Babeş-Bolyai University Cluj-Napoca

Faculty of Mathematics and Computer Science

Specialization Computer Science in English

Diploma Thesis

A decentralized storage and file synchronization system over IPFS

Author:

Alexandru Nagy

Supervisor:

Conf. Dr. Adrian Sergiu Darabant

2019

Universitatea Babeş-Bolyai Cluj-Napoca

Facultatea de Matematică şi Informatică

Specializarea Informatică Engleză

LUCRARE DE LICENŢĂ

Un sistem descentralizat de stocare şi sincronizare fişiere bazat pe IPFS

Absolvent:

Alexandru Nagy

Conducător ştiinţific:

Conf. Dr. Adrian Sergiu Darabant

2019

Contents

1 Introduction **3**

1.1 Motivation 3

1.2 Solution 4

2 Background and Related Work **5**

2.1 IPFS 5

2.1.1 Identities 5

2.1.2 Routing 5

2.1.3 BitSwap - File Exchange 6

2.1.4 Object Merkle DAG 6

2.1.5 Naming 7

2.1.6 Local storage and pinning 7

2.2 IPFS Cluster 8

2.2.1 Establishing Consensus 8

2.2.2 The Raft Algorithm 9

2.3 Related work 9

2.3.1 Filecoin 9

2.3.2 Storj 10

2.3.3 Sia 10

3 System Design **11**

3.1 Requirements 11

3.2 Technologies 12

3.2.1 Firebase 12

3.2.2 IPFS 13

3.2.3 IPFS Cluster 15

3.3 Synchronization 16
3.4 Security 17
3.4.1 Data encryption using AES-256 17
3.4.2 Authentication security 18
3.4.3 Database security 19
3.5 Storage and pinning 19
3.6 Architecture 20

4 Implementation **22**
4.1 Tools 22
4.1.1 Programming language 22
4.1.2 IDE 22
4.1.3 Version control 23
4.2 Libraries 23
4.3 Application 25
4.3.1 Registration and authentication 25
4.3.2 Encryption 26
4.3.3 Storage 27
4.3.4 Synchronization 30
4.3.5 File system events monitoring 31
4.3.6 Deployment 34

5 Evaluation **35**
5.1 Test setup 35
5.2 Test run 36
5.3 Results 38

6 Conclusion **40**

Appendices **43**

A IPFS installation & configuration **43**

B IPFS cluster installation & configuration **44**

Chapter 1

Introduction

1.1 Motivation

The problem of storage limitation is one that constantly appears. Numerous solutions to this problem have developed in the last 20 years and over time a transition was made from storing data on personal owned external physical devices to third party cloud based solutions like Amazon Drive, Google Drive or Dropbox. For some time there were available even unlimited storage plans, which in the end were proven to be limited. Even so, as these companies offer a high amount of storage space for a relatively low price, with very low latency, high data availability and a constant increase of their storage capacity, the aforementioned problem seems to be solved entirely. And it is, indeed, but only at the practical level. The core problem of these commercial solutions is that they are highly centralized, having control over all the stored user data. Furthermore, even though having multiple layers of security, they still present a single point of security and thus a breach in the system could expose entirely the data of a large part if not all the users, as it was the case with Dropbox in 2016, when the data of 68 millions user account details were leaked.

Steps were made in the opposite direction as development in peer-to-peer systems and the rise in popularity of blockchain and decentralized applications stimulated the creation and adoption of new solutions for secure data storage across peer-to-peer networks, such as Sia or Storj, which will be further discussed. They propose a different concept in which users share personal storage space through an incentive based scheme and thus creating a global network of nodes that act as servers of information, where data is stored shredded and encrypted, instead of just a few server farms spread across the world.

1.2 Solution

Our proposal is *IPFS-Drive*, a peer-to-peer application for file storage and synchronization between computers, based on the InterPlanetary File System, an innovative protocol aimed at replacing HTTP through content based addressing. The purpose of the application is to offer a simple and secure way of storing and synchronizing data into a peer-to-peer network, with similar performance, in terms of file transfer speed, to commercial solutions. As the storage capacity can only be scaled by increasing the number of users, this won't be an objective of this thesis. Also, as IPFS-Drive is intended to be a proof-of-concept, the focus will be entirely on building a reliable and functional system and not at all on creating user incentives, such as digital tokens, in order to create a storage market. This should be a topic for the further extension of the proposed application, possibly as a layer on top of it.

Chapter 2

Background and Related Work

2.1 IPFS

The InterPlanetary File System (IPFS) [1] is a peer-to-peer distributed file system aiming to replace HTTP and connect all computing devices with the same system of files. IPFS uses the BitSwap (2.1.3) protocol for file transferring across the network. Any content added to IPFS has an associated unique hash and thus, the content is identified in the network by this hash rather then by its location as in HTTP.

Because IPFS uses peer-to-peer architecture, it has no single point of failure. IPFS incorporates various technologies from previous peer-to-peer systems or related projects, such as Git, BitTorrent or Distributed Hash Tables (DHT).

2.1.1 Identities

Each node in IPFS has assigned an unique identity, which is the cryptographic hash of their public key. Whenever connecting to other node, trust must be established, thus, upon the first connection, it takes place an exchange of public keys. If the key matches the node's id, then a connection is done, otherwise it is dropped.

2.1.2 Routing

Because of IPFS being a peer-to-peer system, all the nodes have the same privilege. As nodes constantly need to discover other peers in the network and find the peers which have a particular content, a routing system is required. This is achieved through Distributed

Hash Tables (DHT) [2], a concept used for coordinating and maintaining data in peer-to-peer systems. A distributed hash table is a type of decentralized distributed systems providing easy lookup of data in a similar manner to a regular hash table, by storing key - value pairs. A detailed explanation of DHTs can be found in [2], Chapter 2.6.2. The maintaining of the table is distributed among all the nodes such that if there is a change in participants, the disruption is minimal. IPFS uses a Distributed Sloppy Hash Table (DSHT) implementation, which provides efficient lookup through massive networks, low coordination overhead and resistance to attacks by preferring long lived nodes.

2.1.3 BitSwap - File Exchange

Using a protocol similar to BitTorrent, data is distributed in IPFS by peers exchanging blocks of data. This protocol is called BitSwap and it works by peers having a set of blocks they seek to acquire, called *want list*, and a set of blocks they have to offer in exchange, called *have list*. Blocks are binary data structures containing a certain amount of a file's data. As any file can be split up in several blocks, it is possible to get the entire file, block by block, from several nodes at once.

2.1.4 Object Merkle DAG

The DHT and BitSwap provide IPFS with the necessary means to be a massive peer-to-peer network for distributing and storing IPFS objects. On top of this there is a Merkle Directed Acyclic Graph, formed by the totality of objects in IPFS, where objects are linked through cryptographic hashes. The Merkle DAG provides IPFS with ways to:

- Address content by identifying it with its unique mulithash checksum.
- Make sure that anything transferred in IPFS is verified with its checksum. IPFS detects any data that is tampered with or corrupted.
- Deduplicate data by storing equal objects only once if they have the exact same content.

An IPFS object is a data structure with two fields: *links* and *data*. A single IPFS object can store up to 256 KB of data. When the data size is greater than 256 KB, the file is split up in chunks of 256 KB each, except for the last one. An empty parent object will then link all the pieces of the file together. Listing 2.1 shows the IPFS Object format:

```
type IPFSLink struct {
    Name string
    // name or alias of this link
    Hash Multihash
    // cryptographic hash of target
    Size int
    // total size of target
}
type IPFSObject struct {
    links []IPFSLink
    // array of links
    data []byte
    // opaque content data
}
```

Listing 2.1: The IPFS Object format [1]

2.1.5 Naming

IPFS objects are represented by their Base58 encoded hash, which is also called CID (content identifier). All hashes begin with *Qm*, because the hash is in fact a multihash.A multihash specifies the hash function and the length of the hash in its first two bytes. An example is the IPFS object with hash *QmarHSr9aSNaPSR6G9KFPbuLV9aEqJfTk1y9B8pdwq K4Rq*. The first two bytes in hex is 1220, where 12 denotes that this is the SHA-256 hash function and 20 is the length of the hash in bytes - 32 bytes.

2.1.6 Local storage and pinning

IPFS uses a local repository in order to store and to retrieve data that is sent through the distributed file system. Thus, any block available to other nodes is stored in some node's local repository. When a node requests a file, they retrieve it from a different nodes and store it in their local repository as well. Within a short lookup period, multiple nodes have the possibility to deliver the specified content. However, as it is a local repository limited by the physical memory available on the node, content is not stored permanently. If a node deems an object to be of importance, that object can be pinned to ensure its survival. If an object is pinned it will stay in the local storage until it is no longer pinned. Otherwise, it will be deleted by the garbage collector. A pinned object is an object that the user has chosen to download to the local storage for faster access. Thus, IPFS does not have to keep fetching the object over the network when it needs to be accessed.

2.2 IPFS Cluster

IPFS Cluster [3] is a software used to orchestrate IPFS daemons running on different hosts, as well as a mean of automatically replicating and pinning content throughout an IPFS network of nodes.

An IPFS Cluster is formed by a number of Peers, each of them associated to one IPFS daemon. The peers share a pinset (also known as shared state) which lists the CIDs which are cluster-pinned and their properties (allocations, replication factor etc.).

Cluster peers communicate in a peer-to-peer manner similarly to IPFS, but separately from it. Thus, every cluster peer needs its own Private Key (different from the one used by the IPFS daemon) and has its own Peer ID. All peers share an additional secret key which ensures they can only communicate with known parties.

2.2.1 Establishing Consensus

Consensus is a fundamental problem in fault-tolerant distributed systems. Consensus involves multiple servers agreeing on values. Once they reach a decision on a value, that decision is final. Typical consensus algorithms make progress when any majority of their servers is available; for example, a cluster of 5 servers can continue to operate even if 2 servers fail. If more servers fail, they stop making progress (but will never return an incorrect result).

Consensus typically arises in the context of replicated state machines, a general approach to building fault-tolerant systems. Each server has a state machine and a log. The state machine is the component that should be fault-tolerant, such as a hash table. It will appear to clients that they are interacting with a single, reliable state machine, even if a minority of the servers in the cluster fail. Each state machine takes as input commands from its log.

2.2.2 The Raft Algorithm

IPFS Clutser is using Raft Algorithm [4] in order to achieve consensus over its pinset. Raft implements consensus by a leader approach. The cluster has one and only one elected leader which is fully responsible for managing log replication on the other servers of the cluster. It means that the leader can decide on new entries placement and establishment of data flow between it and the other servers without consulting other servers. A leader leads until it fails or disconnects, in which case a new leader is elected. Raft decomposes consensus into three sub-problems:

- **Leader election**: In Raft, a server can be in one of three different states, leader, candidate and follower. Raft uses an asymmetric approach to the consensus protocol, which means that there can only be one server acting as leader. A new leader needs to be elected in case of the failure of an existing one.
- **Log replication**: When the leader receives a command from a client, it must ensure that the command is replicated and securely stored on other servers before executing the command in its state machine.
- **Safety**: Raft makes sure that the leader for a term has committed entries from all previous terms in its log. This is needed to ensure that all logs are consistent and the state machines execute the same set of commands.

2.3 Related work

2.3.1 Filecoin

Built by the same company behind IPFS, Protocol Labs, Filecoin [5] is a decentralized storage network running as an incentive layer top of IPFS. As there is a lot of unused space in data stores or personal hard drives, Filecoin proposes a decentralized storage market, which runs on a blockchain. Each transaction on the ledger represents a payment made to a miner in order to store the client's data. Security is achieved through end-to-end encryption of the data at the client, while storage providers do not have access to decryption keys. Filecoin is still under development and there has been no product release until the moment of writing this thesis.

2.3.2 Storj

Storj [6] is a decentralized cloud storage platform built on the Ethereum network. The core technology behind it is peer-to-peer smart contracts for storage. This provides a way for a *renter* to purchase storage space from a *farmer* selling it, with both of them possibly not knowing each other. They negotiate an agreement and move data from the *renter* to the *farmer* for safekeeping. As in Filecoing, all the data is end-to-end ecrypted. The *farmer* must periodically prove cryptographically that it still has the data. The whole Storj network is build using a DHT, similar to IPFS, and works on the pub/sub model. To find a partner, a node can sign an incomplete contract and publish it to the network. Other nodes on the network can subscribe to certain types of contracts (i.e. types they might be interested in), as they can determine what contracts they're interested in and forward on contracts to other nodes they think might be interested.

2.3.3 Sia

Sia [7] offers access to decentralized cloud storage platforms for renters looking to make use of cheaper, faster means of using data centers that are open to anyone. Sia is based on an independent blockchain where cryptographic service level agreements are made between a storage renter and a provider. All the payments are done with Sia coin. At the end of a file contract, the host must provide a storage proof. As a host stores only a file segment, the proof is achieved by using Merkle trees (also used in IPFS) to demonstrate, that the segment actually belongs to the file. Before uploading to the host storage, files are divided into 30 segments, each targeted for distribution to hosts across the world. This distribution assures that no one host represents a single point of failure and reinforces overall network uptime and redundancy. The segments are created using the Reed-Solomon erasure coding, which makes it possible to divide files in a redundant manner, where any 10 of 30 segments can fully recover the user's files. Thus, if 20 out of 30 hosts go offline, the user is still able to recover his data. Each host only stores encrypted segments of user data.

Chapter 3

System Design

3.1 Requirements

To develop a system complying with the goals described in Chapter 1 there must be considered requirements of aspects such as the IPFS-Drive scenario and its use cases, general security considerations as well as IPFS specific considerations. Therefore, IPFS-Drive should account the following requirements:

Functional Requirements

- **Account management** with possibilities of authentication and registration based on e-mail and password
- **Storage** of files and directories for an account and under a certain directory *in* the IPFS network
- **Encryption** of all the data stored *in* the IPFS network and **decrypt** it when retrieving it from IPFS
- **Synchronization** of changes of files and directories for an account
- **User Interface**

The use case diagram showed in Figure 3.1 illustrates these functional requirements presented above.

Non-functional Requirements

- **IPFS** for object(i.e. files and directories) distribution and storage in the network
- **IPFS-Cluster** for physical object storage and pin management
- **Highly secure encryption algorithm**
- **Python 3**

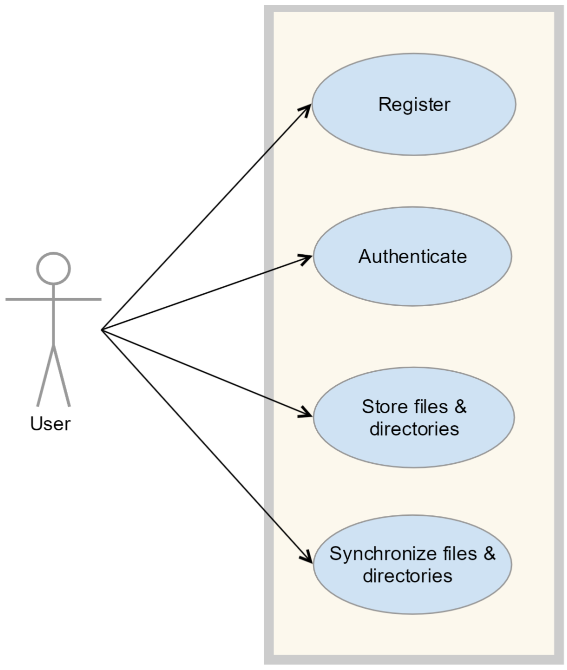


Figure 3.1: Use case diagram

3.2 Technologies

3.2.1 Firebase

Firebase [11] is a mobile and web application development platform that provides developers with multiple tools and services. For IPFS-Dirve there are going to be used the Realtime Database and Authentication.

Authentication

Most applications need to know the identity of a user. Knowing a user's identity allows an application to securely save user data in the cloud. Firebase Authentication provides backend services, easy-to-use SDKs, and ready-made UI libraries to authenticate users. It supports authentication using email and password, providing methods to create and manage users that use their email addresses and passwords to sign in. Firebase Authentication also handles sending password reset emails. The implementation is fast, as the entire authentication system can be set up in under 10 lines of code. Another advantage is that it associates for each account an unique identifier which can be used to retrieve user specific data from the database.

Realtime Database

The Firebase Realtime Database is a cloud-hosted NoSQL database that allows storage and synchronization between your users in real time and helps developers to build serverless applications. With a single API, the Firebase realtime database provides an application with both the current value of the data and any updates to that data. It can also integrate with Firebase Authentication to provide a simple and intuitive authentication process.

3.2.2 IPFS

When an IPFS node is running as a daemon, it exposes an HTTP API that allows controlling the node and running the same commands from the command line. In many cases, using this API this is preferable to embedding IPFS directly in the application. It allows maintaining peer connections that are longer lived than the application and keeping a single IPFS node running instead of several if the application can be launched multiple times. For the application, there following endpoints should be used:

- `/add` - Add a file or directory to IPFS.

  Method: GET

  Required arguments:

  - *path* - The path to a file to be added.

  Request body: Argument "path" is of file type. This endpoint expects a file in the body of the request as 'multipart/form-data'.

Response: On success, the call to this endpoint will return with 200 and the following body:

Listing 3.1: Add to IPFS response body

```
{
    "Name": "<string>",
    "Hash": "<string>",
    "Bytes": "<int64>",
    "Size": "<string>"
}
```

- `/get` - Downloads a file, or directory of files from IPFS. Files are placed in the current working directory.

  Method: GET

  Required arguments:

  - *cid* - The path to the IPFS object(s) to be outputted

  Response: On success, the call to this endpoint will return with 200 and a 'text/-plain' response body.

- `/object/patch/add-link` - Add a link to a given object.

  Method: GET

  Required arguments:

  - *root* - IPFS hash for the object being modified
  - *name* - Name for the new link
  - *ref* - IPFS hash for the object being linked to

  Response: On success, the call to this endpoint will return with 200 and the following body:

Listing 3.2: Add/remove link to/from IPFS object response body

```
{
    "Hash": "<string>"
    "Links": [
        {
            "Name": "<string>",
            "Hash": "<string>",
            "Size": "<uint64>",
        }
    ]
}
```

- `/object/patch/rm-link` - Remove a link from a given object.

  Method: GET

  Required arguments:

  - *root* - IPFS hash of the object to modify
  - *link* - Name of the link to remove

  Response: On success, the call to this endpoint will return with 200 and same body as showed in Listing 3.2.

3.2.3 IPFS Cluster

IPFS Cluster has two software components: *ipfs-cluster-service* and *ipfs-cluster-ctl.* This section will focus only on the first one, because, similar to IPFS nodes, it runs as a daemon exposing a REST API for communicating with the cluster. *ipfs-cluster-ctl* is just a command line interface client using this API. For IPFS-Drive, the following endpoints should be used:

- `/pins/{hash}` - Pin a CID to IPFS Cluster.

  Method: POST

  Response: On success, the call to this endpoint will return with 202 and a ‘text/-plain‘ response body.

When using the Raft consensus implementation, this involves: deciding which peers will be allocated the CID (which IPFS daemon will store it - receive the allocation), forwarding the pin request to the Raft Leader and committing the pin entry to the log. At this point, a success/failure is returned to the user, but cluster has more things to do: receiving the log update and modifying the shared state accordingly, as well as the local state, setting the pin status accordingly depending on the phase of the operation. [3]

- `/pins/{hash}` - Unpin a CID to IPFS Cluster.

  Method: DELETE

  Response: On success, the call to this endpoint will return with 200 and a ‘text/-plain‘ response body.

  The process is very similar to pinning described above. Removed pins are wiped from the shared and local states.

3.3 Synchronization

Synchronization between devices is achieved by storing remotely all the multihashes corresponding to the files and directories uploaded by the user to IPFS in the Firebase Realtime Database. Each user's data is identified in the database by the UID automatically generated at the account creation. Data is stored in the following format:

Listing 3.3: Data format

```
{
  "content": {
    <uid>: {
      <b64 encoded path>: <multihash>
    }
}
```

Firebase Realtime Database makes use of the EventSource / Server-Sent Events protocol, an API for creating an HTTP connection for receiving push notifications from a server. As the data changes, the server will send named events in the following structure:

- *event*: event name

- *data*: JSON encoded data payload

The structure of these messages conforms to the EventSource protocol. The server may send the following five event types: *put, patch, keep-alive, cancel,* or *auth_revoked.*

Thus, if the user is logged in the application on multiple computers, the changes on one computer will be sent to all the other computers where the application is running. This way, it is achieved real time synchronization between computers.

3.4 Security

Because IPFS is a public network, data protection is a key concern of the application. As already mentioned, the application will use end-to-end encryption for all the data that goes in and out of AES-256 symmetric cipher was chosen, which works based on a single encryption key. In addition to this, the application should have extra security layer for authentication and for the database, as all the multihashes are stored there.

3.4.1 Data encryption using AES-256

The Advanced Encryption Standard [8] is a symmetric block cipher used to protect classified information and is implemented in various software and hardware throughout to encrypt sensitive data. It is notorious for its use by the US government and it is categorized by NSA in its Suite B Cryptography set.

Symmetric block ciphers use the same key for encryption and decryption, thus, the the sender and the receiver must use the same secret key. AES operates on 128-bit blocks of data in multiple rounds. AES-256 is using cryptographic keys of 256 bits, to which correspond a number of 14 rounds. A round consists of several processing steps that include substitution, transposition and mixing of the input plaintext and its transformation into the final ciphertext output. AES represents data using 128-bit blocks arranged as 4x4 matrices, which are called *states*.

AES-256 encryption/decryption involves these four operations:

- **KeyExpansion** - Generates 15 *round keys* from the cipher key, as AES requires a separate 128-bit round key block for each round plus one more.

- **AddRoundKey** - Applies XOR bitwise operation in order to combine each byte from the AES state with a block of the round key computed at **KeyExpansion**.
- **SubBytes** - Replaces each byte of the state with another one, according to a lookup table called S-box.
- **ShiftRows** - Left shifts the last three rows of the state for a specific number of steps.
- **MixColumns** - Applies transformations on the columns of the AES state, combining the four bytes in each column.

In Figure 3.2 the pseudo code of the AES encryption algorithm is given. Decryption is done by using the same operations in the reverse order.

```
input  : Plaintext Block $ptxt_b$, Secret Key $sk$
output: AES state $state$
$state = InitState(ptxt_b, sk)$
$AddRoundKey(state, sk_0)$
for $i \leftarrow 1$ to $n_r - 1$ do
    $SubBytes(state)$
    $ShiftRows(state)$
    $MixColumns(state)$
    $AddRoundKey(state, key_i)$
end
$SubBytes(state)$
$ShiftRows(state)$
$AddRoundKey(state, key_{n_r-1})$
```

Figure 3.2: AES encryption algorithm [9]

3.4.2 Authentication security

Besides the unique user id associated to each user when a new e-mail based account is created, when a user signs in using Firebase Authentication, Firebase creates a corresponding id token that uniquely identifies them and grants them access to several resources, such as Realtime Database. These tokens are signed JSON Web Tokens that securely identify a user in a Firebase project and can be reused to authenticate the Realtime Database REST API and make requests on behalf of that user. They contain basic profile information for a user, including the user id string.

3.4.3 Database security

Firebase Realtime Database provides rules for controlling who has read and write access to the database. These rules exists on the Firebase servers and are enforced automatically at all times. Thus, any read or write request is completed only if the rules allow it.

Firebase Authentication integrates with the Firebase Realtime Database in order to allow controlling data access for each user. The Realtime Database Rules have a `auth` variable which is populated with the user's information when the user authenticates. This information includes their unique identifier (uid) as well as linked account data, such as an email address, and other info.

To enforce IPFS-Drive users to access only their own data, such read and write rules are defined for the database as well and displayed below in Listing 3.4. These rules take into account the data format defined in Listing 3.3.

Listing 3.4: Firebase Realtime Database rules

```
{
  "rules": {
    "users": {
      "$uid": {
        ".read": "$uid === auth.uid"
        ".write": "$uid === auth.uid"
      }
    }
  }
}
```

3.5 Storage and pinning

The way IPFS is designed, does not keep any data in the network as long as that specific data is not requested by other peer. When the user adds some data to IPFS, say a file, it is shredded into smaller chunks, hashed and linked together by a parent object corresponding to the file which in turn has a multihash. All these hashes will be available to the network, but unless the file is requested by other peers, it will remain only on the user's computer. In case the user's IPFS node goes down, even if other peers have the multihash of the file, because the user's node can't serve it, the requesting user won't get it. This presents a problem for the application as one of the purposes is to have high data availability and do not depend on a single node.

Two key design concepts for IPFS are pinning and garbage collection. They work hand in hand and are a good starting point for solving the problem. As in IPFS anything is identified by a multihash, every node has a collection of pinned multihashes. By default, anything added from a peer gets pinned in the peer, but also any other data from IPFS can be pinned in a node on request. The garbage collector will only remove from the local repository the multihashes that are not pinned, thus leaving pinned files intact and available for the network.

This leads us to the IPFS Cluster defined earlier which works on these specific principles, only that now the pinset is a collection shared by all nodes in the cluster. Having a cluster set-up helps us achieve the purpose and solve the problem stated earlier. The cluster behaves as a single node and the application communicates with it through a gateway set up on participating node. For any piece of data added to the network, the client will make a pin request to the cluster and because of its implementation, the peers of the cluster will get the data from the node and pin it. Because the cluster is set-up in a high availability and redundancy aware manner, data persistence is achieved in case the peer from which the data was uploaded gets off the network.

3.6 Architecture

The classic client-server architecture may seem suitable for this application, as each user's files must be tracked and also there must be an account management method. But, the available technology allows doing this in a serverless manner, without building a server from scratch. Thus, for IPFS-Drive the client should be able to authenticate and store data to Firebase. To exchange data with other peers in the network, the client establishes itself as an IPFS node through the IPFS daemon. IPFS Cluster is another key component, as it asures that the files pinned by the client remain *alive* in the IPFS network even if they have been deleted from the client's computer or he is offline. Therefore, the IPFS cluster is composed of IPFS nodes, which are as well connected to the IPFS network through the IPFS daemon. These components and their interaction are also illustrated in Figure 3.3 below.

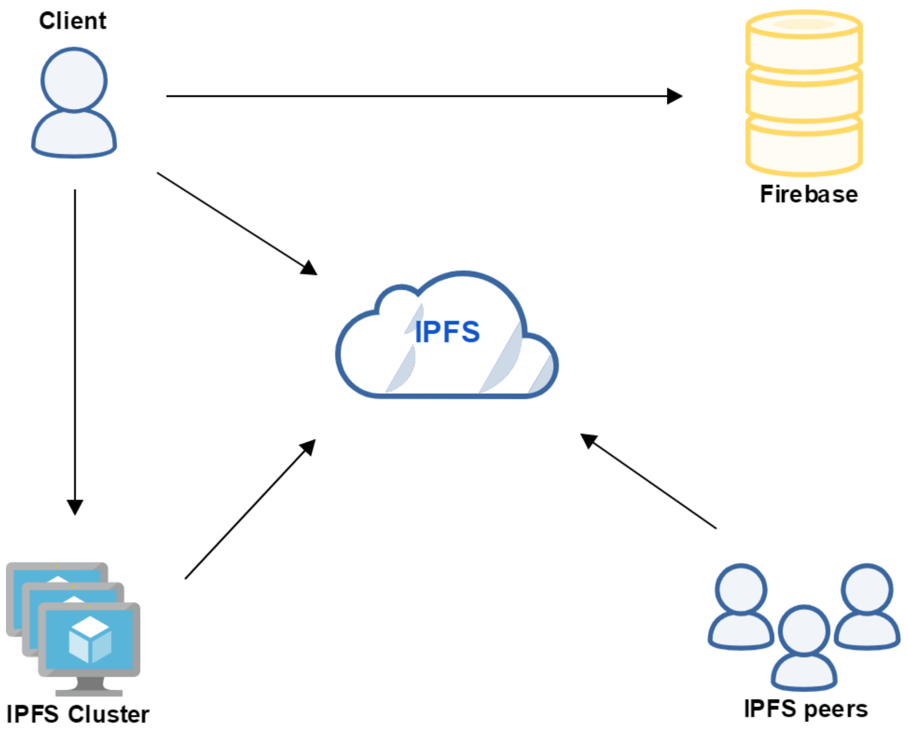


Figure 3.3: IPFS-Drive architecture

Chapter 4

Implementation

4.1 Tools

4.1.1 Programming language

The chosen programming language for developing this application is Python. The decision was made based on the familiarity of the author with the programming language, but most importantly, on the high availability of libraries for all the technologies used. Another advantage was the possibility to implement a simple, but yet very useful, GUI for the application using this programming language. Being an interpreted language and not having a parallel multi threading implementation due to the Global Interpreter Lock (the Python interpreter lock which can be used by one thread at a time, thus forcing the threads to run sequentially) [10], it might not be the suitable choice for an application targeting high performance. But, as for the application purpose, it is considered to be the best choice in regard to the aspects mentioned earlier.

4.1.2 IDE

After choosing the programming language, the next step consisted in choosing the suitable IDE. There are not many choices available for the chosen programming language, but after researching various options the best are: Visual Studio, Eclipse and PyCharm. The latest was chosen as it provides a very friendly and responsive user interface, it comes with a really useful debugger and it offers a free student subscription.

4.1.3 Version control

In order to maintain a healthy development process Git was used as a version control software. Furthermore, Github has been used as the online host for the repository.

4.2 Libraries

Besides the already existing standard libraries for Python, there were used a series of third party libraries for the technology used. Following, each of these libraries is going to be presented. Another library used for developing this application is Tkinter, Python's standard GUI programming toolkit. This is going to be presented as well, because of its importance in building the application.

firebase [12]

This library is the Python interface to the Google's Firebase REST API and provides the *Authentication, Database* and *Storage* classes with their specific methods, of which the application is using:

- **Authentication**
  - sign_in_with_email_and_password(email, password)
  - create_user_with_email_and_password(email, password)
- **Database**
  - child(name) - build paths to data
  - set(data) - set a value for a child
  - remove() - delete an entry
  - get() - return data from a path
  - stream(stream_handler) - listen to live data changes

watchdog [13]

Watchdog is a Python library that implements a transparent API to communicate with native operational system APIs and fall back to polling the disk periodically when needed. It supports APIs from BSD, Linux, Mac, and Windows.

ipfshttpclient [14]

IPFS HTTP Client is the Python library for interacting with the IPFS daemon. It essentially implements methods for accessing the REST API endpoints presented in 2.1. It also provides the *connect()* method to create a TCP connection to the IPFS daemon.

pycrypto [15]

PyCrypto is a collection of hash functions (such as SHA256 and RIPEMD160) and various encryption algorithms (AES, DES, RSA, ElGamal, etc.).

Tkinter [16]

Tkinter is a thin object-oriented layer on top of Tcl and Tk. Tcl (Tool Command Language) is a dynamic programming language, suitable for a very wide range of uses, including web and desktop applications. Tk is a cross-platform graphical user interface toolkit that provides a fast and easy way to develop rich desktop applications. Tk is the standard GUI for Tcl, but also for many other dynamic language. Tkinter is included with standard Python installs for Linux, Microsoft Windows and Mac OS X.

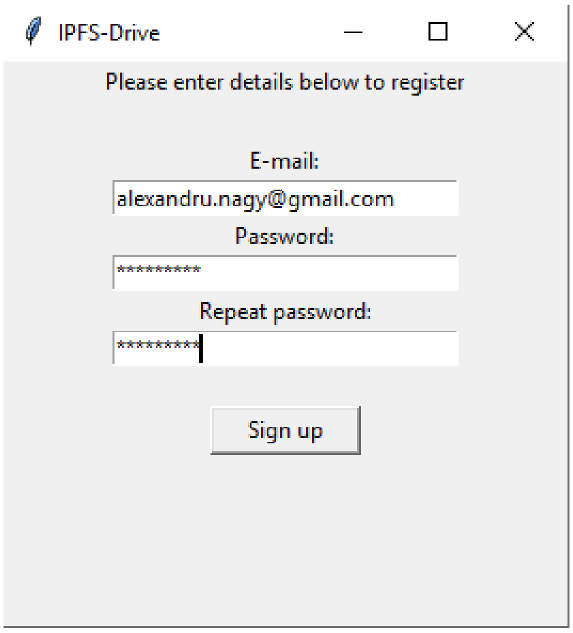
Tkinter works based on some basic concepts, each having a corresponding class which can be extended:

- **Top Level Window** - A window that exists independently on the screen. It is decorated with the OS specific frame and controls. *Tk* is the class that creates this top level window which usually is the main window of an application.
- **Widget** - Tkinter provides various controls, such as buttons, labels and text boxes. These controls are commonly called widgets. Each widget has its own class that implements it. There are four stages to create a widget:
  - creating it within a certain frame
  - configuring it based on its attributes
  - *pack*ing it in order to become visible
  - binding it to a function or event
- **Frame** - It is a widget working as a container responsible for arranging the position of other widgets . The homonym class is the one creating it.

4.3 Application

4.3.1 Registration and authentication

IPFS-Drive prompts the user to an authentication screen (Figure 4.1b) as soon as the application starts. If the user does not have an account already, he can go to the registration screen (Figure 4.1a) and sign-up. At this step the user is required to set up an account based on a valid e-mail address and a password. As soon as the user presses the *Sign up* button, a request for account creation is sent to the Firebase API. If the data introduced is correct (e-mail is valid and is not associated to another account already), the account is created in Firebase (Figure 4.2), the user is prompted with a success message box and then sent back to the authentication screen. Otherwise, the user will be prompted with the corresponding error message box. The flow for the sign in process is completed in a similar manner to registration. After the user enters his data and presses the *Sign in* button, the user is either identified in Firebase Authentication and he is signed in successfully, or he is presented a custom error depending on whether his email and password are valid or an account associated to the email exists.


IPFS-Drive

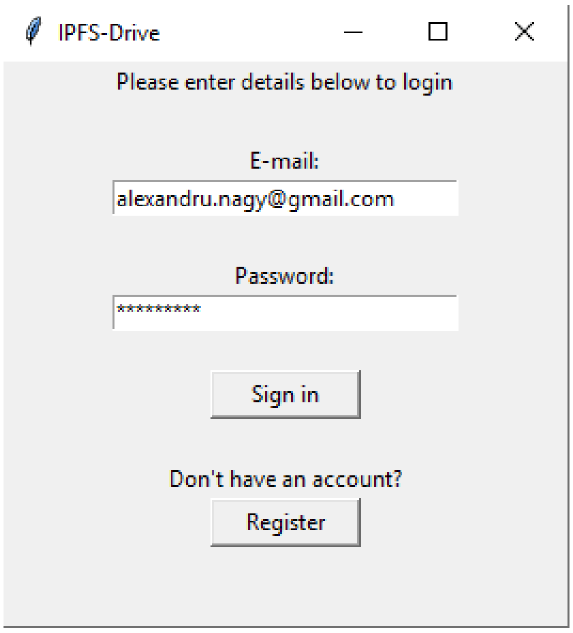
Please enter details below to register

E-mail: alexandru.nagy@gmail.com

Password: *********

Repeat password: *********

(a) Registration


IPFS-Drive

Please enter details below to login

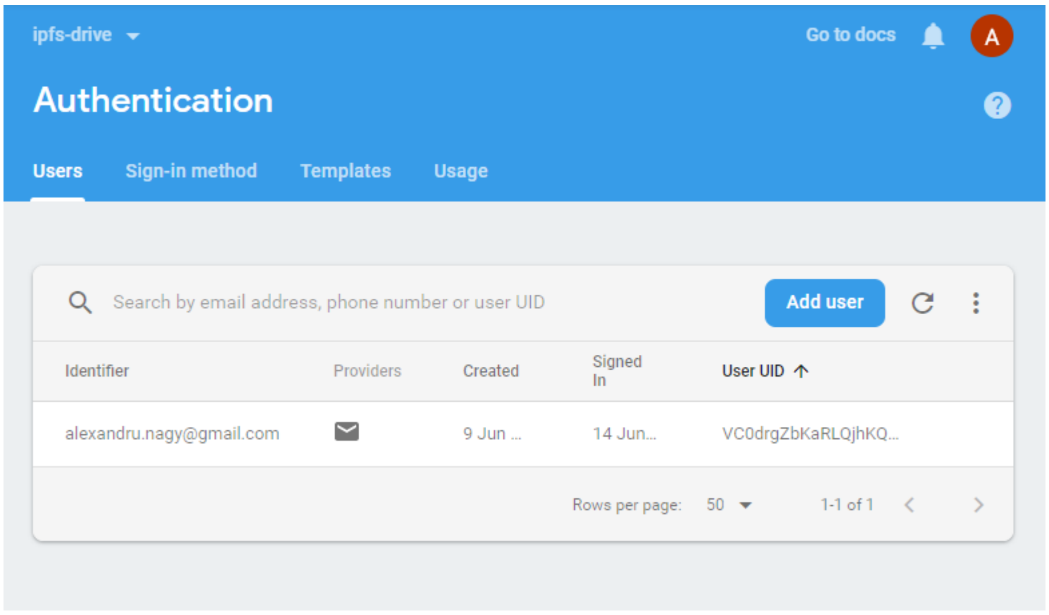
E-mail: alexandru.nagy@gmail.com

Password: *********

Don't have an account?

(b) Authentication

Figure 4.1



Identifier	Providers	Created	Signed In	User UID ↑
alexandru.nagy@gmail.com	✉	9 Jun ...	14 Jun...	VC0drgZbKaRLQjhKQ...

Figure 4.2: Firebase authentication

4.3.2 Encryption

After the user is signed in, the main screen of the application is displayed. Here, there are two fields: *Root directory, Encryption password*, two check boxes: *Add root directory to IPFS-Drive* ,*Synchronize* and a *Start* button. The user must carefully choose a strong encryption password, which must be entered in the *Encryption password field.* For security reasons, the application itself does not ever store this password and it is the sole responsibility of the user to keep it safe. The password will be used for all of the user's files, so a loss of the key will result in the inability to later decrypt his own data. This is the only way to avoid private data exposure, but, as for any other application, it may have flaws in case of an improper use and a lack of security measures. Another aspect to be considered, as for any other password, is its strength. Again, it remains the user's responsibility to set a strong encryption password. This encryption password is not going to be used *per se.* As the AES standard requires, this password is going to be hashed with the SHA-256 algorithm, generating an 32-bit key, which will be used as the encryption key for the user's files.

To avoid triggering any file system event, the encryption of files is going to be done in a hidden temporary *working directory* directory separate from the root directory provided by the user. Thus, the application does not modify any existing file when encrypting it.

Each file is read in 64-kb chunks and each chunk is encrypted with the algorithm specified at 3.4.1 and then written to a specific output file. Decryption works in the same manner, only that in this case the operations used for encryption are performed in the reverse order.

4.3.3 Storage

The *Root directory* field must be completed with the full path of the directory that needs to be backed up and monitored by IPFS-Drive. In order to backup the current content of the root directory, the user must check the*Add root directory to IPFS-Drive* checkbox. Otherwise, only files created after the application was launched will be backed up to IPFS-Drive. If the user wants to synchronize the root directory with the data already stored by him in the network at previous uses of the application, then he must check the *Synchronize* checkbox.

Following, it will be considered as an example the directory described in Figure 4.3, containing more children directories, some documents and pictures.

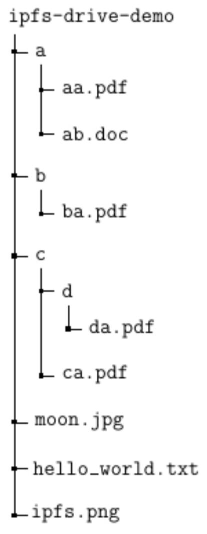
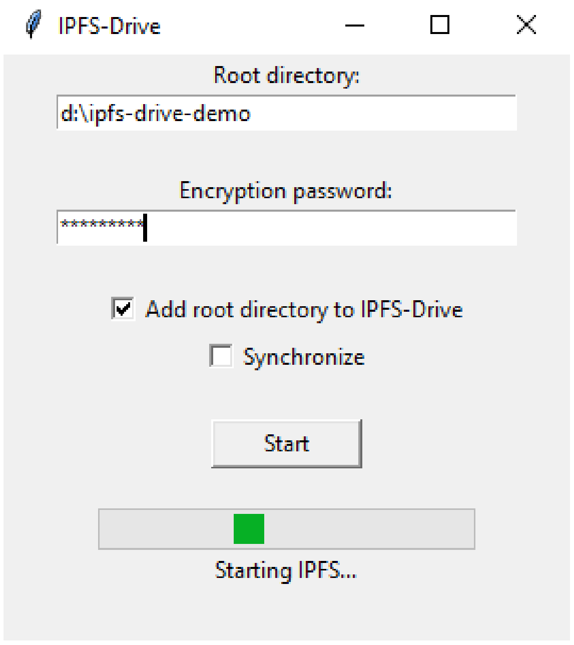


Figure 4.3: Root directory structure

Once the *Start* button is pressed, a progress bar which provides an animated display to let the user know that background work is progressing appears and the application starts the IPFS daemon as a separate process if it is not started already (Figure 4.4). In the latter case, the application is not be responsible for the process after the application is closed. In any other case, the application handles it as a child process and terminates it together with the application.



IPFS-Drive

Root directory: d:\ipfs-drive-demo

Encryption password: *********

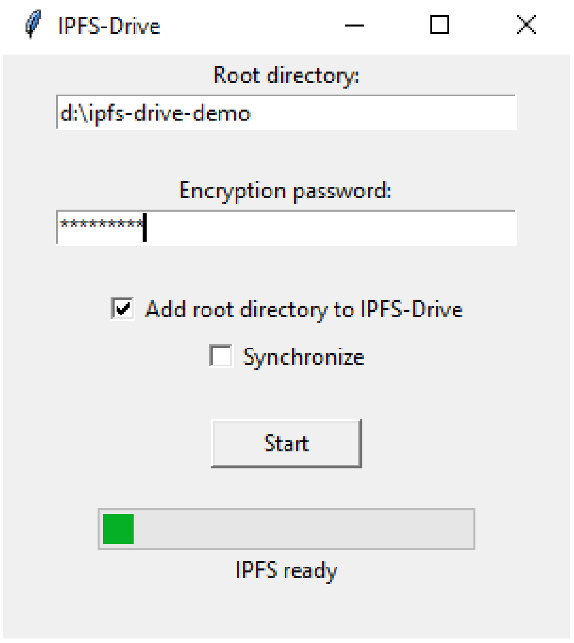
[✓] Add root directory to IPFS-Drive

[ ] Synchronize

Start

Starting IPFS...

(a)



IPFS-Drive

Root directory: d:\ipfs-drive-demo

Encryption password: *********

[✓] Add root directory to IPFS-Drive

[ ] Synchronize

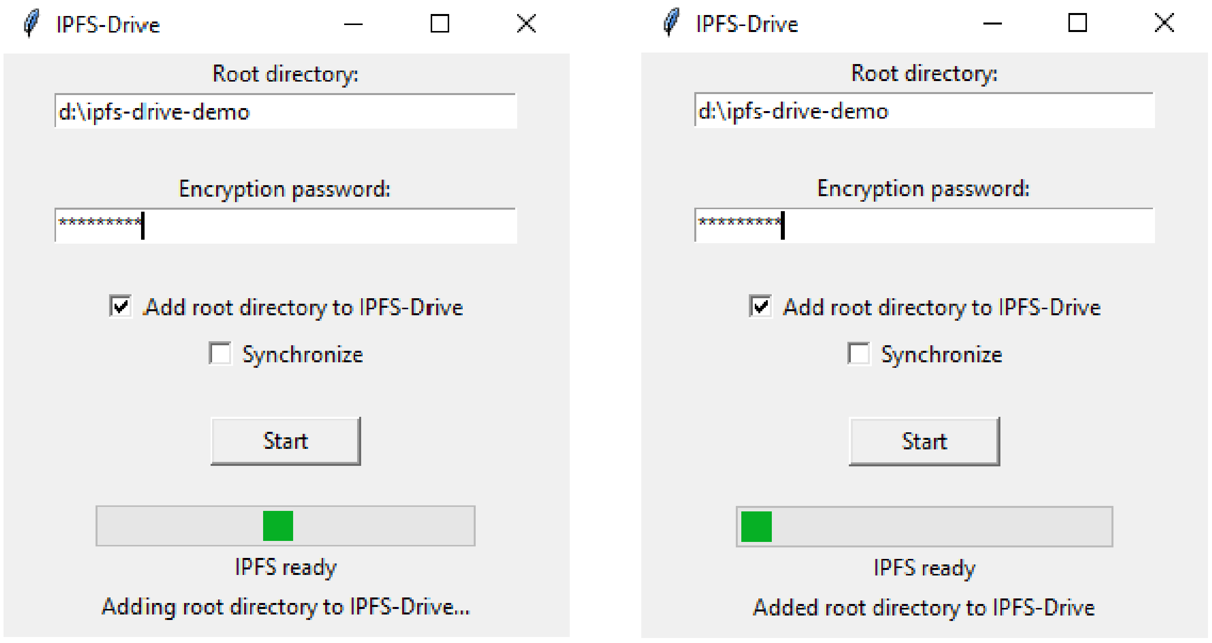
Start

IPFS ready

(b)

Figure 4.4: Starting IPFS daemon

If the *Add root directory to IPFS-Drive* check box is checked, the provided root directory content is encrypted and added to IPFS-Drive (Figure 4.5), following the same steps as showed in Figure 4.9. Each file resulted from the encryption operation will be stored separtely as mentioned. But, when uploaded to IPFS, it will act as a placeholder for the original file and thus seen in the database as it has been uploaded from the root directory. As soon as it is added to IPFS, the root directory content is visible in the Firebase Realtime Database (Figure 4.6), in the format specified in 3.3.


IPFS-Drive

Root directory:

d:\ipfs-drive-demo

Encryption password:

****

[✓] Add root directory to IPFS-Drive

[ ] Synchronize

IPFS ready

Adding root directory to IPFS-Drive...

IPFS-Drive

Root directory:

d:\ipfs-drive-demo

Encryption password:

****

[✓] Add root directory to IPFS-Drive

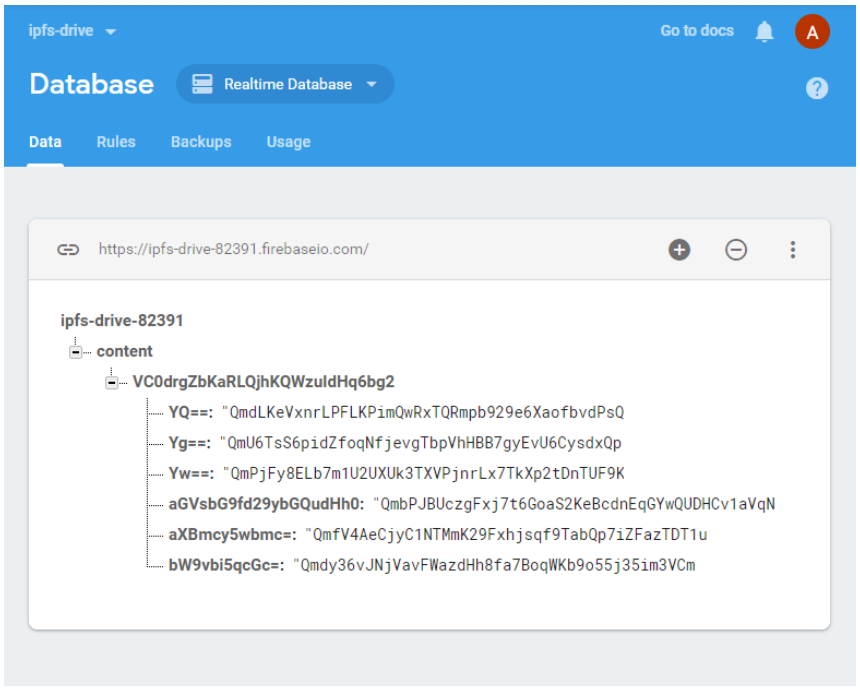
[ ] Synchronize

IPFS ready

Added root directory to IPFS-Drive

(a) (b)

Figure 4.5: Adding root directory to IPFS-Drive



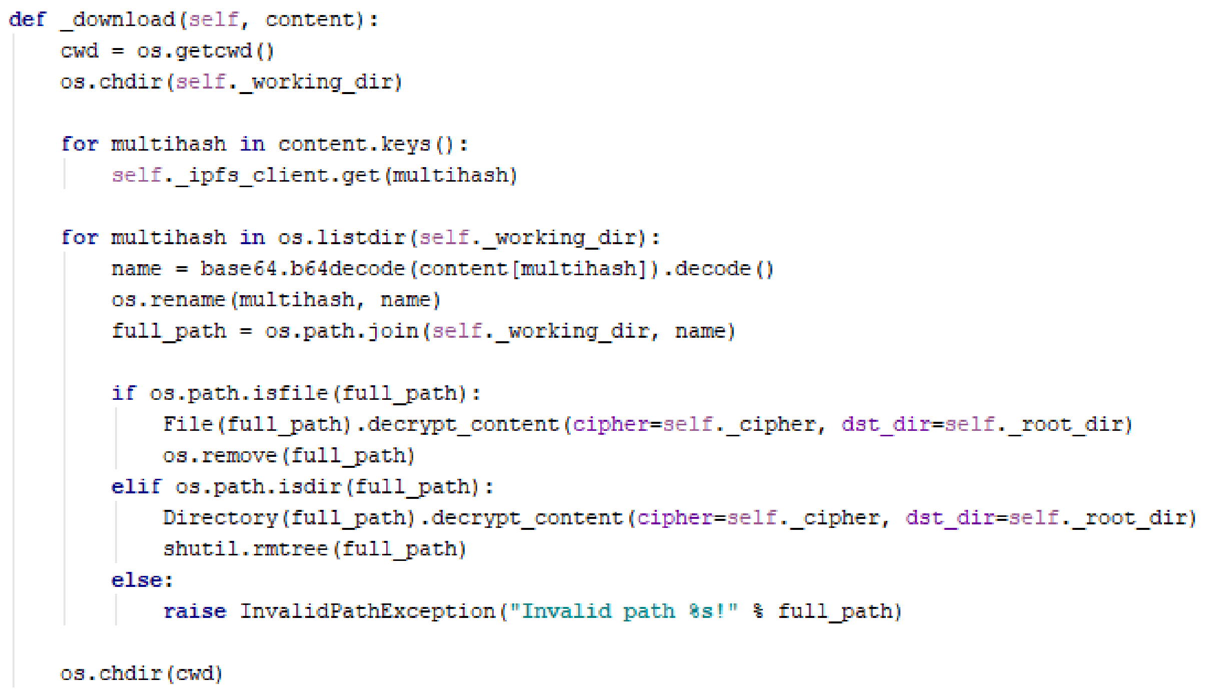
https://ipfs-drive-82391.firebaseio.com/

```
ipfs-drive-82391
 content
  VC0drgZbKaRLQjhKQWzuldHq6bg2
   YQ==: "QmdLKeVxnrLPFLKPimQwRxTQRmpb929e6XaofbvdPsQ
   Yg==: "QmU6TsS6pidZfoqNfjevgTbpVhHBB7gyEvU6CysdxQp
   Yw==: "QmPjFy8ELb7m1U2UXUk3TXVPjnrLx7TkXp2tDnTUF9K
   aGVsbG9fd29ybGQudHh0: "QmbPJBUczgFxj7t6GoaS2KeBcdnEqGYwQUDHCv1aVqN
   aXBmcy5wbmc=: "QmfV4AeCjyC1NTMmK29Fxhjsqf9TabQp7iZFazTDT1u
   bW9vbi5qcGc=: "Qmdy36vJNjVavFWazdHh8fa7BoqWKb9o55j35im3VCm
```

Figure 4.6: Firebase real-time database content

4.3.4 Synchronization

When the user chooses to synchronize and thus, checks the *Synchronize* check box before starting IPFS-Drive,the first step the application does after starting the IPFS daemon is to retrieve all the multihases from the database and request them from the network. Depending on the data replication on different nodes in the network, peer availability and geographical node distribution, the download time may vary. All the user's data is downloaded into a temporary *working directory*, and decrypted into the directory provided by the user as the root directory. The code snippet in Figure 4.7 presents in detail the download algorithm while the sequence diagram in Figure 4.8 shows the entire interaction between all the involved components. After this is done, the application proceeds to the next step of listening for any events from Firebase Realtime Database, as described in Section 3.3. Whenever an event is received, it is handled in a separate thread, such that it won't block the application in the meanwhile.



```python
def _download(self, content):
    cwd = os.getcwd()
    os.chdir(self._working_dir)

    for multihash in content.keys():
        self._ipfs_client.get(multihash)

    for multihash in os.listdir(self._working_dir):
        name = base64.b64decode(content[multihash]).decode()
        os.rename(multihash, name)
        full_path = os.path.join(self._working_dir, name)

        if os.path.isfile(full_path):
            File(full_path).decrypt_content(cipher=self._cipher, dst_dir=self._root_dir)
            os.remove(full_path)
        elif os.path.isdir(full_path):
            Directory(full_path).decrypt_content(cipher=self._cipher, dst_dir=self._root_dir)
            shutil.rmtree(full_path)
        else:
            raise InvalidPathException("Invalid path %s!" % full_path)

    os.chdir(cwd)
```

Figure 4.7: The *download* method of the *Sync* class

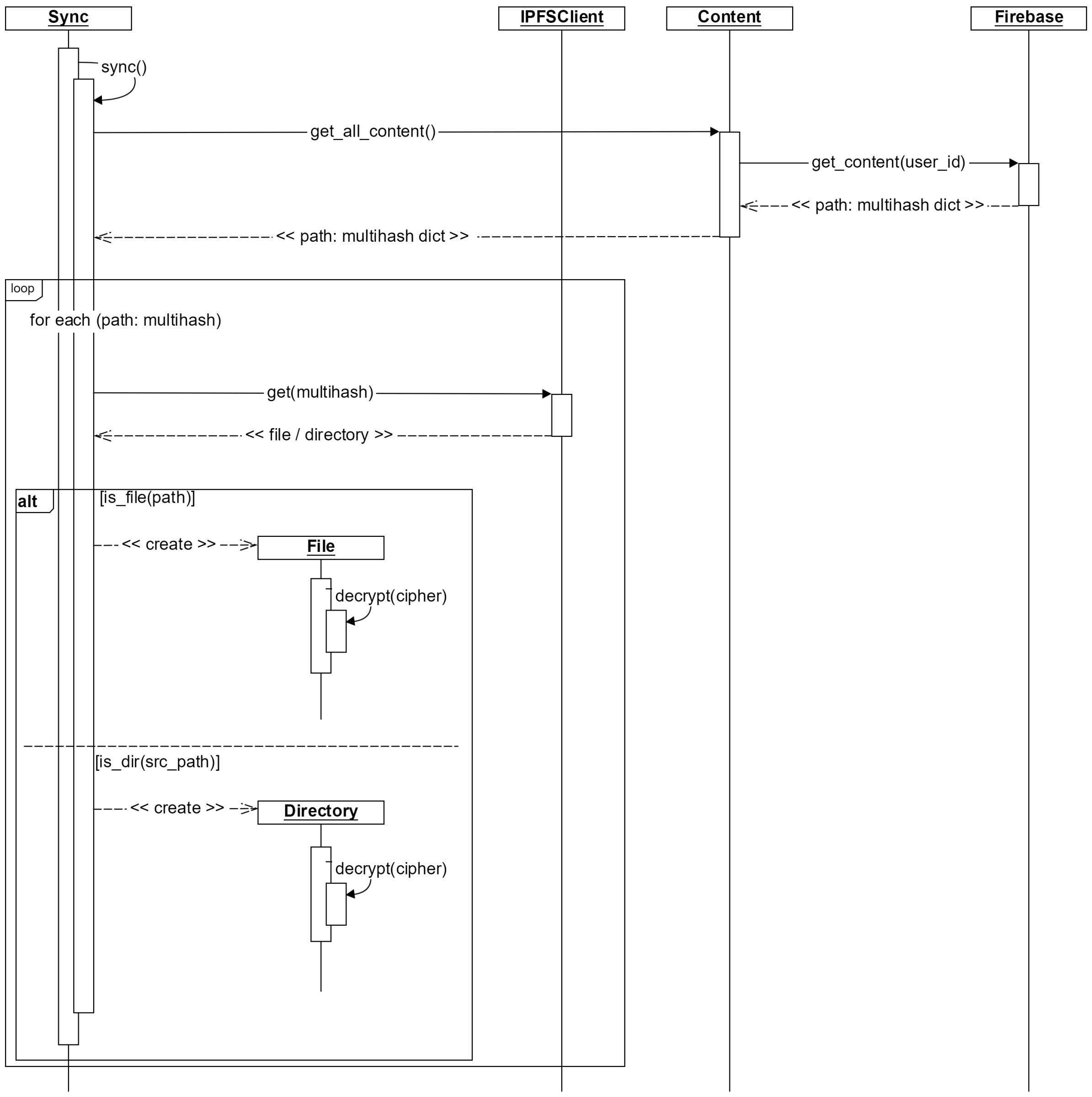


Figure 4.8: Synchronization sequence diagram

4.3.5 File system events monitoring

In order to keep track of everything that happens under the root directory provided by the user, file systems events monitoring is necessary. All of the most used operating systems provide a specialized API for monitoring such events. The *watchdog* library, which was briefly presented earlier (4.2), provides the *EventHandler* class, which can be extended in order to implement specific handlers for the following events:

On created

This event is triggered whenever a new file or directory is created. As it can be seen in Figure 4.9, it generates a whole sequence of actions, such that the newly created entity will be added to IPFS-Drive. It can be seen that the steps are slightly different depending on whether a file was created or a directory, but there can be delimited the big steps: encryption, adding to IPFS, adding the multihash(es) to the local content manager and afterwards to the database and finally pinning the multihash(es) to the IPFS cluster.

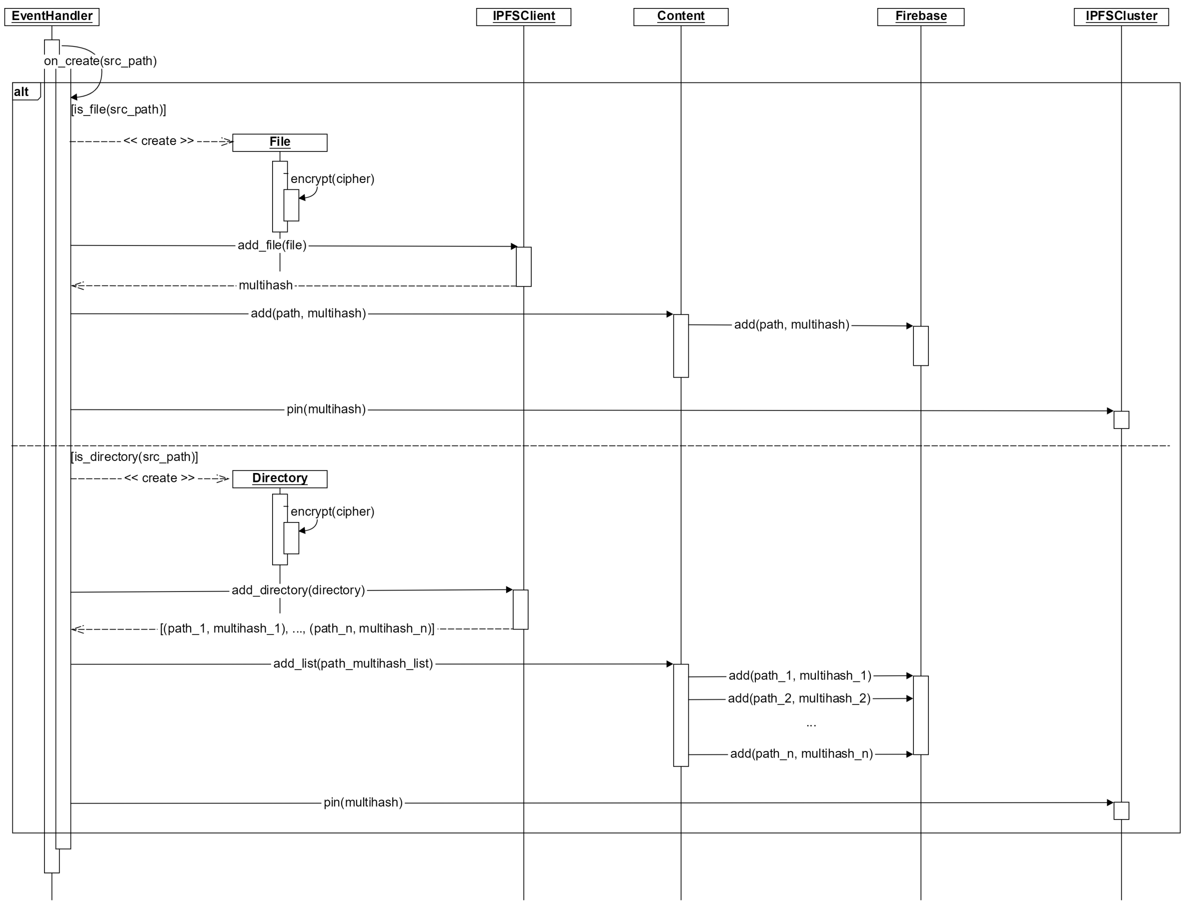


Figure 4.9: On created file/directory

On deleted

This event is triggered whenever a new file or directory is deleted. Figure 4.10 shows the sequence of actions generated by this event. First, if, exists, the multihash of the parent directory is taken from the local content manager. Based on this and on the multihash of the deleted object, the IPFS client will remove the link from the parent object to the removed object and create a new mulithash for the parent. The parent is updated in the local cache and in the database with the new multihash. Then, the object is also removed from the local cache and from the database. Finally, the multihash is unpinned from the IPFS cluster as it is no longer needed.

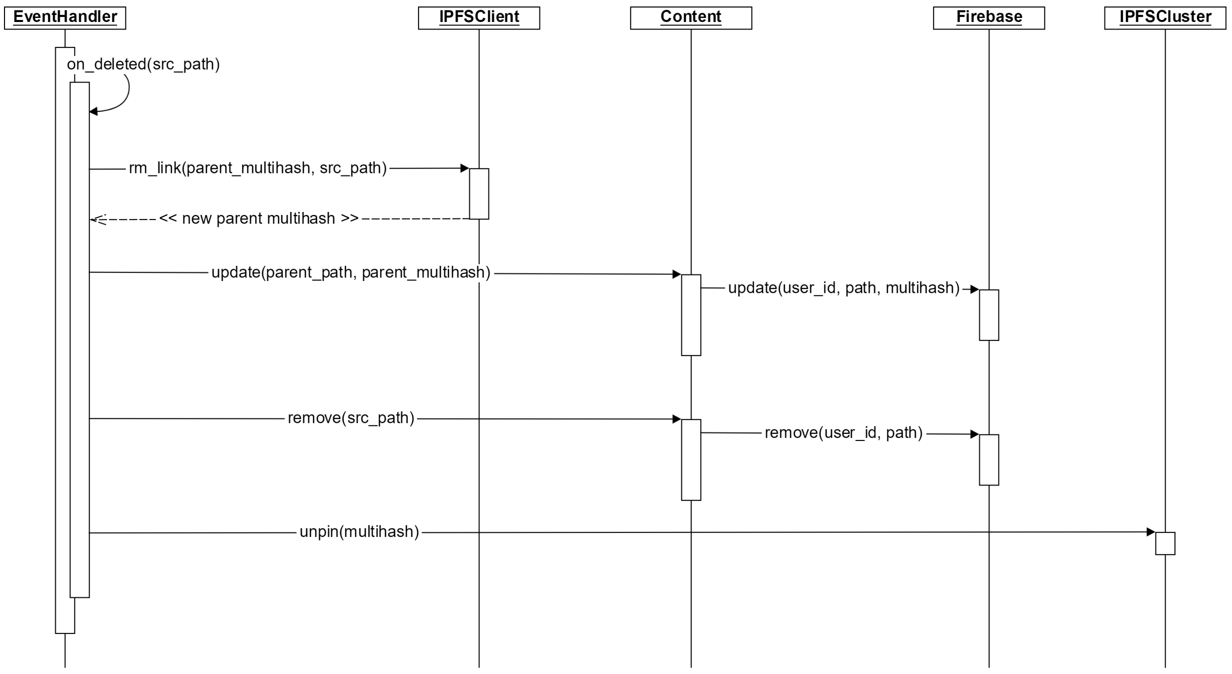

EventHandler
IPFSClient
Content
Firebase
IPFSCluster
on_deleted(src_path)
rm_link(parent_multihash, src_path)
<< new parent multihash >>
update(parent_path, parent_multihash)
update(user_id, path, multihash)
remove(src_path)
remove(user_id, path)
unpin(multihash)

Figure 4.10: On deleted file/directory

On modified

This event is triggered whenever a new file or directory is modified. The modified object is handled in the same way as if it has been created; but further, its new multihash is linked to its parent in IPFS, while the multihash corresponding to the old object is removed.

On moved

This event is triggered whenever a new file or directory is moved and takes into account both the source and the destination. The destination is treated as the creation path, so the flow will be the same as for the *on created* event, while the source path is handled as the removal path, thus it will be handled as the *on deleted* event.

4.3.6 Deployment

IPFS-Drive can be safley deployed on Windows, Mac OS and Linux, as it is an application written entirely in Python using only cross-platform libraries. In order to do this, *PyInstaller* [17] is going to be used. *PyInstaller* is a tool which converts Python applications with their dependencies into executable packages. Thus, the application can be run without installing a Python interpreter or any other modules. *PyInstaller* supports Python 2.7 and Python 3.4+ and is tested against Windows, Mac OS X, and Linux. However, there must be created a separate executable for each platform by running *PyInstaller* on that specific platform.

Chapter 5

Evaluation

5.1 Test setup

For running the IPFS-Drive application as a client, the test setup uses two computers in the same 100MB local network, both running Windows 10 and each having a 3.10GHz Intel CPU, with 16 GB RAM. Both computers are configured in order to run the *go-ipfs* implementations of IPFS and thus be peers in the network. They will be referred to as *Peer1* and *Peer2*.

In order to set-up the cluster and also achieve high availability, there was needed a platform to host three virtual machines, each acting as a cluster peer. Of all the platforms which provide such services at a good quality and a reasonable price, namely Amazon Web Services, Google Cloud and Microsoft Azure, the latest seemed to be the most suitable. Azure offers a student account with free credit for one year, thus allowing for experimenting with various configurations. The IPFS cluster is made by 3 virtual machines hosted on a Microsoft Azure server located in Central US, each having 1 vCPU, 2 GB RAM and running CentOS7. This seems to be the minimum configuration needed in order to run the cluster with high availability. Any less resources causes the machines to run out of memory very fast and thus failing. The configuration of these machines with IPFS and IPFS cluster software is explained in detail in Appendices A and B.

5.2 Test run

The test run is initiated by starting IPFS-Drive on *Peer1* and *Peer2*. The following steps serve both as a test run and as a complete walk through for IPFS-Drive. Some of the steps will contain information already presented as an example during Chapter 5. There are four testcases delimited, together with their outcomes, which make up this test run.

Start IPFS-Drive with *Add root directory to IPFS-Drive* option on

1. *Peer1* creates an account for the e-mail *alexandru.nagy@gmail.com* (Figure 4.1a). The account is then visible in Firebase Authentication (Figure 4.2).

2. *Peer1* and *Peer2* authenticate to IPFS-Drive using the account created at the previous step (Figure 4.1b).

3. *Peer1* sets the *root directory* to `d:/ipfs-drive-demo` and the *encryption password* to `ipfsdrive`. `d:/ipfs-drive-demo` is a directory having the structure presented in Figure 4.3.

4. *Peer1* checks only the *Store root directory to IPFS* checkbox and starts IPFS-Drive.

    (a) IPFS daemon is started (Figure 4.4). *Peer1* is connected to the IPFS network.

    (b) Each file and directory at the first level of `d:/ipfs-drive-demo` is encrypted with the *encryption password* and added to IPFS (Figure 4.5).

    (c) All files and directories at all levels under the *root directory* have now an associated multihash stored in the local cache. The first level file's and directory's multihashes are visible in the Firebase Realtime Database (Figure 4.6) and are pinned in the IPFS Cluster.

    (d) The IPFS-Drive running on *Peer1* listens for file system events under the *root directory*.

Start IPFS-Drive with *Synchronize* option on

5. *Peer2* sets the *root directory* to `d:/ipfs-drive-demo` and the *encryption password* to `ipfsdrive`. On this computer `d:/ipfs-drive-demo` is an empty directory.

6. *Peer2* checks only the *Synchronize* checkbox and starts IPFS-Drive.

   (a) IPFS daemon is started (Figure 4.4). *Peer2* is connected to the IPFS network.
   (b) All the multihashes from Firebase Realtime Database associated to the account *Peer2* is authenticated with are retrieved and downloaded from IPFS.
   (c) All downloaded objects are decrypted using the *encryption password* and renamed to their associated name in the database (i.e. their name on the *source* computer - *Peer1*).
   (d) *Peer2* has the same content as *Peer1* under `d:/ipfs-drive-demo`.
   (e) The IPFS-Drive running on *Peer2* listens for file system events under the *root directory.*
   (f) The IPFS-Drive running on *Peer2* listens for changes in the Firebase Realtime Database.

Create file under the root directory monitored by IPFS-Drive

7. The file `d:/ipfs-drive-demo/a/test.txt` containing the text "This is a test" is created on *Peer1.*
   (a) *On created* event is triggered.
   (b) The file is encrypted and stored on IPFS
   (c) The files's multihash is added to its parent IPFS object (corresponding to `d:/ipfs-drive-demo/a` whose multihash is updated in the database, and pinned to the IPFS cluster.
8. *Peer2* is notified about the change in the database, retrieves the objects's name and multihash from the database, downloads it from IPFS and decrypts it.

Remove file under the root directory monitored by IPFS-Drive

9. *Peer2* removes the file `d:/ipfs-drive-demo/a/test.txt`.
   (a) *On deleted* event is triggered.
   (b) The files's multihash is removed from its parent IPFS object (corresponding to `d:/ipfs-drive-demo/a` whose multihash is updated in the database, and pinned to the IPFS cluster.

10. *Peer1* is **not** notified about the change in the database, as it started IPFS-Drive without the *Synchronize* checkbox checked. Until the next application start, *Peer1* will still have `d:/ipfs-drive-demo/a/test.txt`.

5.3 Results

The previous test run confirms that IPFS-Drive complies to all the functional requirements, as there were gradually tested all the functionalities. In addition to this, there were also made performance measurements while running the testcases. For the first testcase, it was measured how long does it take for each file and directory to be stored to IPFS-Drive. This comprises of the time needed to be encrypted, added to IPFS, added to the database and then pinned to the cluster. For the second testcase, it was measured how long does it take for each file and directory to be synchronized, which includes getting it from the database, downloading it from IPFS and decrypting it. The results are presented in Table 5.1.

	a/	*b/*	*c/*	*moon.jpg*	*ipfs.png*	*hello_world.txt*
Size (MB)	0.440	0.395	0.855	2.10	0.384	0.000012
Storage time (sec)	1.5	0.56	0.81	0.66	0.58	0.51
Sync time (sec)	2.55	2.54	4.33	3.24	2.38	2.19

Table 5.1: Table to test captions and labels

Another set of measurements was done for file creation under the directory monitored by IPFS-Drive. There was measured the amount of time needed for a file to reach another computer using IPFS-Drive with the *Synchronize* option on, since the moment it is created or added to the monitored directory, similar to the third test case presented before. Using the same test setup, there were done 10 iterations for files of size 1, 10, 50 and 100 MB. The results are showed in the graphic from Figure 5.1. It can be seen that IPFS-Drive performs file transfers with low latency and with very acceptable variations between test iterations for same file.

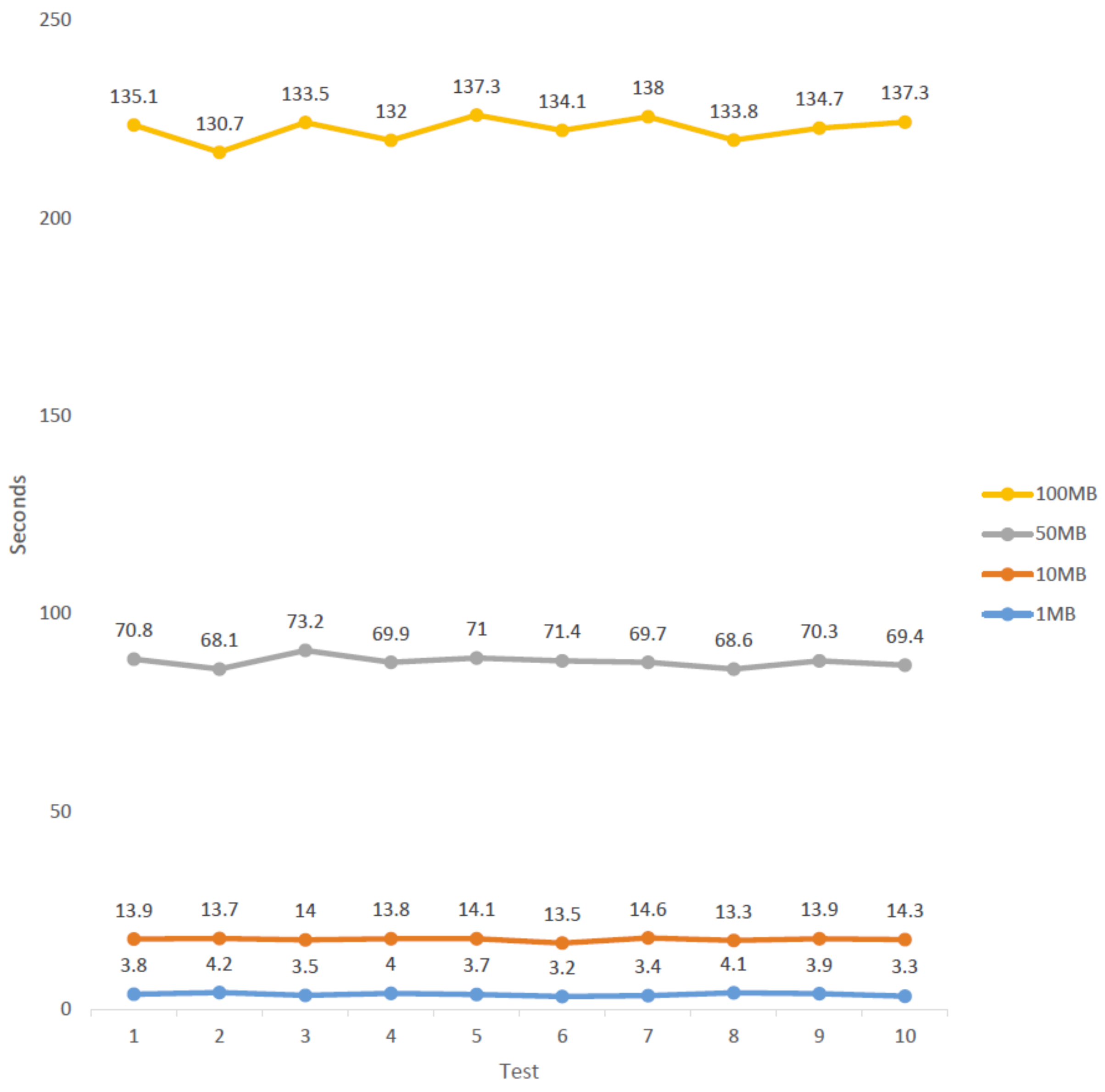


Figure 5.1: Performance testing graphic

Chapter 6

Conclusion

Built on top of existing technologies, IPFS-Drive is aimed at providing a reliable and secure solution for data storage and synchronization in a peer-to-peer network. IPFS is the main underlying structure on which this system is based, which was chosen because of its innovative approach in combining different technologies that already existed and were proved to be efficient. This approach has the potential to disrupt the internet as it is to day and assure the transition from IP based location addressing to hash based content addressing.

Even though the goal of the thesis was reached and a functional solution was built, it still does not achieve complete decentralization and full content based addressing. The storage space for this application is provided by a cluster made up of IPFS nodes, rather than by separate IPFS nodes offering their free space for rent. This was done mainly to avoid switching the focus from to current goals to building an incentive based market. Therefore, the cluster only simulates an existing storage agreement between the IPFS-Drive peer and those in the cluster, but all the underlying data exchange between peers still happens through IPFS, so in a peer-to-peer network.

This work should only be regarded as a proof-of-concept, but, with new steps made forward everyday on the path of decentralization, it has the potential to be extended with extra layers such as a smart contract based storage market, similarly to what Filecoin [5] wants to achieve, a blockhain based identity verification [18] and a distributed database [19] for content management.

Bibliography

[1] Benet, Juan. *IPFS - Content Addressed, Versioned, P2P File System.* Protocol Labs (2014).

[2] Kurose, James F., and Keith W. Ross. *Computer networking: a top-down approach.* Addison Wesley (2011).

[3] *IPFS Cluster.* `cluster.ipfs.io`

[4] Ongaro, D., and J. Ousterhout. *In search of an understandable consensus algorithm.* Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference , pp. 305-319. (2014).

[5] Benet, J., and N. Greco. *Filecoin: A decentralized storage network.* Protocol Labs (2018).

[6] Wilkinson, S., T. Boshevski, J. Brandoff, and V. Buterin. *Storj a peer-to-peer cloud storage network.* (2014).

[7] Vorick, D., and L. Champine. *Sia: Simple Decentralized Storage.* (2014).

[8] Daemen, J., and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard.* Springer Science & Business Media (2002).

[9] Daemen, J., and V. Rijmen. *AES Proposal: Rijndael.* (2001).

[10] Ramalho, L. *Fluent Python: clear, concise, and effective programming.* O'Reilly Media, Inc. (2015).

[11] *Firebase.* `firebase.google.com`

[12] *Firebase library.* `pypi.org/project/firebase`

[13] *Watchdog library.* `pypi.org/project/watchdog`

[14] *IPFS HTTP Client library.* `pypi.org/project/ipfshttpclient`

[15] *PyCrypto.* `pypi.org/project/pycrypto`

[16] *Tkinter library.* `wiki.python.org/moin/TkInter`

[17] *PyInstaller library.* `pypi.org/project/PyInstaller/`

[18] Jacobovitz, O. *Blockchain for identity management*, The Lynne and William Frankel Center for Computer Science Department of Computer Science. Ben-Gurion University, Beer Sheva (2016).

[19] McConaghy, T., Marques, R., Müller, A., De Jonghe, D., McConaghy, T., McMullen, G., Henderson, R., Bellemare, S. and Granzotto, A. *BigchainDB: a scalable blockchain database.* (2016).

Appendix A

IPFS installation & configuration

The installation begins by downloading the latest IPFS package, unpack it and move the binaries to a specific directory, in this case `/usr/local/bin`. The next step is to initialize an IPFS repository - the directory that hosts the IPFS configuration and datastore. The data store is the location where the node hosts its share of network data. Initializing a repository generates a node specific key pair and identity hash.

Initialising an IPFS repo can be done with `ipfs init`. This will generate an IPFS repository at `/.ipfs` with a standard default configuration file. Once initialised, this config file is saved as config in the repository root directory, available for updates and modificaitions whenever needed, as this configuration file determines a lot about the IPFS repository, ranging from bootstrap nodes, peer list, data storage options, CORS configuration, and more.

IPFS uses ports 4001 and 8080, so in order to avoid any communication problems, thr user must be sure that these ports are open and available. Running `ipfs daemon` will turn on the IPFS node and it will be running in the terminal window as long as it is open.

Appendix B

IPFS cluster installation & configuration

IPFS cluster has two components that need to be installed: *ipfs-cluster-service*, which runs a full cluster peer on top of IPFS and *ipfs-cluster-ctl*, which provides a command line interface to manage a running cluster peer. Both are installed in the same way as IPFS, so the binaries will be located as well at `/usr/local/bin`.

In the same manner as earlier for IPFS, the cluster service is initialized with the command `ipfs-cluster-service init`. This will create a directory at `/.ipfs-cluster` containing:

- *service.json* - The configuration file of the cluster;
- *raft* - A directory containing consensus data and snapshots of the cluster;
- *peerstore* -The list of cluster peers, which is cotinuously updated as new peers are added.

The secret key of a cluster is a 32-bit hex encoded random string, of which every cluster peer needs in their configuration file. Initiating a cluster will automatically generate a secret key which can then be obtained from *service.json.* This secret key must to be applied to all other peers configuration files in the cluster, such that the peers can recognize each other as being part of the same cluster.

For the cluster service, the ports 9094, 9095 and 9096 must be open. Running `ipfs-cluster -service daemon` will start the cluster service on one peer.

After the initial peer is set up, the other peers need to be configured following the exact same step. Only that in this case, the peers need to know to which cluster to adhere so they will be bootstrapped by the initial peer. This is achieved by running `ipfs-cluster-service daemon --bootstrap /ip4/`*<initial_peer_ip_address>*`/tcp/9096 /ipfs/`*<initial_peer_identity_hash>*.