarchy. In other words, it is not possible to create any bucket inside of another bucket.

“A bucket is just a logical grouping of files that we can assign permissions and limits to.” (Storj Labs, 2016b)

Buckets can be created, read, updated and deleted (CRUD) over the Bridge’s RESTful web service on api.storj.io .

7.3.1 Shared buckets

There is a possibility to add public keys to buckets. This is part of a feature to share buckets. However, this feature is still not implemented as of February 2017.
The author assumes that the public keys to be stored in these buckets, would be the public keys of the keypair, a respective “invited” user would use for the authentication towards the bridge.

7.3.2 Public buckets

In October 2016, Storj Labs introduced the feature *public bucket* (Storj Labs, 2016c). This feature is limited to client implementations that use the deterministic encryption key derivation from Storj Labs´ libraries (as explained in chapter 7.6), or a similar implementation. The crucial point is that any client given the bucket key, must be able to encrypt and decrypt all the data inside of a bucket by either using the bucket key directly, or deriving keys from the bucket key (recommended).

From a technical point of view, a public bucket is nearly the same as a normal bucket, with just the difference that it contains two more elements as metadata: a bucket-related encryption key and a list of codes for the access level. Being part of the bucket, this information is consequently transmitted to the Bridge.

The Bridge may provide the uploaded key to any client, which would then derive all the keys used to encrypt and decrypt the data within the concerning bucket. As this key is the base for decrypting every file that is or will be inside this bucket, a bucket cannot be made private again, once the key has been uploaded [and thus made public]. Instead, a client would have to download all the files, add them to a new bucket and then delete the public bucket.
The encryption key is uploaded to the Bridge in a hexadecimal encoded form.

The codes for determining the access level are uploaded as plaintext strings. There are two codes: PULL and PUSH. PULL adds the permission for the public to download data from the bucket, PUSH on the other hand adds the permission for the public to upload data. It is possible to either give only the right to download, only the right to upload, or both together.
In either way, the owner of the bucket (= creator of the bucket) keeps the rights to both upload and download data to respectively from the public bucket.

With this feature, two new use cases arise, that could be reasonable for a client software to be implemented. The obvious one is implementing the possibility to create and manage public buckets. The less obvious one is to create an interface for the user to access public buckets that were created by other users.

Furthermore, one should consider that the cost for both traffic and storage space used in public buckets are probably going to be charged to the owner of the bucket. A clear statement will be possible by the time the payment system is entirely implemented.

7.4 **Sharding**

Sharding describes the procedure of splitting a file into multiple fractions. This step is performed by the client and is optional, yet in many cases recommended. Figure 7 illustrates the sharding process with a file that is being encrypted before it is sharded.

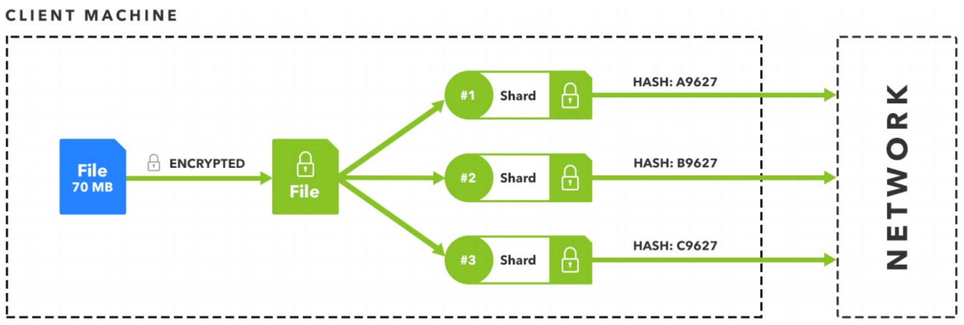


Figure 7: file sharding vizualization (Wilkinson et al., 2016, p. 3)

7.4.1 Advantages of sharding files

Sharding files brings two main advantage: privacy and performance.

The uploaders privacy is increased, because none of the involved farmers stores any entire file, but just a part of it. In this sense, even if a malicious farmer would somehow achieve to decrypt the uploaded data, the farmer would still only be able to read part of the original file, not knowing where the other parts of the file are stored.

The performance of the data transfer (uploads as well as downloads) can be positively influenced by uploading shards simultaneously to different farmers. In this case, the transfer speed is equal to the transfer speed of the slowest farmer multiplied by the size of the shards.
Table 7 illustrates a calculation of this principle using a theoretical example of a file with the size of 32 MB.

Table 7: Comparison of performances between unsharded and sharded upload

FILE UPLOADED ENTIRELY				FILE UPLOAD AS SHARDS			
No.	file size	transfer	time	No.	shard size	transfer	time
1	32 MB	10 Mbit/s	3.2 s.	1	8 MB	10 Mbit/s	0.8 s.
				2	8 MB	25 Mbit/s	0.32 s.
				3	8 MB	4 Mbit/s	2 s.
				4	8 MB	5 Mbit/s	1.6 s.
Total			**3.2 seconds**	**Total**			**2 seconds**

7.4.2 Limits to the advantages

Both the addressed advantages are limited in a similar way. An increased number of shards, also increases the chance of interacting with an unwanted farmer. Concerning privacy, this means that a higher number of farmers increases the possibility to be connected to malicious farmers.

For the performance item, this means that a higher number of farmers increases the possibility to get farmers that are low in performance. Table 8 illustrates this issue using the theoretical example from chapter 7.4.1 .

Table 8: Comparing performances between normally sharded and highly sharded uploads

REASONABLE NUMBER OF SHARDS				INAPPROPRIATE NUMBER OF SHARDS			
No.	file size	transfer	time	No.	shard size	transfer	time
1	8 MB	10 Mbit/s	0.8 s.	1	0.1 MB	10 Mbit/s	0.01 s.
2	8 MB	25 Mbit/s	0.32 s.	2	0.1 MB	4 Mbit/s	0.025 s.
3	8 MB	4 Mbit/s	2 s.			...	
4	8 MB	5 Mbit/s	1.6 s.	183	0.1 MB	0.01 Mbit/s	10 s.
						...	
				319	0.1 MB	25 Mbit/s	0.004 s.
				320	0.1 MB	5 Mbit/s	0.02 s.
Total			**2 seconds**	**Total**			**10 seconds**

The key concept of successful sharding is therefore to find a reasonable balance between the benefits and the risks of the distribution, which can be controlled by the shard size chosen per file. This certainly depends on the actual size of the file, but as a guiding value Storj recommends working with a multiple of Megabytes. (Wilkinson et al., 2016, p. 3).

7.4.3 Order of the shards

When downloading a sharded file, it is important to put the shards together again in the right order. This is fundamental for reproducing the correct data. For this purpose, the Bridge offers the possibility to add an index to each shard's metadata, when they are uploaded. These indexes will then be provided by the Bridge, whenever this file's file download is requested from the Bridge.

7.4.4 Standardized shard sizes

To preserve privacy, it is recommended to have the same shard size for every shard of a file, or even for every single shard uploaded to the network. (Wilkinson et al., 2016, p. 3)

“Standardized sizes dissuade side-channel attempts to determine the content of a given shard, and can mask the flow of shards through the network.” (Wilkinson et al., 2016, p. 3)

This can be achieved by adding meaningless data to shards that are too small in order to “fill it up” to the wanted shard size. Small files could even be put together into one shard.

7.4.5 Examples from the reference implementation

Code snippet 10 shows the sharding process. In this example, the file “doler.amet” is sharded into pieces of 8 Megabytes. The while loop reads one “buffer” at a time. As the buffer has the size of a shard, every iteration of the loop creates one shard. The method createTempFile() adds a random number to the filename. Thus one possible shard-filename of the given code could be: *xyz1324539097.shard* .

```
// sample variables
File targetFolder = new File("/lorem/ipsum/sit/shards");
File inputFile = new File("/lorem/ipsum/sit/doler.amet");
int shardSize = 8 * 1024 * 1024; // 8 MB

FileInputStream inputStream = new FileInputStream(inputFile);
byte[] buffer = new byte[shardSize];
int length;
while ((length = inputStream.read(buffer)) > 0){
    // Create the shard
    File shard = File.createTempFile("xyz",".shard", targetFolder);
    FileOutputStream out = new FileOutputStream(shard);
    out.write(buffer, 0, length);
    out.close();
}
```

Code snippet 10: the sharding process

Code snippet 11 shows the reverse function. It takes various shards and pieces them together into one file.

```
public static void pieceTogetherFile(List<File> shards,
                          File destination) throws IOException {
    FileOutputStream os = new FileOutputStream(destination);
    for(File shard : shards) {
        byte[] shardBytes = readAllBytes(shard.getPath());
        os.write(shardBytes);
    }
    os.close();
}
```

Code snippet 11: creating one single file out of multiple shards

7.4.6 Multithreading for shard up- and downloads

To avoid blocking an application, while an upload or download is taking place, events should be executed in another thread, on the side of the main thread. For the case of up- and downloads to the Storj cloud though, there are even more benefits coming with multithreading. To get the maximum out of the bandwidth the Storj network may provide, it is recommended to upload different shards simultaneously. Using multithreading makes such a concurrency possible.
Storjs Node.js library allows such behaviour and sets its concurrency in a public example to 6 files that may be uploaded at once. (Storj Labs, 2016d)

Furthermore, multithreading may be used in relation with the sharding process. Technically, the uploading process does not have to wait until the sharding process has completed. Instead it might be reasonable to start uploading shards already after the creation of the first shard.

7.5 **Data audits**

7.5.1 Data retrievability

One major challenge of the Storj network is to prove, whether a farmer does indeed store the data it agreed to store, or whether it is a malicious farmer which just claims to store the data, while deleting or manipulating it. Storj's attempt to solve this issue, is to periodically issue data audits for the farmers, to which the farmers are only able to give a correct response, if they can still access the data that was allocated to them.

Storj's implementation of this audit system uses data hashing as its base. For the hashing process, it uses so-called *challenges,* which are essentially 32 byte sized cryptographical salts. A salt is a small amount of random data, which is added to the data that's being hashed – usually for securing the hash. However, in the audit process the challenges are not added for just securing the hashes, but are an essential and inevitable part of the procedure. (Wilkinson et al., 2016, pp. 4,5)

For clients, the data audit generation might best be developed as part of the sharding process, since audits are always tied to shards instead of whole files.

7.5.2 Implementation details

In the audit process, the different challenges are prepended to the shards, one at a time. This concatenation of a challenge and a shard is then being hashed. The resulting hash is the audit. (Wilkinson et al., 2016, p. 4)

$$audit = hash(challenge + sharddata)$$

Source code from Storj Labs reveals, that the hashing function showed in the formula is in reality not just a simple hash function, but instead a combination of four sequential hashing processes using the algorithms SHA-256 and RIPEMD-160.

```
expect(_getChallengeResponse(result)).to.equal(
    utils.rmd160sha256(utils.rmd160sha256(
        challenge + SHARD.toString('hex')
    ))
);
```

Code snippet 12: code from unit test in proof-stream.unit.js from the Storj core repository

Each client is responsible for creating challenges as well as calculating the corresponding audits and send them to the Bridge. This allows the Bridge to prove that farmers store the given shard-data, by issuing challenges and receiving audits as responses, which it can validate against the audits previously created by the client. An audit can only be identical if both the challenge and the shard-data are identical to the data that was previously used by the client to generate the audit itself.
This proof is only secure unless every challenge is only used once. (Wilkinson et al., 2016, pp. 4,5)

Since there is a lot of hashing involved in this process, code snippet 13 should help developers to validate their code respectively make sure that they use the correct hashing functions with the right configurations.
Input: "test"
Output: " 5a30325a141cd691fb3815eff5e0d93ebfee6842"

```
String test = "test";
byte[] tb = test.getBytes();
byte[] hashed = rmd160Sha256(Hex.encode(rmd160Sha256(tb)));
String hexHash = bytesToHex(hashed);
```

Code snippet 13: RIPEMD-160 and SHA-256 hashing function test

7.5.3 Merkle Tree implementation in the auditing process

While further researching the auditing process, one might stumble on the implementation of a technique called Merkle Tree as part of the auditing process. It is a rather fascinating implementation, yet it has little relevance for the implementation of Storj cloud client libraries.

To reduce the audit data stored on the Bridge itself, the Bridge does in fact not exactly follow the procedure described in chapter 0 but uses an implementation of a Merkle Tree instead. Storj Labs' whitepaper gives further insights on this implementation. (Wilkinson et al., 2016, pp. 4,5)

For client developers, the Merkle Tree implementation is of little concern, since for the clients everything remains the same. Its responsibility is just to provide both challenges and audits to the Bridge, while latter is responsible for of the rest. (Wilkinson et al., 2016, pp. 4,5)

7.5.4 Partial audits

Storj Labs has introduced the concept of *partial audits*, to reduce the substantial overhead caused by the hashing processes. The concept is based on the idea of auditing only parts of the whole shard at a time. (Wilkinson et al., 2016, p. 6) As of February 2017 however, this concept is not yet supported by the Bridge (Storj Labs, 2017a).

7.5.5 Number of audits per shard

Storj Labs has not yet communicated any recommendation about the amount of challenges respectively audits to be generated and send to the Bridge. Regarding this question, the only advice that can be found so far is that it would make sense to use a number which is a power of 2, as such amounts of audits are needed for building the Merkle Tree. (Wilkinson et al., 2016, p. 5)

7.6 **Encryption**

How to encrypt files before uploading them to the Storj cloud is a decision on which client developers have complete freedom of choice. Therefore, they also have the responsibility over the security of the files, as the encryption method is crucial to this aspect. Technically it would even be possible to use no encryption at all, which however is strongly inadvisable.

The libraries provided by Storj Labs create a separate key for every file that is uploaded. This is a measure to provide high levels of security, which is indispensable in a trustless network like the one created by Storj Labs. Even though it is not crucial to use a separate encryption key for each file like Storj Labs' libraries do [but only separate initialization vectors], it is certainly a considerable practice.

7.6.1 Data portability and key migration

As data is being encrypted locally on clients, data portability is an issue. How can a user encrypt data in one client, then transfer it over the Storj cloud to another client, and decrypt it there? That is a task the receiving client cannot solve without having the appropriate encryption key for the data. To have full data portability between multiple clients, the encryption keys must therefore be transferred to all involved clients.

A solution for the key migration problem could be an automated key exchange over the internet. This solution could be developed in many different forms and implementations. The benefit of such a solution is that the user does not have to be aware about the existence of that problem, since the exchange can happen fully automated. Nevertheless, there is also an enormous drawback to this solution. By exposing the keys to the internet, this implementation opens a whole field of security issues, with which this system would have to deal with.

Opposed to that solution, there is the solution to manually copy the encryption keys. While this solution is safer indeed, it is also very impractical. Not only would a user eventually have to copy many keys, but this method also just establishes data portability for files, that were uploaded before the keys are copied. Every file that is encrypted afterwards is consequently excluded from this data portability, until the user copies the file again. (Antonopoulus, 2014, p. 85)

To solve this latter problem, an implementation of a key pool could be implemented. In such an implementation, a client does not create the encryption keys directly prior to using them. Instead, it creates a whole pool of keys at once, which the user then copies. The data portability would then be present until the keys of the pool are exhausted. The biggest problem of this method is the uncertainty about the amount of encryption keys used by a user. As the chosen pool size would probably never match the requirements of all users, there would always be some users who store many more keys than they would actually have to, as well as some users who would have to repeat the copy process, as they would not have enough keys. (Antonopoulus, 2014, p. 85)

Another solution is to create keys, which are all derived from the same seed. This method is called *Deterministic Key Derivation*. It contrasts with all the other presented methods, in the way the keys are generated. In the *non-deterministic* method, keys are derived in the ordinary way, by using a separate entropy for each key and derive the key from that number. As all the keys are derived from the same seed using one-way hash functions, this method brings a great advantage over the other methods. The only thing that must be copied to create data portability on multiple devices is the seed. Furthermore, it only has to be copied once, in order to enable the creation of all the used keys, as well as all the keys that will be used in the future. (Antonopoulus, 2014, p. 85) (Wilkinson et al., 2016, p. 21)
In consequence of the used one-way hash functions, a hacker getting hold of any encryption key would neither be able to derive any other encryption key, nor be able to derive the seed from it. (Antonopoulus, 2014, p. 85)

7.6.2 Storj Labs’ Deterministic Key Derivation

Storj Labs integrates an own deterministic key derivation implementation in its libraries. For client-developers who seek compatibility with these libraries, implementing the similar derivation process is required. This chapter gives further insight on the exact procedure to follow, in order create a client that’s compatible with Storj Labs’ libraries. This means, that a correct implementation of the presented steps will help to provide a client, which can download data that was uploaded by using Storj Labs’ tools (at the time this thesis is written this means particularly the Storj CLI) and vice versa.

As randomly generated seeds are difficult for human beings to process, in their natural form they are not convenient to be manually copied from device to device. For this reason, Storj Labs implemented a *mnemonic* solution as described in Bitcoin Improvement Proposal 39. With this implementation, seeds can be transformed into 12 English words (= mnemonic) and vice versa[2]. This way, a user only needs to transfer 12 words to copy the complete information needed to generate all keys necessary for encryption and decryption. It is essential that these words are handled carefully, as they are the key element to a user’s complete encryption data. For this step, developers should seek a library that integrates an implementation of BIP39. (Wilkinson et al., 2016, p. 21) (Antonopoulus, 2014, p. 86)

A generated seed provides the basis for the generation of bucket keys. Bucket keys in turn, provide the basis for generating file keys.

[2] To be precise, the *entropies* (which seeds base on) can be transformed to mnemonics and vice versa.

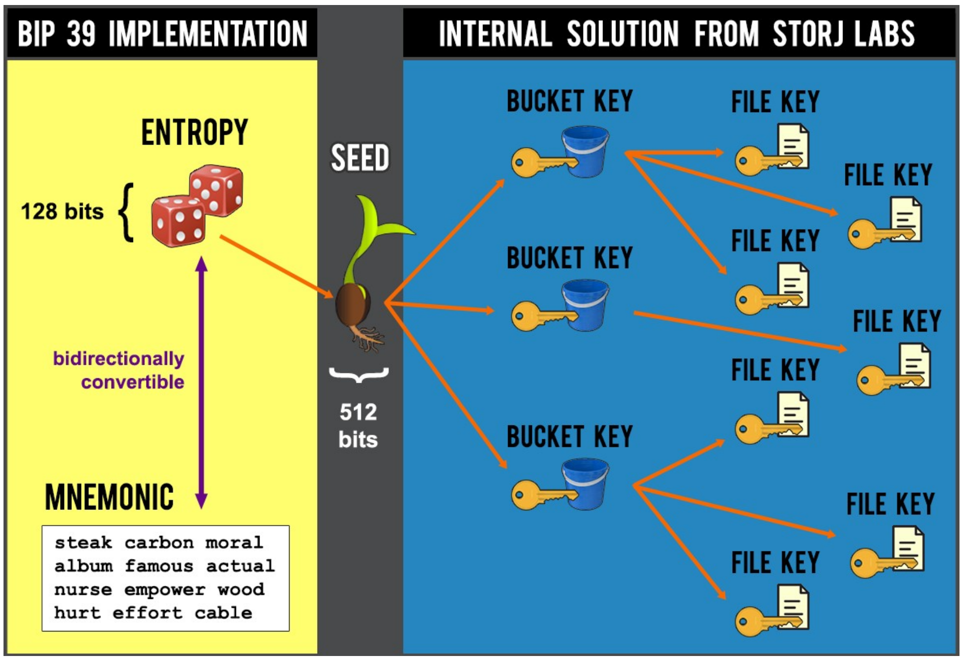


Figure 8: Storj Labs' Deterministic Key Derivation implementation

Figure 8 illustrates this key generation process. Please note that the purple arrow represents a two-way function while the orange arrows represent one-way hash functions. Consequently, with the information of the entropy (respectively of the mnemonic) the seed can be generated, but from the seed neither the entropy nor the mnemonic may be derived. However, with the information of the seed all bucket keys can be generated. Finally, the bucket keys provide the basis for generating all file keys for the files inside of the respective bucket. From a file key, it is however neither possible to derive any other file key nor the bucket key. The same situation is true for the bucket keys: from a bucket key, it is neither possible to derive keys of other buckets, nor to derive the seed.

This procedure also explains perfectly why only one key is uploaded to the Bridge, when a public bucket is created. The uploaded key is the bucket key of the concerning bucket, which the Bridge then shares with all the clients accessing the public bucket. Each involved client then derives the file keys it needs on its own.

7.6.3 Implementation details

This chapter contains a step by step instruction of all the steps needed to implement a key derivation that is compatible with the one Storj Labs integrates in its libraries. It contains all steps of the process, from generating a seed up to an encrypted file. (Storj Labs, 2017c)

First an entropy has to be generated. This step should be implemented by using a trustworthy cryptography library. For compatibility with Storj Labs' libraries, the entropy must be 128 bits in size.

Out of the entropy, a 512 bits long seed may be generated. For this purpose, a library like *bitcoinj* should be used. Such a library will also provide the conversion of mnemonics to entropies and vice versa. BIP 39 furthermore presents an optional solution of a password protection for mnemonics, this feature is however not provided in the implementations from Storj Labs. It is crucial, that the exact same wordlist used by Storj Labs, is applied. As Storj Labs uses the implementation from Bitcore, the wordlist they use can be found on BitPay's Bitcore Github repository: https://github.com/bitpay/bitcore-mnemonic/blob/master/lib/words/english.js

```
// mnemonic used as a sample
String mnemonicSample = "steak carbon essence album famous actual "
                    + "machine empower innocent hurt effort lecture";

// transform mnemonic from String to a list
ArrayList<String> mySeedList = new
ArrayList<String>(Arrays.asList(mnemonicSample.split("\\s+")));

// === CREATE SEED === //
// generating the seed, without using a password ( = empty string )
byte[] seed = MnemonicCode.toSeed(mySeedList, "");
```

Code snippet 14: importing a mnemonic with bitcoinj

Code snippet 14 shows the implementation of a mnemonic import using the bitcoinj library. The hexadecimal representation of the resulting seed would be the following (128 characters / 64 bytes / 512 bits):
438fe281402ecf836cc409901ad8a78d4b34151b4ac8a6fb3df2623225787
ce331fc9dc9a12e4311cd6f96b5482310902fbd3ead2e6c7b9db12d5608d
51b95cf

Creating the bucket key involves appending the bucket id to the seed (both in binary form), and hashing the data using the algorithm SHA-512. The first 256 bits (32 bytes) of the result are the bucket key. If the key were to be uploaded to the Bridge, in order to make a Bucket public, it would have to be transformed into a hexadecimal representation.

```
// using a sample bucket id
String bucketIdSample = "55d5fb0c894440e0440b2932";

byte[] bucketIdBinary = hexToBytes(bucketIdSample);

// === CREATE BUCKET KEY === //
// concatenate the seed and bucketId-bytes.
byte[] bkSource = ArrayUtils.addAll(seed, bucketIdBinary);
byte[] bkUncut = Hashing.sha512().hashBytes(bk_source).asBytes();
byte[] bucketKey = Arrays.copyOfRange(bkUncut, 0, 32);
```

Code snippet 15: generation of bucket key using bucket id and imported seed from above

Code snippet 15 shows the code to the described procedure. The resulting bucket key would have to be put into a hexadecimal representation to upload it to the Bridge for creating a public bucket. In this hexadecimal form, it looks as follows (64 characters / 32 bytes / 256 bits):
bcedc9a3e7e913a3243fa36c6819fff2efe0b6c1d23251f31104daf01cd41cc4

In the next step, the file id must be appended to the bucket key (both in binary form). The resulting data must then be hashed using the algorithm SHA-512. The first 256 bits (32 bytes) of the result are the file key.

```
// using a sample file id
String fileIdSample = "7f587306167139efdc40b0dd";

byte[] fileIdBinary = hexToBytes(fileIdSample);

// === CREATE FILE KEY === //
// concatenate the bucket key and fileId-bytes.
byte[] fkSource = ArrayUtils.addAll(seed, fileIdBinary);
byte[] fkUncut = Hashing.sha512().hashBytes(fk_source).asBytes();
byte[] fileKey = Arrays.copyOfRange(fkUncut, 0, 32);
```

Code snippet 16: generation of file key using sample file id and bucket key from above

The hexadecimal representation file key of this example would be the following code (64 characters / 32 bytes / 256 bits):
3a6efcbefe44cec06b63d606c1dd459f9a4bccc77c156044d7df8796038fc620

In the last step, a given file is encrypted with an AES-CTR algorithm. For this purpose, an encryption key and an initialization vector are necessary. The encryption key is generated by SHA-256 hashing the file key. In contrary to the hashing procedures from the previous steps, for this step not the bytes are hashed, but the hexadecimal representation as a String (based on UTF-8). The initialization vector is the first 16 bytes of a RIPEMD-160 hash of the file id (in binary form).

```
// === CREATE ENCRYPTION KEY === //

String fileKeyHex = bytesToHex(fileKey);
byte[] encryptionKey = Hashing.sha256().hashString(fileKeyHex,
                           Charset.defaultCharset()).asBytes();

// === CREATE INITIALIZATION VECTOR === //
byte[] fileIdHashed = rmd160(fileIdSample.getBytes());
byte[] initVector = Arrays.copyOfRange(fileIdHashed, 0, 16);
```

Code snippet 17: generation of the encryption key and the initialization vector

The hexadecimal representation of the resulting encryption key would be the following code (64 characters / 32 bytes / 256 bits):
b903081d9b21d683b7646614fa9766312a09921a9e830c46c55b79ad20ebfcd2,
and the initialization vector (32 characters / 16 bytes / 128 bits):
7d47fad9a1bfd26139d9d99683845d9a

Code snippet 18 shows the actual encryption of the file with the generated encryption key and initialization vector.

```
// === ENCRYPT FILE === //
Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding");
SecretKeySpec keySpec = new SecretKeySpec(encryptionKey, "AES");
IvParameterSpec ivParamSpec = new IvParameterSpec(initVector);
cipher.init(Cipher.ENCRYPT_MODE, secretKeySpec, ivParameterSpec);

FileOutputStream fos = new FileOutputStream(outputFile);
CipherOutputStream cos = new CipherOutputStream(fos, cipher);
FileInputStream fis = new FileInputStream(inputFile);


byte[] data = new byte[1024];
int read = fis.read(data);
while (read != -1) {
    cos.write(data, 0, read);
    read = fis.read(data);
}

cos.flush();
cos.close();
fos.close();
fis.close();
```

Code snippet 18: encrypting the file

7.6.4 The file id problem with uploads

The attentive reader might have noticed, that there is an inconsistency with the order of the uploading procedure as described above. As explained earlier in this thesis, a file is first encrypted and sharded, then uploaded, and then the Bridge generates a file entry for the file, which also contains the file id. However, as described above, the encryption using deterministic encryption keys uses the id for the generation of the encryption key. This implementation does not seem to add up, if the encryption takes place before the bridge generates the file id.

The solution to this issue, is that the file id is generated in a deterministic way too. It is done in such way, that the derivation of the id might also be done by a client library. For this purpose, the bucket id and the filename are used. First, the bucket id must be prepended to the filename. Then the resulting concatenation must be hashed with SHA-256 and RIPEMD-160. The first 24 characters of this result's hexadecimal representation are the file id. Code snippet 19 shows, the regarding code of the reference implementation.

```
String concatenated = bucketId + fileName;
byte[] asBytes = concatenated.getBytes();
byte[] hashed = rmd160Sha256(asBytes);
String hexadecimal = bytesToHex(hashed);
String fileId = hexadecimal.substring(0, 24);
```

Code snippet 19: deterministic derivation of file ids

7.7 Procedure order of encryption and sharding

Whenever the Storj community speaks about preparing a file for upload, it is usually seen as a sequence of an encrypting process followed by a sharding process. Nevertheless, it is technically also perfectly possible, to change the order. This chapter points out the main advantages and disadvantages of the sharding-first sequence has against the encrypting-first sequence.

7.7.1 The encrypting-first sequence

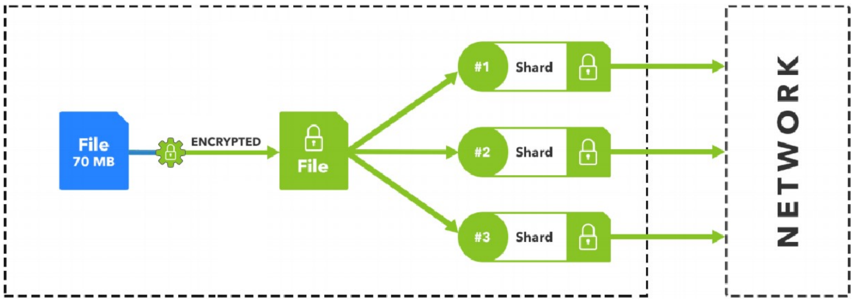


Figure 9: model of the encrypting-first sequence (Wilkinson et al., 2016, p. 3, adapted)

Figure 9 illustrates the *encrypting-first sequence*, which is the commonly implemented sequence. As this is also true for the libraries which Storj Labs implements (Wilkinson et al., 2016, p. 3), the compatibility with the Storj Labs' libraries is one advantage of this sequence.

7.7.2 The sharding-first sequence

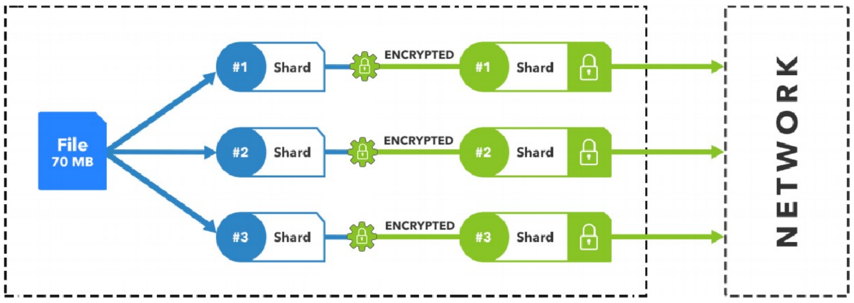


Figure 10: model of the sharding-first sequence (Wilkinson et al., 2016, p. 3, adapted)

Figure 10 illustrates the alternative *sharding-first sequence*. It has particularly performance related advantages over the encrypting-first sequence:

- The client may start the upload of a shard right after this particular shard has been encrypted. The upload process does not have to wait for the whole file to be encrypted.
- The point mentioned above is also true for downloads, although in the reverse order. This means, that a client can start to decrypt a shard when it has been downloaded. The decryption process does not have to wait for every shard to be downloaded.

- The encryption process allows a solution in multiple threads, meaning that each shard may be encrypted respectively decrypted in an own thread.

The main disadvantages of the encrypting-first sequence compared to the sharding-first sequence are security related. It is considered good practice to use a different key for each encryption. If you were to use the same key for multiple encryptions however, it would be vital to use a different initialization vector (IV) per encryption. A secure implementation of a sharding-first sequence would therefore require the creation and storage of a separate initialization vector for each shard. This could generate a significant amount of data to be stored for the client. On the other hand, if this measure was not implemented, the data's security would be severely weakened.

" … any reuse of the per-packet value, called the IV, with the same nonce and key is catastrophic. An IV collision immediately leaks information about the plaintext in both packets." (Housley, 2004, p. 3)

The next security issue of the sharding-first sequence, is that it partly neutralizes the security benefit the shard distribution entails. If shards are encrypted individually, it is easier for a hacker to decrypt a shard, because for a decryption of the shard, only the encryption key and shard itself would be needed. In the encrypting-first sequence on the contrary, the hacker would need the encryption key plus *all* the shards to be able to perform a decryption.

7.7.3 Conclusion

The choice of the chosen sequence appears to be a question of priorities between security and performance. If compatibility towards Storj Labs' libraries is an aim, implementing the encrypting-first sequence is recommendable.

7.8 **Exchange reports**

Exchange reports are reports which the Bridge uses to feed its reputation system, which assesses the farmers of the network. They are sent as POST calls to the Bridge. The endpoint to be called on the Bridge is the URL http://storj.api.io/reports/exchanges .

Since the exchange reports are still under development, this topic is based on assumptions on behalf of the author. Since there is neither documentation, nor any support, the only sources of information the author got hold of, is the source code of the exchange reports implementation on the Bridge, which might not yet be in its final version. Therefore, this chapter needs to be read with caution.

Exchange reports are supportive for the Bridge, but do not provide any direct benefit for clients. For this reason, the Bridge ensures that the clients send their reports, by making the recipience of the reports the trigger for starting the mirroring process (and possibly also the trigger for issuing the first audit for a shard). Therefore, an implementation of the exchange reports is indeed important for client libraries too.

7.8.1 Variables of the exchange reports

What are the variables an exchange report has to include? Following the source code from the Storj core (as shown in Code snippet 20, attached to the appendix), an exchange report contains 8 variables.

```
reporterId: storj.utils.rmd160('client'),
farmerId: storj.utils.rmd160('farmer'),
clientId: storj.utils.rmd160('client'),
dataHash: storj.utils.rmd160('data'),
exchangeStart: Date.now(),
exchangeEnd: Date.now(),
exchangeResultCode: 1000,
exchangeResultMessage: 'SUCCESS'
```

Code snippet 20: from report.unit.js, showing the variables of an exchange report

Table 9 shows the thesis' authors interpretation of the values of these variables when being used in a client implementation.

Table 9: Exchange report variables values for client implementations

reporterId	The clients public key, used for the authentication
farmerId	The farmers id (hash)
clientId	The clients public key, used for the authentication
dataHash	The shards hash
exchangeStart	Unix timestamp of the time before transferring a shard
exchangeEnd	Unix timestamp of the time after transferring a shard
exchangeResultCode	see chapter 0
exchangeResultMessage	see chapter 0

7.8.2 Result messages and codes

The only possible result messages of an exchange report that were found in the Bridge’s source code (reports.js, attached to the appendix), are presented in Table 10.

Table 10: possible result messages for exchange reports

MIRROR_SUCCESS	**SHARD_UPLOAD**	**DOWNLOAD_ERROR**

The message “MIRROR_SUCCESS” is probably only used for the communication between farmers and the Bridge, as the clients are excluded from the mirroring process. The two relevant result messages for the client are therefore SHARD_UPLOAD and DOWNLOAD_ERROR.

There were only two result codes found in the source code of the Storj core (exchange-report.js, attached to the appendix), being **1000** for success and **1100** for failure.

7.9 **Useful libraries and frameworks**

There are various libraries that are supportive for some tasks inside of a given Storj cloud client library. The libraries as presented here are based on the experience on Android and might therefore not be available for other platforms. Nevertheless, it is rather probable that libraries doing a similar thing would be available for the respective cases. Making use of such libraries is recommended as it makes developing easier while providing a high level of quality to the development.

In the Android studio, these libraries are added to the project by creating a dependency in the *build.gradle* of the respective project.

7.9.1 Spongy Castle

Spongy Castle is an adaption of *Bouncy Castle*, which has been created to run on Android platforms. On other systems, bouncy castle may be available instead of spongy castle.

Bouncy / spongy castle provides all kinds of algorithms for modern encryption and hashing methods. It provided all the hashing and encryption needed in the Storj library developed for this thesis. While the scope of bouncy / spongy castle is extensive indeed, the documentation is unfortunately rather limited.

Spongy Castle is developed by *the Legion of the Bouncy Castle Inc.* and available for both Java and C# projects. (The Legion of Bouncy Castle, 2013)

7.9.2 Gson

Gson is a library which simplifies working with JSON based data sources. It allows for serializing and deserializing objects from respectively to JSON. Such a library is strongly recommended to use for Storj libraries, as all communication with the Bridge bases on JSON. Gson is developed by Google and is available for Java (including Android). (Google, 2017)

7.9.3 Volley

Volley (or sometimes referred to as Google Volley) is a library that supports developers in making HTTP calls. It does not only simplify making HTTP calls, but also provides some features to it, like managing multiple concurrent network connections or caching. It allows setting up a request queue, onto which an app may put multiple requests, which are then processed by volley asynchronously.

For the a Storj library, this library is especially useful for the communication with the Bridge, as it involves many requests with little data to be transferred. On the other hand, it is not recommended to be used for up- / and downloading Shards, as this involves a great extent of data transfer.

“Volley is not suitable for large download or streaming operations, since Volley holds all responses in memory during parsing. For large download operations, consider using an alternative like DownloadManager.” (Android Developer Pages, 2017)
Volley is developed by Google and was created for the Android environment. (Segato, 2015)

7.9.4 Download manager

On Android systems, the download manager may be used as a service for transferring shards with as it is designed for efficiently transferring files. In addition to this, it provides a user interface which Android users are familiar with. Figure 11 shows the download managers graphical representation of a Storj-download of the file *freeCat.jpg,* using the reference implementation.

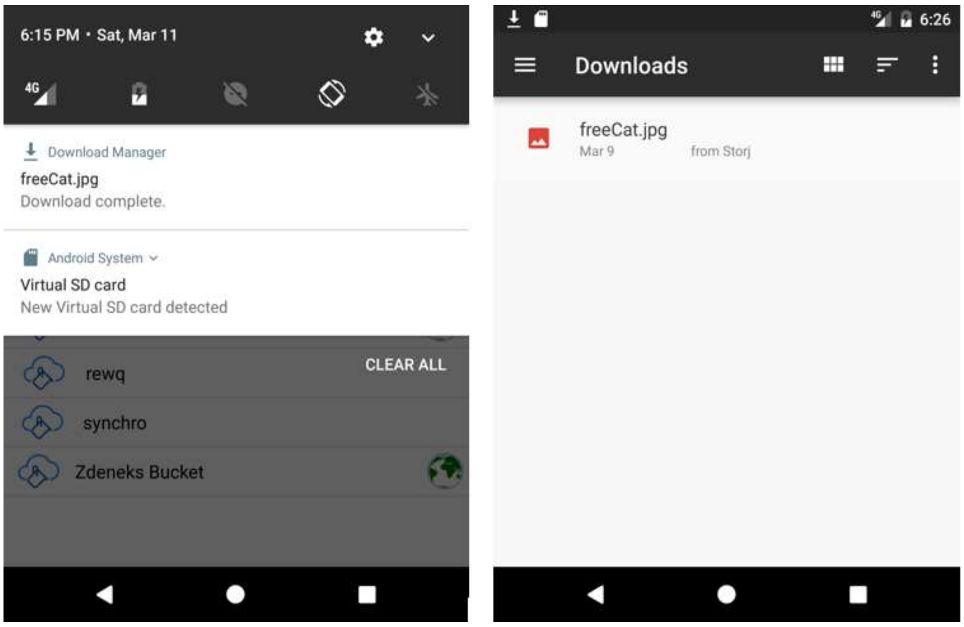


Figure 11: Screenshots from download manager showing download of file freeCat.jpg

7.9.5 Shared Preferences

Shared Preferences is a framework that allows persistent storing of key-value pairs in Android. Even though "shared" sound like this data would be accessible from other applications, this sharing feature is configurable and not the case in the standard setting. However, for improving the security, an encryption of the sensitive values should be considered. On Android, the *Android Keystore System* could be helpful for this purpose.

This framework is useful for the Storj context for storing things like the authentication keys, the encryption seed and the mnemonic as well as configuration variables. Yet on a design level, it is questionable whether this persistence should be provided directly by the library, or rather by the client software.

8 CONCLUSION

The thesis answers the research questions quite precisely. The limitations to the research are because the Storj cloud is still at a rather early stage of development, which is why not all details on the functioning of the Storj cloud have been clarified yet.

The main obstacle in developing a Storj cloud client library lies in the research of information due to lacking support and documentation, and thus perceiving a general understanding of the functioning of the Storj cloud. Yet this thesis, might improve this situation to a certain extent. From a technical point of view, the most challenging domains are the elaborated hashing and encryption functions used by Storj Labs, as well as safely supporting multithreading. In consideration of the different procedures to support, the most sophisticated to be named are the up- and downloading process, the authentication, the deterministic key derivation and the sharding process.

In order to ensure compatibility to the libraries generated by Storj Labs, the most important aspects to consider are using the same procedure for upload preparations – which is first encrypting and then sharding – as well as the implementation of Storj's deterministic key derivation.

Developing a sustainable library is possible with Storj, although a certain flexibility is required, as many changes for the Storj cloud are still expected to come. The best way to avoid complications, is to use one of Storj Labs libraries whenever this is possible.

The knowledge conjunct to the development of such a library might be fairly valuable. The author acquired knowledge about many fields of modern IT, such as encryption, hashing, decentralization, Android development, blockchain technologies, or the HTTP protocol to name a few.

In the future, Storj Labs will support more systems by providing libraries for them, which will reduce the need for third parties to develop their own libraries. Nevertheless, early adopters could get themselves into a good position inside of the Storj environment, the sooner they enter it.
The Storj cloud itself is just in the transition from experimental system to a productive system. The ongoing integration of the payment system is a major step in this endeavour. How the Storj Labs will perform in the market of cloud storage providers will soon be seen. Companies who follow this market development could achieve a competitive advantage by switching from traditional clouds to the Storj cloud, should the Storj cloud prevail.

9 LIST OF TABLES

10 LIST OF FIGURES

11 LIST OF CODE SNIPPETS

12 REFERENCES

Amazon webservices. (2017). *Protecting Data in Amazon S3*. Retrieved 2 March 2017 from Amazon developer guide: http://docs.aws.amazon.com/AmazonS3/latest/dev/DataDurability.html

Android Developer Pages. (2017). *Transmitting Network Data Using Volley*. Retrieved 30 January 2017 from Android developer's guide: https://developer.android.com/training/volley

Antonopoulus, A. (2014). *Mastering Bitcoin.* Sebastopol, California, USA: O'Reilly.

AppDynamics Inc. (2016). *Node.js Supported Environments*. Retrieved 7 February 2017 from AppDynamics documentation: https://docs.appdynamics.com/display/PRO42/Node.js+Supported+Environme nts

Bloomberg. (2017). *Company Overview of Storj Labs, Inc.* Retrieved 22 January 2017 from bloomberg.com: http://www.bloomberg.com/research/stocks/private/snapshot.asp?privcapId= 309570101

D. Johnson, A. Menezes, S. Vanstone. (2016). *The Elliptic Curve Digital Signature Algorithm (ECDSA)*. Waterloo, Ontario, Canada. Retrieved 18 February 2017 from UC Santa Barbara Engineering: http://cs.ucsb.edu/~koc/ccs130h/notes/ecdsa-cert.pdf

Dropbox. (2017). *Where does Dropbox store my data?* Retrieved 14 February 2017 from Dropbox Help Center: https://www.dropbox.com/en/help/7

Fuller, B. (2016). *Page on Github*. Retrieved 18 February 2017 from Github: https://github.com/braydonf/sips/blob/be135822ff969331737651d3db288560 ef3178f3/sip-downloads.md

Gartner Inc. (2016). *Gartner Says by 2020 "Cloud Shift" Will Affect More Than $1 Trillion in IT Spending*. Stamford, Connectitut, United States of America. Retrieved 4 February 2017 from Gartner Newsroom: http://www.gartner.com/newsroom/id/3384720

Google. (2017). *Github repository.* Retrieved 9 March 2017 from Github: https://github.com/google/gson

HAMK. (2017). *Thesis guide*. Hämeenlinna, Häme Province, Finland: HAMK.

Holloh, N. (2017). *Analyse der User Experience moderner Cloudspeicher-Anwendungen im Hinblick auf deren Eignung für Blockchain-basierte Cloudspeicherdienste*.

Bachelor's thesis. Frankfurt School of Finance and Management

Housley, R. (2004). *Request for Comments: 3686.* Reston, Virginia, U.S.A.: Internet Society. Retrieved 27 February 2017 from RFC Editor: https://www.rfc-editor.org/rfc/pdfrfc/rfc3686.txt.pdf

Hoyes, S. (2014). *Our Vision.* Retrieved 28 January 2017 from the Storj blog: http://blog.storj.io/post/97521475738/our-vision

Jaeger, P., Lin, J., Grimes, J., Simmons, S. (2009). *Where is the cloud? Geography, economics, environment, and jurisdiction in cloud computing*. Chicago, Illinois, USA. Retrieved 14 February 2017 from First Monday journal: http://pear.accc.uic.edu/ojs/index.php/fm/article/view/2456/2171

littleskunk. (2017). *Storj community chat*. (s.n., Interviewer) Retrieved 8 February 2017 from the Storj community chat: https://community.storj.io/

Node.js. (2017). *Downloads*. Retrieved 20 February 2017 from Node.js website: https://nodejs.org/en/download/

P. Mell, T. G. (2011). *The NIST Definition of Cloud*. United States of America. Retrieved 4 March 2017 from Winthrop University website: http://faculty.winthrop.edu/domanm/csci411/Handouts/NIST.pdf

Public Law of USA. (2002). *Homeland Security Act*. Retrieved 27 February 2017 from Department of Homeland Security: https://www.dhs.gov/xlibrary/assets/hr_5005_enr.pdf

Rawle, C. (2016). *Storj: A New Way Of Storing On The Cloud*. Retrieved 29 January 2017 from Beehive Startups website: https://beehivestartups.com/storj-a-new-way-of-storing-on-the-cloud-9cae4a664aea

Segato, G. (2015). *An Introduction to Volley*. Retrieved 11 February 2017 from envatotuts+ website https://code.tutsplus.com/tutorials/an-introduction-to-volley--cms-23800

Storj Labs. (2016a). *What is Storj? Part 2 of 3: The Storj Toolset.* Retrieved from Storj blog: http://blog.storj.io/post/147241544533/what-is-storj-part-2-of-3-the-storj-toolset

Storj Labs. (2016b). *Getting Started Guide*. Retrieved 21 February 2017 from Storj documentations: https://docs.storj.io/docs

Storj Labs. (2016c). *Release note on Github*. Retrieved 20 February 2017 from Github: https://github.com/Storj/bridge/releases/tag/v1.1.0

Storj Labs. (2016d). *Source code of the file 6a-upload-file.js.* Retrieved 12 February 2017 from Github: https://github.com/Storj/core/blob/b1301cc94b68420d904b420d3cd2f015c4255262/example/6a-upload-file.js

Storj Labs. (2016e). *Storj Bridge API manual*. Retrieved 1 March 2017, from Storj API: website: https://storj.io/api.html

Storj Labs. (2017a). *Repository on Github.* Retrieved 22 February 2017 from Github: https://github.com/Storj/bridge

Storj Labs. (2017b). *Page on Github*. Retrieved 22 February 2017 from Github: https://github.com/Storj/bridge/blob/master/doc/auth.md

Storj Labs. (2017c). *Page on Github.* Retrieved 27 February 2017 from Github: https://github.com/frdwrd/core/blob/bb8f6c7403129b7dc27165b21a487dd5b74c2049/doc/file-encryption.md

Storj Labs. (2017d). *Github Release page*. Retrieved 1 March 2017 from Github: https://github.com/Storj/storjshare-gui/releases

Storj Labs. (2017e). *Repository on Github*. Retrieved from 5 March 2017 from Github: https://github.com/Storj/storj.js

The Legion of Bouncy Castle. (2013). *Bouncy Castle website*. Retrieved 26 February 2017 from Bouncy Castle website: https://www.bouncycastle.org/

United States Department of Justice. (2001). *The USA PATRIOT Act: Preserving Life and Liberty.* Retrieved 19 January 2017 from US Department of Justice: https://www.justice.gov/archive/ll/highlights.htm

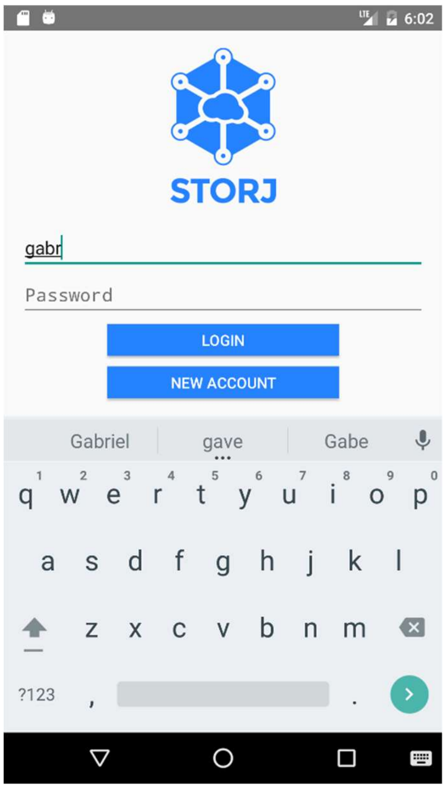
Wilkinson, S. (2016). *Storj Master Plan*. Retrieved 26 February 2017 from Medium website: https://medium.com/@storjproject/storj-master-plan-45bfb63c6b38#.6igjujz7a

Wilkinson, S. (2017). *S3 Failure Highlights the Need for Decentralized Services like Storj*. Retrieved 7 March 2017 from Medium website: https://medium.com/@storjproject/s3-failure-highlights-the-need-for-decentralized-services-like-storj-ab30a5769cf8

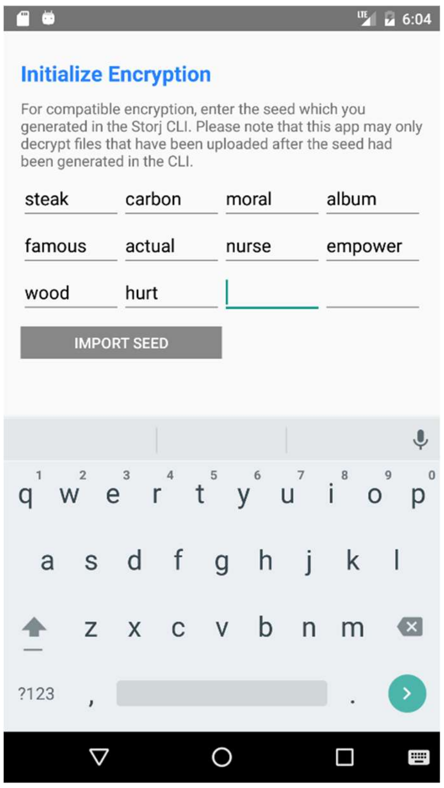
Wilkinson, S., Boshevski, T., Brandoff, J., Prestwich, J., Hall, G., Gerbes, P., Hutchins, P., Pollard, C. (2016). *Storj - A Peer-to-Peer Cloud Storage Network v2.0*. Retrieved 10 January 2017 from https://storj.io/storj.pdf www.netmarketshare.com. (2017). *Android Market Share on Mobile/Tablet*. Retrieved from NETMARKETSHARE website: https://www.netmarketshare.com/report.aspx?qprid=9&qpaf=&qpcustom=Android&qpcustomb=1

Appendix 1

SCREENSHOTS FROM THE REFERENCE IMPLEMENTATION



Screen 1: login page

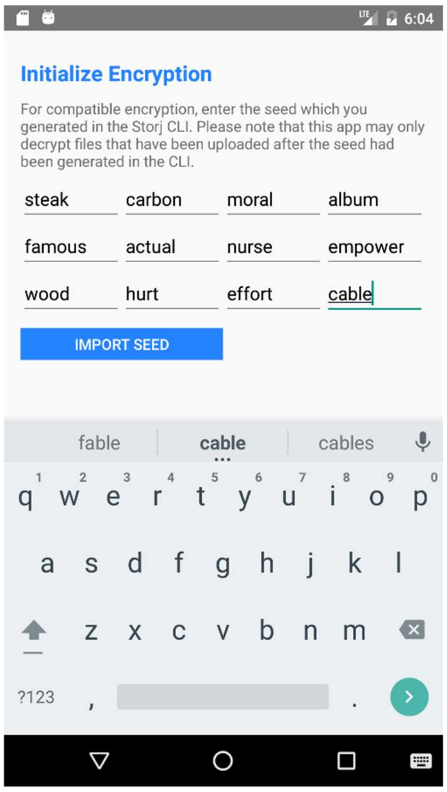


Initialize Encryption

For compatible encryption, enter the seed which you generated in the Storj CLI. Please note that this app may only decrypt files that have been uploaded after the seed had been generated in the CLI.

steak	carbon	moral	album
famous	actual	nurse	empower
wood	hurt		

Screen 2: mnemonic import

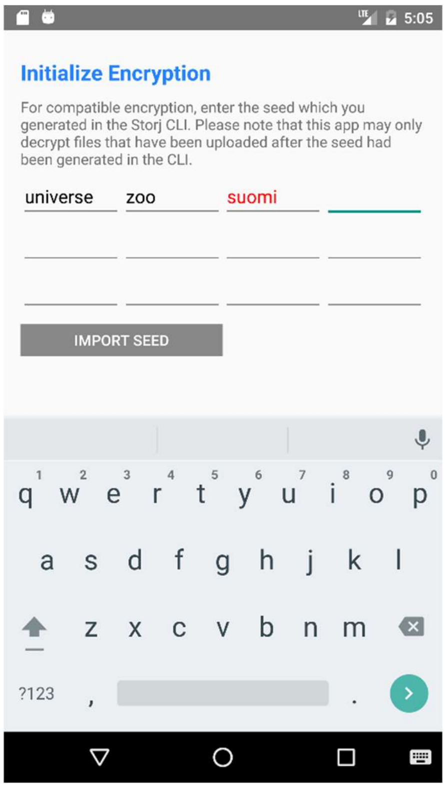


Initialize Encryption

For compatible encryption, enter the seed which you generated in the Storj CLI. Please note that this app may only decrypt files that have been uploaded after the seed had been generated in the CLI.

steak	carbon	moral	album
famous	actual	nurse	empower
wood	hurt	effort	cable

Screen 3: correct mnemonic



Initialize Encryption

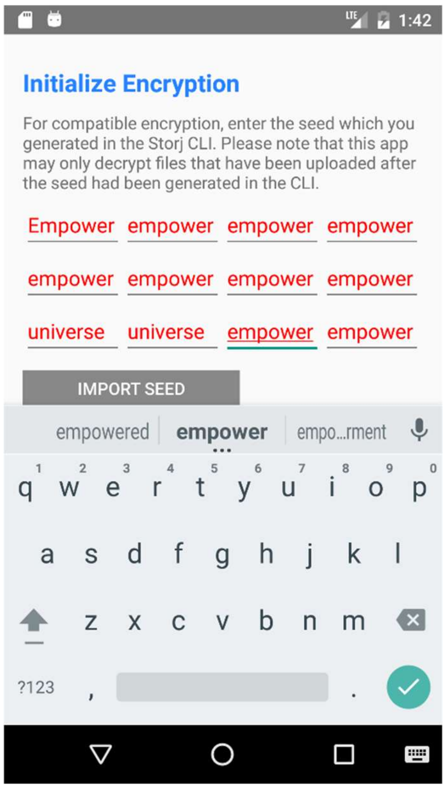
For compatible encryption, enter the seed which you generated in the Storj CLI. Please note that this app may only decrypt files that have been uploaded after the seed had been generated in the CLI.

universe	zoo	suomi	

Screen 4: wrong word in mnemonic

Appendix 1

SCREENSHOTS FROM THE REFERENCE IMPLEMENTATION

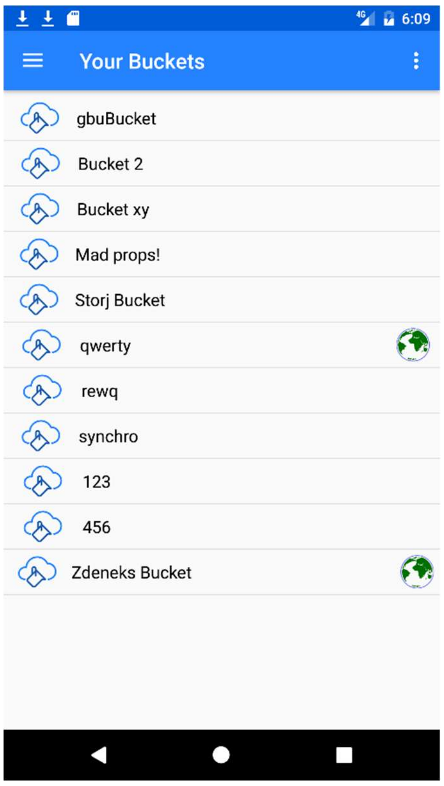


Initialize Encryption

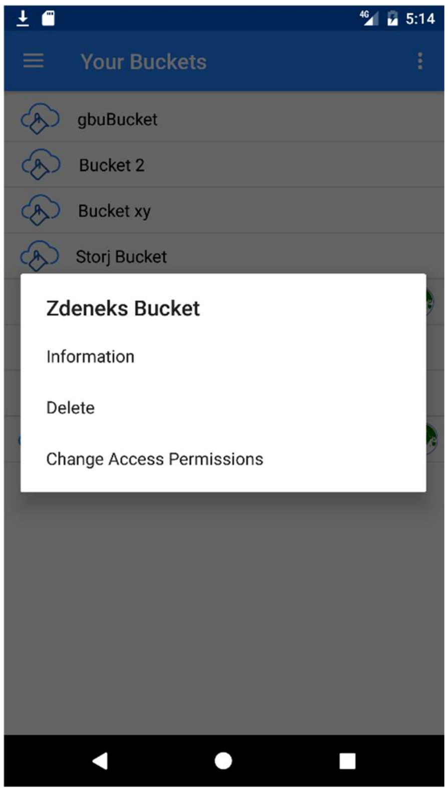
For compatible encryption, enter the seed which you generated in the Storj CLI. Please note that this app may only decrypt files that have been uploaded after the seed had been generated in the CLI.

Empower empower empower empower
empower empower empower empower
universe universe empower empower

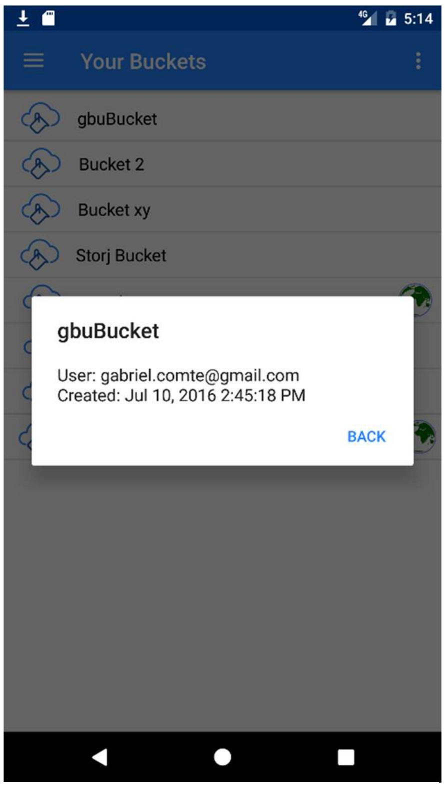
Screen 5: wrong mnemonic consisting of correct words



Screen 6: buckets overview with earth symbol for public buckets



Screen 7: available actions for buckets



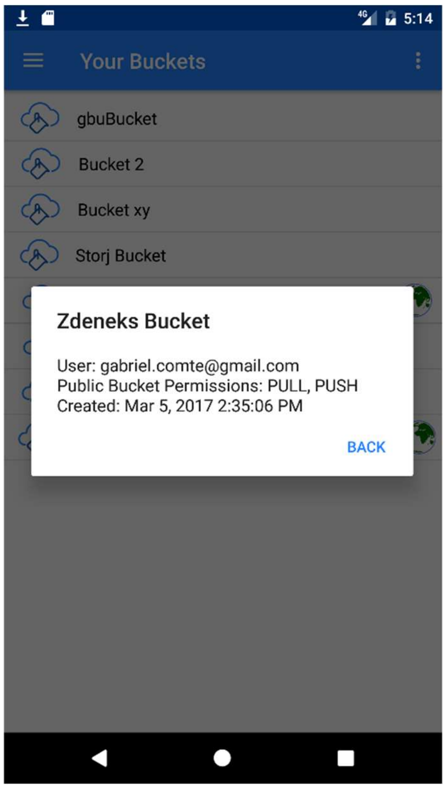
gbuBucket

User: gabriel.comte@gmail.com
Created: Jul 10, 2016 2:45:18 PM

Screen 8: information dialog for normal (private) buckets

Appendix 1

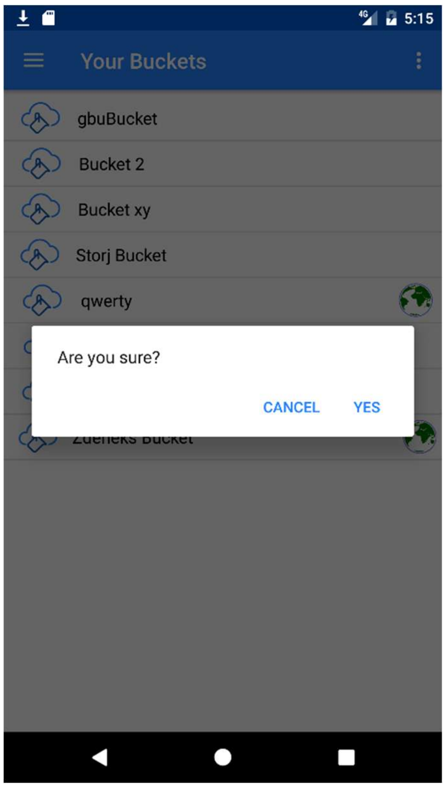
SCREENSHOTS FROM THE REFERENCE IMPLEMENTATION



Zdeneks Bucket

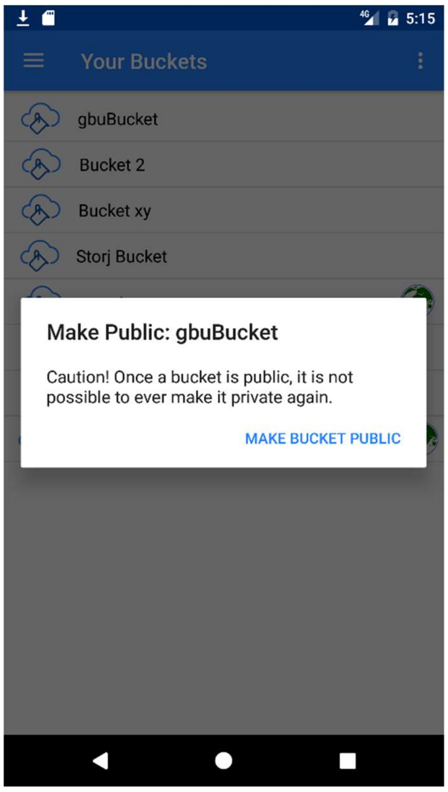
User: gabriel.comte@gmail.com
Public Bucket Permissions: PULL, PUSH
Created: Mar 5, 2017 2:35:06 PM

Screen 9: information dialog for public buckets



Are you sure?

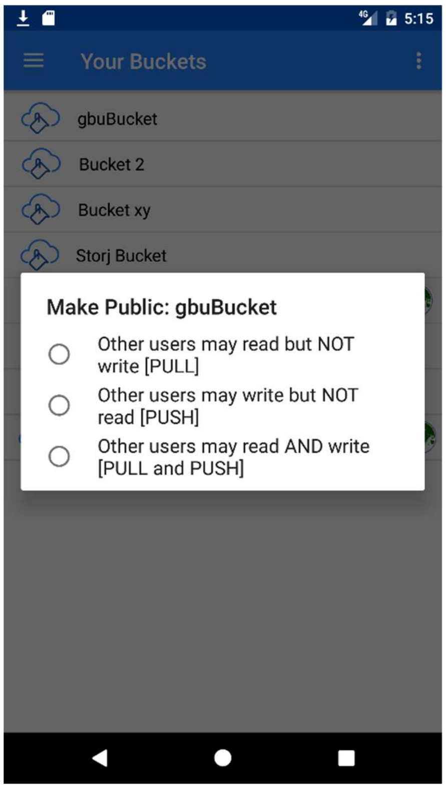
Screen 10: delete bucket verification



Make Public: gbuBucket

Caution! Once a bucket is public, it is not possible to ever make it private again.

Screen 11: make bucket public first dialog



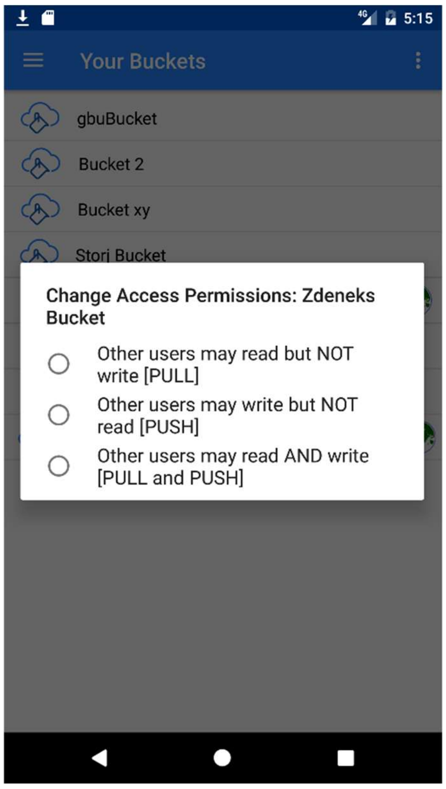
Make Public: gbuBucket

- Other users may read but NOT write [PULL]
- Other users may write but NOT read [PUSH]
- Other users may read AND write [PULL and PUSH]

Screen 12: make bucket public second dialog

Appendix 1

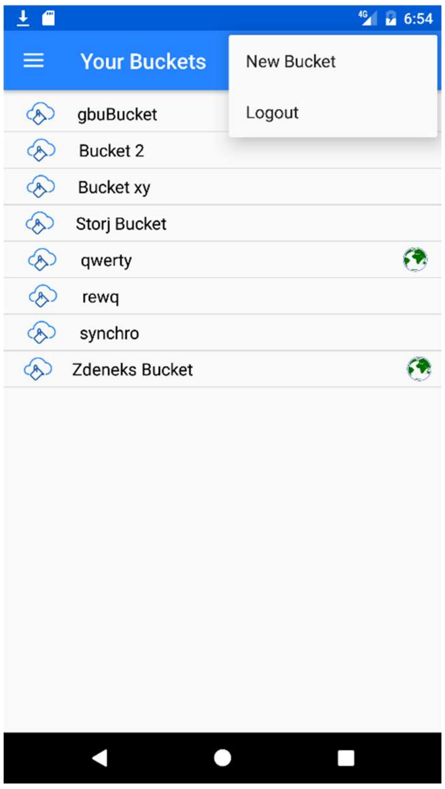
SCREENSHOTS FROM THE REFERENCE IMPLEMENTATION



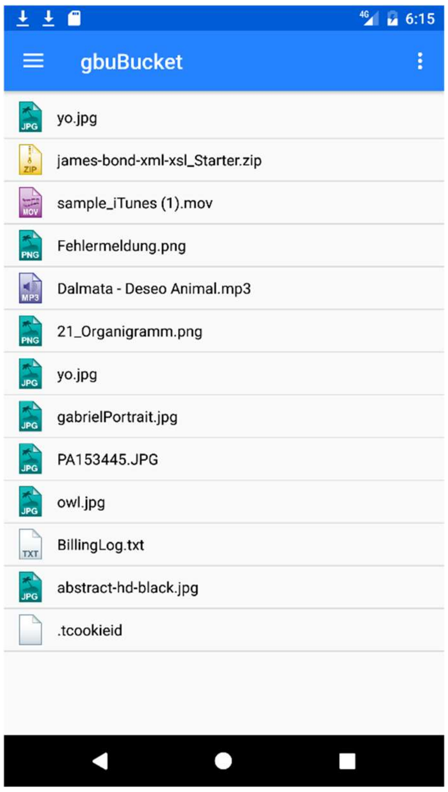
Change Access Permissions: Zdeneks Bucket

- Other users may read but NOT write [PULL]
- Other users may write but NOT read [PUSH]
- Other users may read AND write [PULL and PUSH]

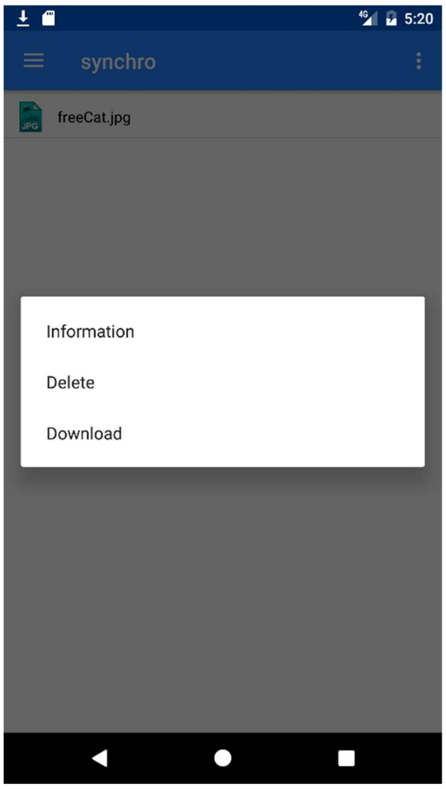
Screen 13: changing permissions for a public bucket



Screen 14: actions new bucket and logout

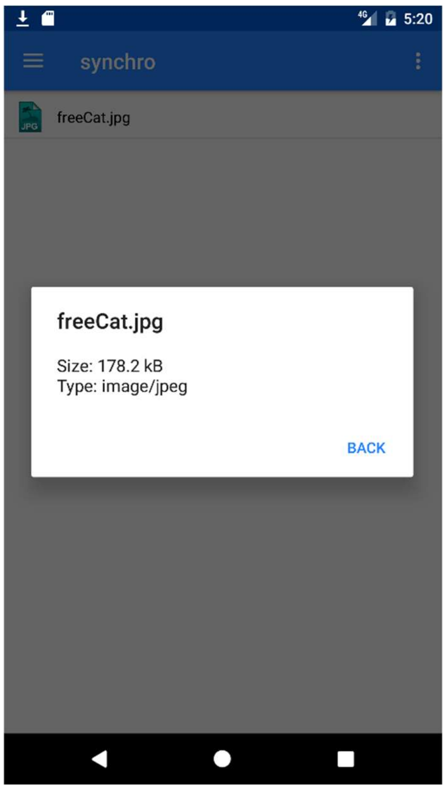


Screen 15: files overview



Screen 16: available actions for files

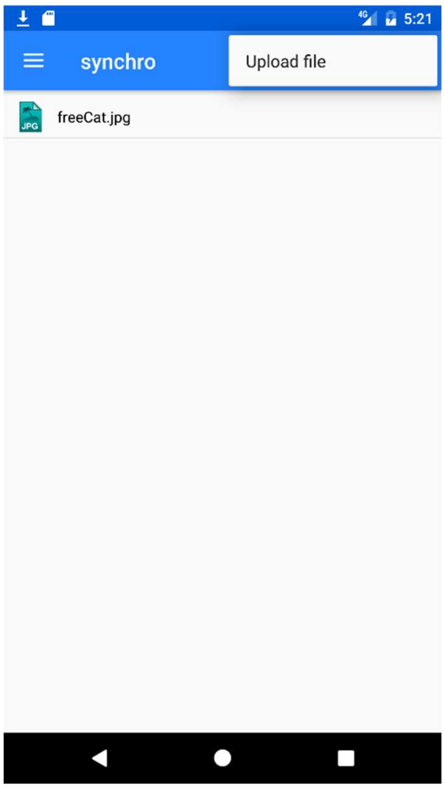
SCREENSHOTS FROM THE REFERENCE IMPLEMENTATION



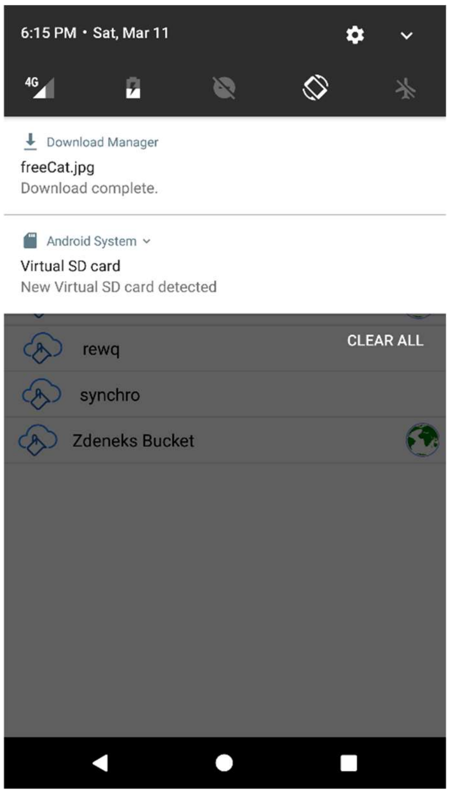
freeCat.jpg

Size: 178.2 kB
Type: image/jpeg

Screen 17: file information



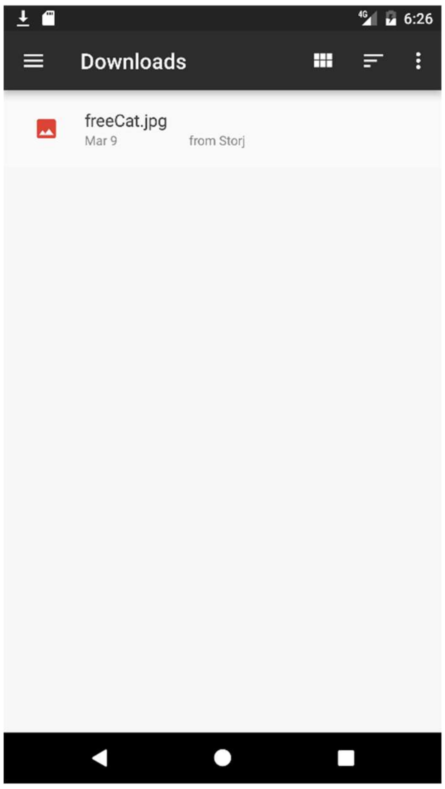
Screen 18: actions new bucket and logout



freeCat.jpg
Download complete.

Virtual SD card
New Virtual SD card detected

Screen 19: download notification



Screen 20: completed download in Android download manager

Appendix 2

THESIS RELEVANT CODE FROM STORJ LABS' GITHUB REPOSITORY

core/lib/bridge-client/exchange-report.js from https://github.com/Storj/core accessed on 23. February 2017

```
'use strict';

var assert = require('assert');

/**
 * Represents a report to a bridge regarding the result of a shard exchange
 * @constructor
 * @param {Object} options
 * @param {String} options.reporterId
 * @param {String} [options.farmerId]
 * @param {String} [options.clientId]
 */
function ExchangeReport(options = {}) {
  /* eslint complexity: [2, 7] */
  if (!(this instanceof ExchangeReport)) {
    return new ExchangeReport(options);
  }

  assert(options.reporterId, 'Invalid reporterId');

  this._r = {
    dataHash: options.dataHash || null,
    reporterId: options.reporterId,
    farmerId: options.farmerId,
    clientId: options.clientId,
    exchangeStart: options.exchangeStart || null,
    exchangeEnd: options.exchangeEnd || null,
    exchangeResultCode: options.exchangeResultCode || null,
    exchangeResultMessage: options.exchangeResultMessage || null
  };
}

ExchangeReport.SUCCESS = 1000;
ExchangeReport.FAILURE = 1100;

/**
 * Starts recording duration of exchange
 * @param {String} dataHash - The shard hash as reference
 */
ExchangeReport.prototype.begin = function(dataHash) {
  assert(dataHash, 'You must supply a dataHash to begin an exchange report');
  this._r.dataHash = dataHash;
  this._r.exchangeStart = Date.now();
};

/**
 * Ends the recording time a set result code and message
 * @param {Number} resultCode - Exchange result code
 * @param {String} resultMessage - Exchange result message
 */
ExchangeReport.prototype.end = function(resultCode, resultMessage) {
  assert(resultCode, 'You must supply a result code');
  assert(resultMessage, 'You must supply a result message');
  this._r.exchangeEnd = Date.now();
  this._r.exchangeResultCode = resultCode;
  this._r.exchangeResultMessage = resultMessage;
};

/**
 * Returns a plain report object
 * @returns {Object}
 */
ExchangeReport.prototype.toObject = function() {
  return JSON.parse(JSON.stringify(this._r));
};

module.exports = ExchangeReport;
```

Appendix 2

THESIS RELEVANT CODE FROM STORJ LABS' GITHUB REPOSITORY

bridge/lib/server/routes/reports.js from https://github.com/Storj/bridge, accessed on 23. February 2017

```
'use strict';

const Router = require('./index');
const log = require('../../logger');
const middleware = require('storj-service-middleware');
const errors = require('storj-service-error-types');
const inherits = require('util').inherits;
const BucketsRouter = require('./buckets');
const constants = require('../../constants');
const async = require('async');
const storj = require('storj-lib');

/**
 * Handles endpoints for reporting
 * @constructor
 * @extends {Router}
 */
function ReportsRouter(options) {
  if (!(this instanceof ReportsRouter)) {
    return new ReportsRouter(options);
  }

  Router.apply(this, arguments);
}

inherits(ReportsRouter, Router);

/**
 * Creates an exchange report
 * @param {http.IncomingMessage} req
 * @param {http.ServerResponse} res
 * @param {Function} next
 */
ReportsRouter.prototype.createExchangeReport = function(req, res, next) {
  const self = this;
  var exchangeReport = new this.storage.models.ExchangeReport(req.body);
  var projection = {
    hash: true,
    contracts: true
  };

  this.storage.models.Shard.find({
    hash: exchangeReport.dataHash
  }, projection, function(err, shards) {
    if (err) {
      return next(new errors.InternalError(err.message));
    }

    if (!shards || !shards.length) {
      return next(new errors.NotFoundError('Shard not found for report'));
    }

    // TODO: Add signature/identity verification

    // NB: Kick off mirroring if needed
    self._handleExchangeReport(exchangeReport, (err) => {
      /* istanbul ignore next */
      if (err) {
        return log.warn(err.message);
      }

      /* istanbul ignore next */
      log.info('exchange report triggered a mirroring operation');
    });
    log.info('received exchange report');
    exchangeReport.save(function(err) {
      if (err) {
        return next(new errors.BadRequestError(err.message));
      }

      log.info('exchange report saved');
      res.status(201).send({});
    });
  });
};

/**
 * @private
 */
ReportsRouter.prototype._handleExchangeReport = function(report, callback) {
  const {dataHash, exchangeResultMessage} = report;

  switch (exchangeResultMessage) {
    case 'MIRROR_SUCCESS':
    case 'SHARD_UPLOADED':
    case 'DOWNLOAD_ERROR':
      this._triggerMirrorEstablish(constants.M_REPLICATE, dataHash, callback);
      break;
    default:
      callback(new Error('Exchange result type will not trigger action'));
  }
};

ReportsRouter._sortByResponseTime = function(a, b) {
  const aTime = a.contact.responseTime || Infinity;
  const bTime = b.contact.responseTime || Infinity;
  return (aTime === bTime) ? 0 : (aTime > bTime) ? 1 : -1;
};

/**
 * Loads some mirrors for the hash and establishes them
 * @private
 */
ReportsRouter.prototype._triggerMirrorEstablish = function(n, hash, done) {
  const self = this;

  function _getMirrors(callback) {
    self.storage.models.Mirror.find({ shardHash: hash })
      .populate('contact')
      .exec(callback);
  }

  function _getMirrorCandidate(mirrors, callback) {
    let established = [], available = [];

    mirrors.forEach((m) => {
      if (!m.contact) {
        log.warn('Mirror %s is missing contact in database', m._id);
      } else if (!m.isEstablished) {
        available.push(m);
      } else {
        established.push(m);
      }
    });

    if (available.length === 0) {
      return callback(new Error('No available mirrors'));
    }

    if (established.length >= n) {
      return callback(new Error('Auto mirroring limit is reached'));
    }

    available.sort(ReportsRouter._sortByResponseTime);

    callback(null, available.shift());
  }

  function _getRetrievalTokenFromFarmer(mirror, callback) {
    self.contracts.load(hash, (err, item) => {
      if (err) {
        return callback(err);
      }

      let farmers = Object.keys(item.contracts);
      let pointer = null;
      let test = () => farmers.length === 0 || pointer !== null;
      let contact = storj.Contact(mirror.contact.toObject());

      async.until(test, (done) => {
        self.getContactById(farmers.shift(), (err, result) => {
          if (err) {
            return done();
          }

          let farmer = storj.Contact(result.toObject());

          self.network.getRetrievalPointer(
            farmer,
            item.getContract(farmer),
            (err, result) => {
              pointer = result;
              done();
            }
          );
        });
      }, () => {
        if (!pointer) {
          return callback(new Error('Failed to get pointer'));
        }

        callback(null, pointer, mirror, contact, item);
      });
    });
  }

  function _establishMirror(source, mirror, contact, item, callback) {
    self.network.getMirrorNodes(
      [source],
      [contact],
      (err) => {
        if (err) {
          return callback(err);
        }

        mirror.isEstablished = true;
        mirror.save();
        item.addContract(contact, storj.Contract(mirror.contract));
        self.contracts.save(item, callback);
      }
    );
  }

  async.waterfall([
    _getMirrors,
    _getMirrorCandidate,
    _getRetrievalTokenFromFarmer,
    _establishMirror
  ], done);
};

/**
 * @private
 */
ReportsRouter.prototype.getContactById = BucketsRouter.prototype.getContactById;

/**
 * @private
 */
ReportsRouter.prototype._definitions = function() {
  return [
    ['POST', '/reports/exchanges', middleware.rawbody,
     this.createExchangeReport.bind(this)]
  ];
};

module.exports = ReportsRouter;
```

Appendix 2

THESIS RELEVANT CODE FROM STORJ LABS' GITHUB REPOSITORY

bridge/test/server/routes/reports.unit.js from https://github.com/Storj/bridge, accessed on 23. February 2017

Please Note: As this source file is very big, only the most relevant code snippet of the whole file is attached here in the appendix.

```
describe('#_handleExchangeReport', function() {

  let _triggerMirrorEstablish;

  before(() => {
    _triggerMirrorEstablish = sinon.stub(
      reportsRouter,
      '_triggerMirrorEstablish'
    ).callsArg(2);
  });
  after(() => _triggerMirrorEstablish.restore());

  it('should callback error if not valid report type', function(done) {
    reportsRouter._handleExchangeReport({
      shardHash: 'hash',
      exchangeResultMessage: 'NOT_VALID'
    }, (err) => {
      expect(err.message).to.equal(
        'Exchange result type will not trigger action'
      );
      done();
    });
  });

  it('should trigger a mirror on SHARD_UPLOADED', function(done) {
    reportsRouter._handleExchangeReport({
      shardHash: 'hash',
      exchangeResultMessage: 'SHARD_UPLOADED'
    }, done);
  });

  it('should trigger a mirror on MIRROR_SUCCESS', function(done) {
    reportsRouter._handleExchangeReport({
      shardHash: 'hash',
      exchangeResultMessage: 'MIRROR_SUCCESS'
    }, done);
  });

  it('should trigger a mirror on DOWNLOAD_ERROR', function(done) {
    reportsRouter._handleExchangeReport({
      shardHash: 'hash',
      exchangeResultMessage: 'DOWNLOAD_ERROR'
    }, done);
  });

});
```